3

A camada de enlace de dados

Neste capítulo, estudaremos os princípios de projeto da segunda camada em nosso modelo, a de enlace de dados. Neste estudo, trataremos de algoritmos que permitem uma comunicação eficiente e confiável de unidades de informação inteiras, chamadas quadros (em vez de bits individuais, como na camada física), entre dois computadores adjacentes. Por adjacentes queremos dizer que as duas máquinas estão fisicamente conectadas por meio de um canal de comunicação que funciona conceitualmente como um fio (p. ex., um cabo coaxial, uma linha telefônica ou um canal sem fio). A propriedade essencial de um canal que o torna semelhante a um fio é o fato de que os bits são entregues na ordem exata em que são enviados.

A princípio, você pode pensar que esse problema é tão trivial que não há nada para estudar – a máquina *A* simplesmente coloca os bits no fio e a máquina *B* os retira de lá. Infelizmente, os canais de comunicação algumas vezes produzem erros. Além disso, eles têm uma taxa de dados finita, e há um atraso de propagação diferente de zero entre o momento em que um bit é enviado e o momento em que ele é recebido. Essas limitações têm implicações importantes para a eficiência da transferência de dados. Os protocolos usados para comunicações devem levar todos esses fatores em consideração. Tais protocolos são o assunto deste capítulo.

Após uma introdução às principais questões de projeto presentes na camada de enlace de dados, começaremos nosso estudo dos protocolos verificando a natureza dos erros e como eles podem ser detectados e corrigidos. Em seguida, estudaremos uma série de protocolos de complexidade crescente e mostraremos como cada um deles resolve um número cada vez maior de problemas dessa camada. Por fim, concluiremos o capítulo com alguns exemplos de protocolos de enlace de dados.

3.1 QUESTÕES DE PROJETO DA CAMADA DE ENLACE DE DADOS

A camada de enlace de dados usa os serviços da camada física para enviar e receber bits pelos canais de comunicação (possivelmente não confiáveis), que podem perder dados. Ela tem diversas funções, entre as quais:

1. Fornecer uma interface de serviço bem definida à camada de rede (Seção 3.1.1).
2. Enquadrar sequências de bytes como segmentos autocontidos (Seção 3.1.2).
3. Detectar e corrigir erros de transmissão (Seção 3.1.3).
4. Regular o fluxo de dados de modo que receptores lentos não sejam atropelados por transmissores rápidos (Seção 3.1.4).

Para alcançar esses objetivos, a camada de enlace de dados recebe os pacotes da camada de rede e os encapsula em **quadros** para transmissão. Cada um contém um cabeçalho (header) de quadro, um campo de carga útil (payload), que conterá o pacote, e um final (trailer) de quadro, como mostra a Figura 3.1. O gerenciamento de quadros constitui o núcleo das atividades da camada de enlace de dados. Nas próximas seções, examinaremos em detalhes todas as questões mencionadas. Além disso, quando são usadas redes sem fio não confiáveis, o uso de protocolos para melhorar o enlace de dados posteriormente quase sempre melhora o desempenho.

Embora este capítulo trate explicitamente da camada de enlace de dados e de seus protocolos, muitos dos princípios que estudaremos aqui, como o controle de erros e o controle de fluxo, em algumas redes, são encontrados em

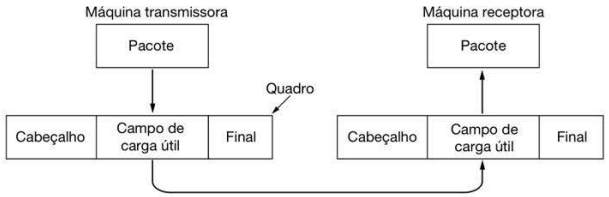

Figura 3.1 Relacionamento entre pacotes e quadros.

protocolos de transporte e também em outros protocolos. Isso ocorre porque a confiabilidade é um objetivo geral, e ela é alcançada quando todas as redes funcionam juntas. De fato, em muitas delas, essas funções são encontradas principalmente nas camadas superiores, com a camada de enlace de dados realizando a tarefa mínima, que é "suficientemente boa". Contudo, independentemente do lugar onde são encontrados, os princípios são quase idênticos. Na camada de enlace de dados, eles surgem com frequência em sua forma mais simples e mais pura, o que faz dessa camada um bom lugar para examiná-los em detalhes.

3.1.1 Serviços oferecidos à camada de rede

A função da camada de enlace de dados é fornecer serviços à camada de rede, sendo o principal transferir dados da camada de rede da máquina de origem para a camada de rede da máquina de destino. Na camada de rede da máquina de origem há uma entidade, chamada processo, que entrega alguns bits à camada de enlace de dados para transmissão ao destino. A tarefa da camada de enlace de dados é transmitir os bits à máquina de destino, de forma que eles possam ser entregues à camada de rede dessa máquina, como mostra a Figura 3.2(a). A transmissão propriamente dita segue o trajeto descrito na Figura 3.2(b); no entanto, é mais fácil pensar em termos de dois processos da camada de enlace de dados que se comunicam por intermédio de um protocolo de enlace de dados. Por essa razão, utilizaremos implicitamente o modelo da Figura 3.2(a) em todo este capítulo.

A camada de enlace de dados pode ser projetada de modo a oferecer diversos serviços. Os serviços reais oferecidos podem variar de um protocolo para outro. Três possibilidades razoáveis que consideraremos são:

1. Serviço não orientado a conexões sem confirmação.
2. Serviço não orientado a conexões com confirmação.
3. Serviço orientado a conexões com confirmação.

O serviço não orientado a conexões sem confirmação consiste em fazer a máquina de origem enviar quadros

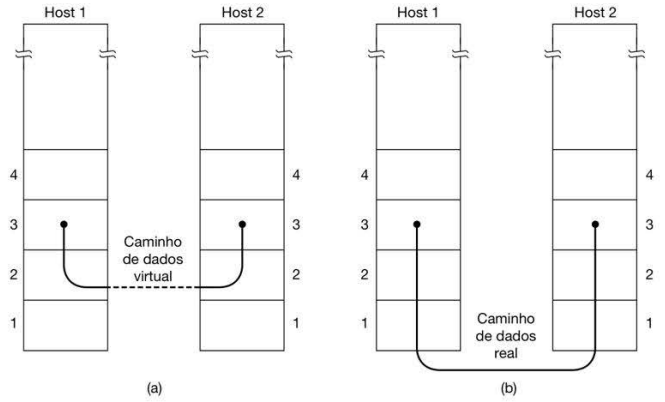

Figura 3.2 (a) Comunicação virtual. (b) Comunicação real.

independentes à máquina de destino, sem que esta confirme o recebimento deles. A Ethernet é um bom exemplo de uma camada de enlace de dados que oferece essa classe de serviço. Nenhuma conexão lógica é estabelecida antes ou liberada depois do processo. Se um quadro for perdido em decorrência de ruídos na linha, não haverá qualquer tentativa de detectar a perda ou de recuperá-lo na camada de enlace de dados. Essa classe de serviço é apropriada quando a taxa de erros é muito baixa, e a recuperação fica a cargo das camadas mais altas, bem como para o tráfego em tempo real, no qual, a exemplo da voz, os dados atrasados causam mais problemas do que os dados recebidos com falhas.

O próximo passo em termos de confiabilidade é o serviço não orientado a conexões com confirmação. Quando esse serviço é oferecido, ainda não há conexões lógicas sendo usadas, mas cada quadro enviado é confirmado individualmente. Dessa forma, o transmissor sabe se um quadro chegou corretamente ou não. Caso não tenha chegado dentro de um intervalo específico, o quadro poderá ser reenviado. Esse serviço é útil em canais não confiáveis, como os sistemas sem fio. O padrão 802.11 (WiFi) é um bom exemplo desse tipo de serviço da camada de enlace.

Talvez valha a pena destacar que oferecer recursos de confirmação no nível da camada de enlace de dados é uma questão de otimização, nunca uma exigência. A camada de rede sempre pode enviar um pacote e esperar que ele seja confirmado por seu par na máquina remota. Se a confirmação não chegar durante o intervalo do timer de retransmissão, o transmissor poderá enviar a mensagem inteira mais uma vez. O problema dessa estratégia é que ela pode ser ineficaz. Os enlaces normalmente têm um quadro com comprimento máximo estrito imposto pelo hardware e atrasos de propagação conhecidos. A camada de rede não conhece esses parâmetros. Ela pode enviar um pacote grande subdividido em, digamos, 10 quadros, dos quais dois são perdidos, em média. O tempo necessário para efetivar a transmissão do pacote com sucesso poderá ser muito longo. Em vez disso, se quadros individuais forem confirmados e retransmitidos, então os erros podem ser corrigidos mais diretamente e mais rapidamente. Em canais confiáveis, como a fibra óptica, o overhead de um protocolo de enlace de dados muito sofisticado talvez seja desnecessário, mas, em canais sem fio (com sua inerente falta de confiabilidade), o custo geralmente compensa.

Voltando aos nossos serviços, o serviço mais sofisticado que a camada de enlace de dados é capaz de oferecer à camada de rede é o orientado a conexões. Com ele, as máquinas de origem e de destino estabelecem uma conexão antes de qualquer dado ser transferido. Cada quadro enviado pela conexão é numerado, e a camada de enlace de dados garante que cada quadro seja, de fato, recebido. Além disso, essa camada garante que todos os quadros sejam recebidos uma única vez e na ordem correta. Assim, os serviços orientados a conexões fornecem aos processos da camada de rede o equivalente a um fluxo de bits confiável. Isso é apropriado para enlaces longos, não confiáveis, como um canal de satélite ou um circuito telefônico interurbano. Se o serviço não orientado a conexões com confirmação fosse usado, é possível imaginar que as confirmações perdidas poderiam fazer um quadro ser enviado e recebido várias vezes, desperdiçando largura de banda.

Quando o serviço orientado a conexões é usado, as transferências passam por três fases distintas. Na primeira, a conexão é estabelecida, fazendo ambos os lados inicializar as variáveis e os contadores necessários para controlar os quadros que são recebidos e os que não são. Na segunda, um ou mais quadros são realmente transmitidos. Na terceira e última fase, a conexão é desfeita, liberando-se as variáveis, os buffers e os outros recursos usados para mantê-la.

3.1.2 Enquadramento

Para oferecer serviços à camada de rede, a camada de enlace de dados deve usar o serviço fornecido a ela pela camada física. O que a camada física faz é aceitar um fluxo de bits brutos e tentar entregá-lo ao destino. Se o canal tiver ruído, como acontece na maioria dos enlaces sem fio e em alguns enlaces com fio, a camada física acrescentará alguma redundância aos seus sinais, para reduzir a taxa de erros de bits para um nível tolerável. Contudo, o fluxo de bits recebido pela camada de enlace de dados não tem garantia de estar livre de erros. Alguns bits podem ter valores diferentes e o número de bits recebidos pode ser menor, igual ou maior que o transmitido. A camada de enlace de dados é responsável por detectar e, se necessário, corrigir erros.

Em geral, a estratégia adotada pela camada de enlace de dados é dividir o fluxo de bits em quadros distintos, calcular um pequeno valor (um token), chamado de checksum (soma de verificação), para cada quadro e incluir essa soma de verificação no quadro quando ele for transmitido. (Os algoritmos de checksum serão discutidos mais adiante neste capítulo.) Quando um quadro chega a seu destino, o checksum é recalculado com base no que foi recebido. Se o valor recém-calculado for diferente do que está contido no quadro, a camada de enlace de dados saberá que houve um erro e tomará providências para lidar com ele (p. ex., descartando o quadro defeituoso e possivelmente também enviando de volta um relatório de erros).

A divisão do fluxo de bits em quadros é mais difícil do que parece à primeira vista. Um bom projeto deve tornar fácil para um receptor encontrar o início de novos quadros usando pouca largura de banda do canal. Nesta seção, examinaremos quatro métodos:

1. Contagem de caracteres.
2. Bytes de flag com inserção de bytes.
3. Bits de flag com inserção de bits.
4. Violações de codificação da camada física.

O primeiro método de enquadramento utiliza um campo no cabeçalho para especificar o número de bytes no quadro. Quando vê a contagem de caracteres, a camada de enlace de dados de destino sabe quantos bytes devem vir em seguida e, consequentemente, onde está o final do quadro. Essa técnica é mostrada na Figura 3.3(a) para quatro pequenos quadros, como exemplo, de tamanhos 5, 5, 8 e 8 bytes, respectivamente.

O problema com esse algoritmo é que a contagem pode ser adulterada por um erro de transmissão. Por exemplo, se a contagem 5 no segundo quadro da Figura 3.3(b) se tornar 7, em virtude da inversão de um único bit, o destino perderá a sincronização e não será capaz de localizar o início do quadro seguinte. Mesmo que o checksum esteja incorreto, de modo que o destino saiba que o quadro está defeituoso, ele ainda não terá informações suficientes para saber onde começa o quadro seguinte. Enviar um quadro de volta à origem solicitando retransmissão também não ajuda, pois o destino não sabe quantos bytes deverão ser ignorados para chegar ao início da retransmissão. Por essa razão, o método de contagem de caracteres quase não é mais usado.

O segundo método de enquadramento contorna o problema de ressincronização após um erro, fazendo cada quadro começar e terminar com bytes especiais. Normalmente o mesmo byte, chamado **byte de flag**, é usado como delimitador de início e de final, como mostra a Figura 3.4(a), na qual ele é representado por FLAG. Dois bytes de flag consecutivos indicam o final de um quadro e o início do próximo. Assim, se o receptor perder a sincronização, ele poderá simplesmente procurar dois bytes de flag para encontrar o final do quadro atual e o início do seguinte.

Entretanto, ainda existe um problema. É bem possível que o byte de flag ocorra nos dados, especialmente quando são transmitidos dados binários, como fotografias ou músicas. Essa situação poderia interferir no enquadramento. Uma forma de solucionar esse problema é fazer a camada de enlace de dados do transmissor incluir um caractere de escape especial (ESC) imediatamente antes de cada byte de flag "acidental" nos dados. Assim, o byte de flag de enquadramento pode ser distinguido daquele nos dados pela ausência ou presença de um byte de escape antes dele. A camada de enlace de dados na extremidade receptora remove o byte de escape antes de entregar os dados à camada de rede. Essa técnica é chamada de **inserção de bytes** (**byte stuffing**).

É claro que a próxima pergunta é: o que acontecerá se um byte de escape ocorrer em uma posição dentro dos dados? Nesse caso, ele também será preenchido com um byte de escape. No receptor, o primeiro byte de escape é removido, deixando o byte de dados seguinte (que poderia ser outro byte de escape ou o byte de flag). Alguns exemplos são mostrados na Figura 3.4(b). Em todos os casos, a sequência de bytes entregue após a remoção dos bytes inseridos é exatamente igual à sequência de bytes original. Ainda podemos procurar por um limite de quadro buscando dois bytes de flag em sequência, sem nos preocuparmos em desfazer os escapes.

O esquema de inserção de bytes representado na Figura 3.4 é uma ligeira simplificação do que é utilizado no protocolo ponto a ponto, ou **PPP** (**Point-to-Point Protocol**), comum na Internet para carregar pacotes por enlaces de comunicação. Descreveremos o PPP na Seção 3.5.1.

O terceiro método de delimitação do fluxo de bits contorna uma desvantagem da inserção de bytes, ou seja, o fato de ela estar ligada ao uso de bytes de 8 bits. O enquadramento também pode ser feito em nível de bit, de modo que os quadros podem conter um número qualquer de bits, compostos de unidades de qualquer tamanho. Ele foi desenvolvido para o então muito popular protocolo de controle de enlace de dados de alto nível, ou **HDLC** (**High-level Data Link Control**). Cada quadro começa e termina com um padrão de bits especial, 01111110, ou 0x7E em hexadecimal (um byte de flag). Sempre que encontra cinco valores

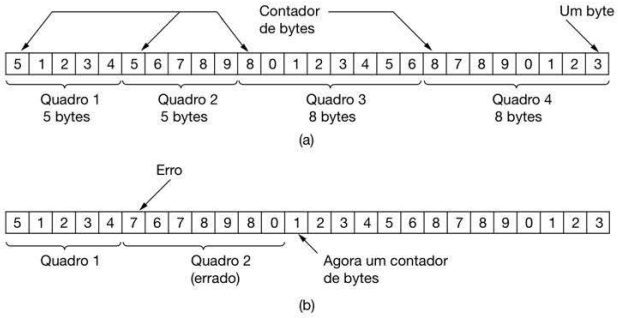

Figura 3.3 Um fluxo de caracteres. (a) Sem erros. (b) Com um erro.

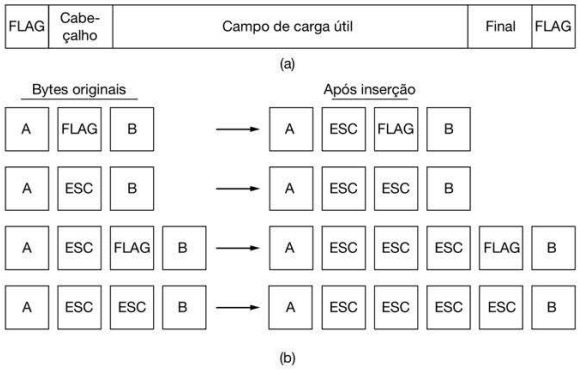

Figura 3.4 (a) Quadro delimitado por bytes de flag. (b) Quatro exemplos de sequências de bytes, antes e depois da inserção de bytes.

1 consecutivos nos dados, a camada de enlace de dados do transmissor automaticamente insere um bit 0 no fluxo de bits que está sendo enviado. Essa **inserção de bits** é semelhante à inserção de bytes, na qual um byte de escape é inserido no fluxo de caracteres enviado antes de ocorrer um byte de flag nos dados. Isso também garante uma densidade mínima de transições, o que ajuda a camada física a manter a sincronização. O USB (Universal Serial Bus) utiliza a inserção de bits por esse motivo.

Ao ver cinco bits 1 consecutivos sendo recebidos, seguidos por um bit 0, o receptor automaticamente remove o bit 0. A inserção de bits, assim como a inserção de bytes, é completamente transparente para a camada de rede de ambos os computadores. Se os dados do usuário contiverem o padrão de flag 01111110, esse flag será transmitido como 011111010, mas será armazenado na memória do receptor como 01111110. As camadas superiores não tomam conhecimento dessa inserção de bits. A Figura 3.5 mostra um exemplo de inserção de bits.

Com a inserção de bits, o limite entre dois quadros pode ser reconhecido sem qualquer tipo de ambiguidade pelo padrão de flags. Desse modo, se o receptor perder o controle de onde estão os dados, bastará varrer a entrada em busca de sequências de flags, uma vez que elas nunca ocorrem dentro dos dados, apenas nos limites dos quadros.

Com a inserção de bits e de bytes, um efeito colateral é que o comprimento de um quadro agora depende do conteúdo dos dados que ele transporta. Por exemplo, se não houver bytes de flag nos dados, 100 bytes podem ser transportados em um quadro de aproximadamente 100 bytes. Todavia, se os dados consistirem unicamente de bytes de flag, cada um terá um bit de escape associado e o quadro terá aproximadamente 200 bytes de comprimento. Com a inserção de bits, o aumento seria de aproximadamente 12,5%, pois 1 bit é inserido a cada byte.

O último método de enquadramento é usar um atalho da camada física. No Capítulo 2, vimos que a codificação de bits como sinais normalmente inclui redundância para ajudar o receptor. Essa redundância significa que alguns sinais não ocorrerão em dados regulares. Por exemplo, no código de linha 4B/5B, 4 bits de dados são mapeados para 5 bits de sinal, para garantir transições de bits suficientes. Isso significa que 16 das 32 possibilidades de sinal não são usadas. Podemos usar alguns sinais reservados para indicar o início e o final dos quadros. Com efeito, estamos usando "violações de código" (caracteres inválidos) para delimitar

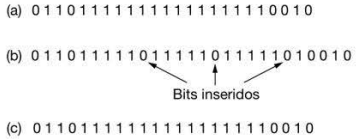

Figura 3.5 Inserção de bits. (a) Os dados originais. (b) Como os dados são exibidos na linha. (c) Como os dados são armazenados na memória do receptor após a remoção de bits.

os quadros. A beleza desse esquema é que, por serem sinais reservados, é fácil encontrar o início e o final dos quadros, e não é preciso inserir bits nos dados.

Muitos protocolos de enlace de dados, por segurança, usam uma combinação desses métodos. Um padrão comum usado para Ethernet e para 802.11 é fazer um quadro começar com um padrão bem definido, chamado **preâmbulo**. Esse padrão pode ser muito longo (72 bits é típico para redes 802.11), a fim de permitir que o receptor se prepare para um pacote que está chegando. O preâmbulo é, então, seguido por um campo de comprimento (ou seja, um contador) no cabeçalho, que é usado para localizar o final do quadro.

3.1.3 Controle de erros

Após resolvermos o problema da delimitação do início e do final de cada quadro, vamos ao problema seguinte: como ter certeza de que todos os quadros serão entregues na camada de rede de destino e na ordem apropriada? Suponha, para o momento, que o receptor consegue saber se um quadro que ele recebe contém informações corretas ou defeituosas (veremos os códigos usados para detectar e corrigir erros de transmissão na Seção 3.2). Para serviços não orientados a conexões, sem confirmação, pode ser suficiente que o emissor apenas continue enviando quadros sem se importar se eles chegaram corretamente, mas sem dúvida essa não seria uma boa opção para serviços orientados a conexões confiáveis.

A forma mais comum de garantir uma entrega confiável é dar ao transmissor algum tipo de feedback sobre o que está acontecendo no outro extremo da linha. Normalmente, o protocolo solicita que o receptor retorne quadros de controle especiais com confirmações positivas ou negativas sobre os quadros recebidos. Se receber uma confirmação positiva sobre um quadro, o transmissor saberá que ele chegou em segurança ao destino. Em contrapartida, uma confirmação negativa significa que algo saiu errado e que o quadro deve ser retransmitido.

Uma complicação adicional decorre da possibilidade de problemas de hardware fazerem um quadro desaparecer completamente (p. ex., em uma rajada de ruídos). Nesse caso, o receptor não reagirá de forma alguma, pois não há motivo para isso. De modo semelhante, se o quadro de confirmação se perder, o emissor não saberá como prosseguir. Deve ficar claro que um protocolo no qual o transmissor envia um quadro e depois espera por uma confirmação, positiva ou negativa, permanecerá suspenso para sempre caso um quadro tenha sido completamente perdido – por exemplo, em consequência de mau funcionamento do hardware ou canal de comunicação deficiente.

Essa possibilidade é tratada com a introdução de timers na camada de enlace de dados. Quando o transmissor envia um quadro, em geral ele também inicializa um timer. Este é ajustado para ser desativado após um intervalo suficientemente longo para o quadro chegar ao destino, ser processado e ter sua confirmação enviada de volta ao transmissor. Em geral, o quadro será recebido de forma correta e a confirmação voltará antes que se alcance o tempo-limite do timer e, nesse caso, ele será cancelado.

No entanto, se a confirmação ou o quadro se perder, o timer será desativado, alertando o transmissor para um problema potencial. A solução óbvia é simplesmente transmitir o quadro outra vez. Entretanto, quando os quadros podem ser transmitidos várias vezes, existe o perigo de o receptor aceitar o mesmo quadro duas ou mais vezes e de repassá-lo à camada de rede mais de uma vez. Para evitar que isso aconteça, geralmente é preciso atribuir números de sequência aos quadros transmitidos, de modo que o receptor possa distinguir as retransmissões dos originais.

A questão do gerenciamento dos timers e dos números de sequência para garantir que cada quadro seja realmente passado para a camada de rede no destino exatamente uma vez, nem mais nem menos, é uma parte importante das atribuições da camada de enlace de dados (e das camadas mais altas). Mais adiante neste capítulo, veremos uma série de exemplos cada vez mais sofisticados, para entender como é feito esse gerenciamento.

3.1.4 Controle de fluxo

Outro importante aspecto de projeto que ocorre na camada de enlace de dados (e também nas camadas mais altas) é o que fazer com um transmissor que sistematicamente deseja transmitir quadros mais rápido do que o receptor pode aceitá-los. Essa situação pode ocorrer quando o transmissor está rodando em um computador rápido e poderoso e o receptor está rodando em uma máquina lenta e inferior. Uma situação comum é quando um smartphone solicita uma página Web de um servidor muito mais poderoso, que abre a mangueira de incêndio e jorra os dados para o pobre e infeliz telefone, até que esteja completamente inundado. Mesmo que a transmissão não tenha erros, o receptor pode não conseguir lidar com os quadros com a rapidez com que chegam, e perderá alguns deles.

Sem dúvida, algo deve ser feito para impedir que essa situação ocorra. Em geral duas abordagens são usadas. Na primeira, chamada de **controle de fluxo baseado em feedback**, o receptor envia de volta ao transmissor informações que permitam a ele enviar mais dados, ou que pelo menos mostrem ao transmissor a sua situação real. Na segunda, chamada de **controle de fluxo baseado na velocidade**, o protocolo tem um mecanismo interno que limita a velocidade de envio de dados pelos transmissores, sem usar o feedback do receptor.

Neste capítulo, estudaremos os esquemas de controle de fluxo baseados em feedback, principalmente porque os esquemas baseados na velocidade são vistos apenas como parte da camada de transporte (Capítulo 5). Os esquemas baseados em feedback são vistos na camada de enlace e em camadas superiores. O último caso é mais comum hoje em dia, em que o hardware da camada de enlace é projetado

para trabalhar com rapidez suficiente para não causar perda. Às vezes, por exemplo, considera-se que as implementações de hardware da camada de enlace, como as **placas de interface de rede**, ou **NICs** (**Network Interface Cards**), atuam na "velocidade do fio", o que significa que podem tratar de quadros tão rapidamente quanto eles chegam ao enlace. Portanto, qualquer overrun não será um problema de enlace, já que é tratado por camadas mais altas.

Existem diversos esquemas de controle de fluxo baseados em feedback. No entanto, a maioria deles utiliza o mesmo princípio básico. O protocolo contém regras bem definidas sobre quando um transmissor pode enviar o quadro seguinte. Com frequência, essas regras impedem que os quadros sejam enviados até que o receptor tenha concedido permissão para a transmissão, implícita ou explicitamente. Por exemplo, quando uma conexão é estabelecida, o receptor pode informar: "Você pode me enviar *n* quadros agora, mas, depois que eles tiverem sido enviados, não envie mais nada até ser informado de que deve prosseguir". Examinaremos os detalhes em breve.

3.2 DETECÇÃO E CORREÇÃO DE ERROS

Vimos no Capítulo 2 que os canais de comunicação têm diversas características. Alguns canais, como a fibra óptica nas redes de telecomunicações, têm taxas de erro muito pequenas, de modo que os erros de transmissão são uma ocorrência rara. Mas outros canais, especialmente enlaces sem fio e antigos circuitos terminais, têm taxas de erro muito maiores. Para esses enlaces, os erros de transmissão são a norma. Eles não podem ser evitados a um custo razoável em termos de desempenho. Ou seja, os erros de transmissão estão aqui para ficar e precisamos aprender a lidar com eles.

Os projetistas de redes desenvolveram duas estratégias básicas para lidar com os erros, e ambas acrescentam informações redundantes aos dados enviados. Uma estratégia é incluir informações redundantes suficientes para permitir que o receptor deduza quais foram os dados transmitidos. A outra é incluir apenas a redundância suficiente para permitir que o receptor deduza que houve um erro (mas não qual erro) e solicite uma retransmissão. A primeira estratégia usa **códigos de correção de erros**, enquanto a segunda usa **códigos de detecção de erros**. O uso de códigos de correção de erros normalmente é conhecido como **correção antecipada de erros**, ou **FEC** (**Forward Error Correction**).

Cada uma dessas técnicas ocupa um nicho em ambientes específicos. Em canais altamente confiáveis, como os de fibra, é mais econômico utilizar um código de detecção de erros e simplesmente retransmitir o bloco defeituoso ocasional. Contudo, em canais como enlaces sem fio, que geram muitos erros, é melhor adicionar redundância suficiente a cada bloco para que o receptor seja capaz de descobrir qual era o bloco transmitido originalmente. A FEC é usada em canais com ruído porque as retransmissões podem ter erros tanto quanto a primeira transmissão.

Uma consideração importante para esses códigos é o tipo de erro que provavelmente ocorrerá. Nem os códigos de correção nem os de detecção de erros podem lidar com todos os erros possíveis, pois os bits redundantes que oferecem proteção provavelmente serão recebidos com erro, como os bits de dados (o que pode comprometer sua proteção). Seria bom se o canal tratasse os bits redundantes de forma diferente da que trata os bits de dados, mas isso não acontece. Todos eles são apenas bits para o canal. Isso significa que, para evitar erros não detectados, o código precisa ser forte o suficiente para lidar com os erros esperados.

Um modelo é aquele em que os erros são causados por valores extremos de ruído térmico, que abafa o sinal rápida e ocasionalmente, fazendo surgir erros de único bit isolados. Outro modelo consiste em erros que tendem a vir em rajadas, em vez de isolados, oriundos dos processos físicos que os geram – como uma atenuação profunda em um canal sem fio ou interferência elétrica transitória em um canal com fio.

Os dois modelos importam na prática e enfrentam diferentes dilemas. O fato de os erros acontecerem em rajadas tem vantagens e desvantagens em relação aos erros isolados de único bit. Uma vantagem é que os dados de computadores sempre são enviados em blocos de bits. Suponha que o tamanho do bloco seja 1.000 bits e que a taxa de erros seja 0,001 por bit. Se os erros fossem independentes, a maioria dos blocos teria um erro. Todavia, se os erros surgirem em rajadas de 100, apenas um bloco em 100 será afetado, em média. A desvantagem dos erros em rajada é que, quando ocorrem, são muito mais difíceis de corrigir que os erros isolados.

Também existem outros tipos de erros. Às vezes, o local de um erro será conhecido, talvez porque a camada física tenha recebido um sinal analógico que foi longe do valor esperado para 0 ou 1 e declarado que o bit foi perdido. Essa situação é chamada de **canal de cancelamento**. É mais fácil corrigir erros em canais de cancelamento do que em canais que invertem bits, pois, mesmo que o valor do bit tenha sido perdido, pelo menos sabemos qual deles tem erro. Contudo, normalmente não temos o benefício dos cancelamentos.

Em seguida, examinaremos os códigos de correção de erros e os códigos de detecção de erros. Contudo, tenha dois pontos em mente. Primeiro, cobrimos esses códigos na camada de enlace porque esse é o primeiro lugar em que nos deparamos com o problema de transmitir grupos de bits de modo confiável. Contudo, os códigos são muito usados porque a confiabilidade é uma preocupação geral. Os códigos de correção de erros também são vistos na camada

física, principalmente para canais com ruído e em camadas superiores, particularmente para mídia em tempo real e distribuição de conteúdo. Os códigos de detecção de erros normalmente são usados nas camadas de enlace, de rede e de transporte.

O segundo ponto que se deve ter em mente é que os códigos de detecção e/ou correção de erros são matemática aplicada. A menos que você seja particularmente adepto aos campos de Galois ou às propriedades das matrizes esparsas, deverá obter códigos com boas propriedades a partir de uma fonte confiável, em vez de criar os seus próprios. De fato, é isso que muitos padrões de protocolo fazem, com os mesmos códigos aparecendo repetidas vezes. A seguir, estudaremos um código simples com detalhes e depois descreveremos rapidamente os códigos avançados. Desse modo, podemos entender os dilemas a partir do código simples e falar sobre os códigos usados na prática por meio dos códigos avançados.

3.2.1 Códigos de correção de erros

Vamos examinar quatro tipos de código de correção de erros:

1. Códigos de Hamming.
2. Códigos de convolução binários.
3. Códigos de Reed-Solomon.
4. Códigos de verificação de paridade de baixa densidade.

Todos esses códigos acrescentam redundância às informações enviadas. Um quadro consiste em m bits de dados (ou seja, mensagem) e r bits redundantes (ou seja, verificação). Em um **bloco de código**, os r bits de verificação são calculados unicamente como uma função dos m bits de dados com os quais eles são associados, como se os m bits fossem pesquisados em uma grande tabela para encontrar seus r bits de verificação correspondentes. Em um **código sistemático**, os m bits de dados são enviados diretamente, com os bits de verificação, em vez de ser codificados eles mesmos antes de ser enviados. Em um **código linear**, os r bits de verificação são calculados como uma função linear dos m bits de dados. A operação OU exclusivo (XOR), ou adição de módulo 2, é uma escolha popular. Isso significa que a codificação pode ser feita com operações como multiplicações de matrizes ou circuitos lógicos simples. Os códigos que veremos nesta seção são blocos de código lineares, sistemáticos, a menos que indiquemos de outra forma.

Considere que o tamanho total de um bloco seja n (ou seja, $n = m + r$). Descreveremos isso como um código (n,m). Uma unidade de n bits contendo bits de dados e verificação é conhecida como uma **palavra de código** de n bits. A **taxa de código**, ou simplesmente taxa, é a fração da palavra de código que transporta informações não redundantes, ou m/n. As taxas usadas na prática variam bastante. Elas poderiam ser 1/2 para um canal com ruído, em que metade da informação recebida é redundante, ou perto de 1 para um canal de alta qualidade, com apenas um pequeno número de bits de verificação acrescentados a uma mensagem grande.

Para entender como os erros podem ser tratados, primeiro é necessário examinar de perto o que realmente é um erro. Dadas duas palavras de código que podem ser transmitidas ou recebidas – digamos, 10001001 e 10110001 –, é possível determinar quantos bits correspondentes diferem. Nesse caso, são 3 os bits divergentes. Para determinar quantos bits apresentam diferenças, basta efetuar uma operação XOR entre as duas palavras de código e contar o número de bits 1 no resultado. Por exemplo:

$$\begin{array}{r} 10001001 \\ \underline{10110001} \\ 00111000 \end{array}$$

O número de posições de bits em que duas palavras de código diferem entre si é chamado **distância de Hamming**, em homenagem a Richard Hamming (Hamming, 1950). Isso significa que, se duas palavras de código estiverem a uma distância de Hamming uma da outra igual a d, será necessário corrigir d erros de bits isolados para converter uma palavra na outra.

Dado o algoritmo para calcular os bits de verificação, é possível construir uma lista completa das palavras de código válidas e, a partir dela, encontrar as duas palavras de código com a menor distância de Hamming. Essa é a distância de Hamming do código completo.

Na maioria das aplicações de transmissão de dados, todas as 2^m mensagens de dados possíveis são válidas; no entanto, em virtude da forma como os bits de verificação são calculados, nem todas as 2^n palavras de código possíveis são usadas. De fato, quando existem r bits de verificação, somente a pequena fração de $2^m/2^n$ ou $1/2^r$ das mensagens possíveis será uma palavra de código válida. É a dispersão com que a mensagem é embutida no espaço das palavras de código que permite que o receptor detecte e corrija erros.

As propriedades de detecção e de correção de erros de um código dependem de sua distância de Hamming. Para detectar d erros de modo confiável, você precisa de um código de distância $d + 1$, pois com tal código não há como d erros de bits transformarem uma palavra de código válida em outra. Quando o receptor descobre uma palavra de código inválida, isso pode indicar que houve um erro de transmissão. Da mesma forma, para corrigir d erros, você precisa de um código de distância $2d + 1$ porque, dessa forma, as palavras de código válidas estarão tão distantes que, mesmo com d alterações, a palavra de código original continuará mais próxima do que qualquer outra. Isso significa que a palavra de código original pode ser determinada exclusivamente com base na suposição de que um número maior de erros é menos provável.

Como um exemplo simples de código de correção de erros, considere um código que contenha apenas quatro palavras de código válidas:

0000000000, 0000011111, 1111100000 e 1111111111

Esse código tem uma distância igual a 5, o que significa que ele pode corrigir erros duplos ou detectar erros quádruplos. Se a palavra de código 0000000111 for detectada e esperarmos apenas erros de 1 ou 2 bits, o receptor saberá que a original deve ter sido 0000011111. No entanto, se um erro triplo transformar 0000000000 em 0000000111, o erro não será corrigido da maneira adequada. De maneira alternativa, se esperarmos todos esses erros, podemos detectá-los. Como nenhuma das palavras de código recebidas é uma palavra de código válida, um erro deve ter ocorrido. Deve ficar evidente que, nesse exemplo, não podemos corrigir erros duplos e detectar erros quádruplos, pois isso exigiria que interpretássemos uma palavra de código recebida de duas maneiras.

Em nosso exemplo, a tarefa de decodificar encontrando a palavra de código válida mais próxima da palavra de código recebida pode ser feita por inspeção. Infelizmente, no caso mais geral, quando todas as palavras de código precisam ser avaliadas como candidatas, essa tarefa pode ser uma busca demorada. Em vez disso, os códigos práticos são criados de modo que admitam atalhos para encontrar o que provavelmente foi a palavra de código original.

Suponha que desejemos criar um código com m bits de mensagem e r bits de verificação que permitirão a correção de todos os erros simples. Cada uma das 2^m mensagens válidas tem n palavras de código inválidas a uma distância da mensagem igual a 1. Essas palavras inválidas são formadas pela inversão sistemática de cada um dos n bits da palavra de código de n bits formada a partir dela. Portanto, cada uma das 2^m mensagens válidas exige $n + 1$ padrões de bits dedicados a ela. Como o número total de padrões de bits é 2^n, devemos ter $(n + 1)2^m \leq 2^n$. Utilizando $n = m + r$, esse requisito passa a ser

$$(m + r + 1) \leq 2^r \qquad (3.1)$$

Se m for determinado, o limite para o número de bits de verificação necessários para corrigir erros isolados será mais baixo.

Esse limite teórico mais baixo pode, na verdade, ser alcançado pela utilização de um método criado por Hamming (1950). Nos **códigos de Hamming**, os bits da palavra de código são numerados consecutivamente, começando com o bit 1 da extremidade esquerda, o bit 2 imediatamente à sua direita, e assim por diante. Os bits que são potências de 2 (1, 2, 4, 8, 16, etc.) são de verificação. Os outros (3, 5, 6, 7, 9, etc.) são preenchidos com os m bits de dados. Esse padrão pode ser visto para um código de Hamming (11,7) com 7 bits de dados e 4 bits de verificação na Figura 3.6. Cada bit de verificação força a paridade de algum conjunto de bits, incluindo seu próprio conjunto, a ser par (ou ímpar). Um bit pode ser incluído em vários cálculos de verificação de bits. Se quiser ver para quais bits de verificação o bit de dados na posição k contribui, reescreva k como uma soma de potências de 2. Por exemplo, $11 = 1 + 2 + 8$ e $29 = 1 + 4 + 8 + 16$. Um bit é verificado apenas por aqueles bits de verificação que ocorrem em sua expansão (p. ex., o bit 11 é verificado pelos bits 1, 2 e 8). No exemplo, os bits de verificação são calculados para somas de paridade par para uma mensagem que é o código ASCII letra "A".

Essa construção resulta em um código com uma distância de Hamming igual a 3, o que significa que ela pode corrigir erros simples (ou detectar erros duplos). O motivo para a numeração muito cuidadosa dos bits de mensagem e de verificação torna-se aparente no processo de decodificação. Quando uma palavra de código chega, o receptor refaz os cálculos do bit de verificação, incluindo os valores dos bits de verificação recebidos. Chamamos estes de resultados de verificação. Se os bits de verificação forem corretos, então, para somas de paridade par, cada resultado de verificação deve ser zero. Nesse caso, a palavra de código é aceita como válida.

Entretanto, se os resultados da verificação não forem todos zero, um erro foi detectado. O conjunto de resultados de verificação forma a **síndrome de erro** que é usada para localizar e corrigir o erro. Na Figura 3.6, um erro de único bit ocorreu no canal, de modo que os resultados de verificação são 0, 1, 0 e 1 para $k = 8, 4, 2$ e 1, respectivamente. Isso gera uma síndrome de 0101 ou $4 + 1 = 5$. Pelo projeto do esquema, isso significa que o quinto bit está com erro. A inversão do bit incorreto (que pode ser um bit de verificação

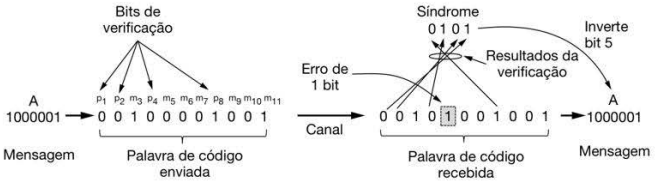

Figura 3.6 Exemplo de um código de Hamming (11,7) corrigindo um erro de único bit.

ou de dados) e o descarte dos bits de verificação geram a mensagem correta de um ASCII "A".

As distâncias de Hamming são valiosas para entender os blocos de códigos, e os códigos de Hamming são usados na memória de correção de erros. Contudo, a maioria das redes usa códigos mais fortes, como o **código de convolução**. Este é o único que veremos que não é um bloco de código. Em um código de convolução, um codificador processa uma sequência de bits de entrada e gera uma sequência de bits de saída. Não existe tamanho de mensagem ou limite de codificação natural, como em um bloco de código. A saída depende dos bits de entrada atual e anterior. Ou seja, o codificador tem memória. O número de bits anteriores do qual a saída depende é chamado **comprimento restritivo** do código. Os códigos de convolução são especificados em termos de sua taxa e comprimento restritivo.

Os códigos de convolução são muito usados em redes implantadas, por exemplo, como parte do sistema de telefonia móvel GSM, em comunicações por satélite e nas redes 802.11. Como exemplo, um código de convolução popular aparece na Figura 3.7. Esse código é conhecido como código de convolução da NASA, com $r = 1/2$ e $k = 7$, pois ele foi usado inicialmente para as missões espaciais Voyager, a partir de 1977. Desde então, ele tem sido bastante reutilizado, por exemplo, como parte do padrão 802.11.

Na Figura 3.7, cada bit de entrada no lado esquerdo produz dois bits de saída no lado direito, os quais são somas de operações XOR entre a entrada e o estado interno. Por lidar com bits e realizar operações lineares, esse é um código de convolução binário, linear. Como 1 bit de entrada produz 2 bits de saída, o código é 1/2. Ele não é sistemático, pois nenhum dos bits de saída é simplesmente o bit de entrada.

O estado interno é mantido em seis registradores de memória. Toda vez que outro bit é inserido, os valores nos registradores são deslocados para a direita. Por exemplo, se 111 for inserido e o estado inicial contiver zeros, o estado interno, escrito da esquerda para a direita, se tornará 100000, 110000 e 111000 após o primeiro, o segundo e o terceiro bits serem inseridos. Os bits de saída serão 11, seguidos por 10 e, depois, 01. São necessários sete deslocamentos para esvaziar uma entrada, de modo que ela não afete a saída. O comprimento da restrição desse código é, portanto, $k = 7$.

Um código de convolução é decodificado encontrando-se a sequência de bits de entrada que tem maior probabilidade de ter produzido a sequência observada de bits de saída (que inclui quaisquer erros). Para valores pequenos de k, isso é feito com um algoritmo bastante usado, desenvolvido por Viterbi (Forney, 1973). O algoritmo percorre a sequência observada, mantendo para cada etapa e para cada estado interno possível a sequência de entrada que teria produzido a sequência observada com o mínimo de erros. A sequência de entrada que exige o mínimo de erros no final é a mensagem mais provável.

Os códigos de convolução têm sido populares na prática porque é fácil fatorar a incerteza de um bit sendo 0 ou 1 na decodificação. Por exemplo, supondo que –1V seja o nível lógico 0 e +1V seja o nível lógico 1, poderíamos receber 0,9V e –0,1V para 2 bits. Em vez de mapear esses sinais como 1 e 0 imediatamente, gostaríamos de tratar 0,9V como "provavelmente 1" e –0,1V como "talvez 0" e corrigir a sequência como um todo. As extensões do algoritmo de Viterbi podem trabalhar com essas incertezas para oferecer uma correção de erros mais forte. Essa técnica de trabalho com a incerteza de um bit é chamada de **decodificação de decisão soft**. Por sua vez, a tarefa de decidir se cada bit é 0 ou 1 antes da correção de erros subsequente é chamada **decodificação de decisão hard**.

O terceiro tipo de código de correção de erros que descreveremos é o **código de Reed-Solomon**. Assim como os códigos de Hamming, os de Reed-Solomon são blocos de código lineares, e eles normalmente também são sistemáticos. Diferentemente dos códigos de Hamming, que operam sobre bits individuais, os códigos de Reed-Solomon operam sobre m símbolos de bit. Naturalmente, a matemática é mais complicada, de modo que descreveremos sua operação por analogia.

Os códigos de Reed-Solomon são baseados no fato de que cada polinômio de grau n é determinado unicamente por $n + 1$ pontos. Por exemplo, uma linha que tem a forma $ax + b$ é determinada por dois pontos. Os pontos extras na mesma linha são redundantes, o que é útil para a correção de erros. Imagine que temos dois pontos de dados que representam uma linha e enviamos esses dois pontos de dados mais dois pontos de verificação escolhidos para que se encontrem na mesma linha. Se um dos pontos for recebido com erro, ainda podemos recuperar os pontos de

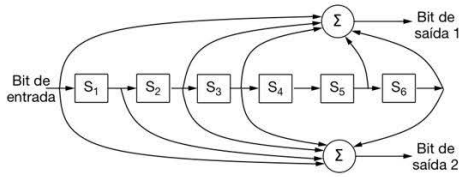

Figura 3.7 O código de convolução binário da NASA usado no padrão 802.11.

dados passando uma linha pelos pontos recebidos. Três dos pontos estarão na linha e um ponto, aquele com erro, não estará. Encontrando a linha, corrigiremos o erro.

Os códigos de Reed-Solomon, na realidade, são definidos como polinômios que operam por campos finitos, mas que funcionam de maneira semelhante. Para símbolos de m bits, as palavras de código têm $2^m - 1$ símbolos de comprimento. Uma escolha popular é tornar $m = 8$, de modo que os símbolos são bytes. Uma palavra de código tem, então, 255 bytes de comprimento. O código (255, 233) é bastante utilizado; ele acrescenta 22 símbolos redundantes a 233 símbolos de dados. A decodificação com correção de erros é feita com um algoritmo desenvolvido por Berlekamp e Massey, que pode realizar com eficiência a tarefa de ajuste para códigos de tamanho moderado (Massey, 1969).

Os códigos de Reed-Solomon são bastante utilizados na prática, em virtude de suas fortes propriedades de correção de erro, particularmente para erros em rajada. Eles são usados para DSL, dados sobre cabo, comunicações por satélite e talvez de forma mais intensa em CDs, DVDs e discos Blu-ray. Por serem baseados em símbolos de m bits, um erro de único bit e um erro em rajada de m bits são tratados simplesmente como um erro de símbolo. Quando $2t$ símbolos redundantes são somados, um código de Reed-Solomon é capaz de corrigir até t erros em qualquer um dos símbolos transmitidos. Isso significa, por exemplo, que o código (255, 233), que tem 22 símbolos redundantes, pode corrigir até 16 erros de símbolo. Como os símbolos podem ser consecutivos e ter 8 bits cada um, uma rajada de erros de até 128 bits pode ser corrigida. A situação é ainda melhor se o modelo de erro for de cancelamento (p. ex., um arranhão em um CD que destrói alguns símbolos). Nesse caso, até $2t$ erros podem ser corrigidos.

Os códigos de Reed-Solomon normalmente são usados em combinação com outros códigos, como um código de convolução. O pensamento é o seguinte: os códigos de convolução são eficazes no tratamento de erros de bit isolados, mas eles falharão, provavelmente com uma rajada de erros, se houver muitos erros no fluxo de bits recebido. Acrescentando um código Reed-Solomon dentro do código de convolução, a decodificação Reed-Solomon pode liquidar as rajadas de erros, uma tarefa na qual é muito bom. O código completo, então, oferece boa proteção contra erros simples e em rajada.

O último código de correção de erros que analisaremos é o de **verificação de paridade com baixa densidade**, ou **LDPC** (**Low-Density Parity Check**). Os códigos LDPC são blocos de código lineares que foram inventados por Robert Gallagher em sua tese de doutorado (Gallagher, 1962). Assim como na maioria das teses, eles foram prontamente esquecidos, para serem reinventados apenas em 1995, quando os avanços na computação os tornaram viáveis.

Em um código LDPC, cada bit de saída é formado a partir de apenas uma fração dos bits de entrada. Isso leva a uma representação matricial do código, que tem uma baixa densidade de 1s, daí o nome para o código. As palavras de código recebidas são decodificadas com um algoritmo de aproximação que melhora interativamente com um melhor ajuste dos dados recebidos a uma palavra de código válida. Isso corrige erros.

Códigos LDPC são úteis para grandes tamanhos de bloco e têm excelente capacidade de correção de erros que, na prática, superam muitos outros códigos (incluindo aqueles que já examinamos). Por esse motivo, eles estão sendo rapidamente incluídos em novos protocolos e fazem parte do padrão para difusão de vídeo digital, 10 Gbps Ethernet, redes de linha de energia e a versão mais recente do 802.11. Você deverá ver muito mais deles nas redes do futuro.

3.2.2 Códigos de detecção de erros

Os códigos de correção de erros são muito utilizados em enlaces sem fio, conhecidos por serem ruidosos e propensos a erros em comparação à fiação de cobre ou à fibra óptica. Sem códigos de correção de erros, seria difícil conseguir algo. Contudo, usando-se fio de cobre ou fibra de alta qualidade, a taxa de erros é muito mais baixa e, assim, a detecção de erros e a retransmissão em geral são mais eficientes para lidar com o erro ocasional.

Examinaremos três códigos de detecção de erros. Todos eles são blocos de código lineares e sistemáticos:

1. Paridade.
2. Checksums.
3. Verificações de redundância cíclica (CRCs).

Para ver como eles podem ser mais eficientes do que os códigos de correção de erros, considere o primeiro código de detecção de erros, em que um único **bit de paridade** é acrescentado aos dados. O bit de paridade é escolhido de modo que o número de bits 1 na palavra de código seja par (ou ímpar). Fazer isso é equivalente a calcular o bit de paridade (par) como a soma de módulo 2 ou a operação XOR dos bits de dados. Por exemplo, quando 1011010 é enviado na paridade par, um bit é acrescentado ao final, para torná-lo 10110100. Com a paridade ímpar, 1011010 torna-se 10110101. Um código com um único bit de paridade tem uma distância de 2, pois qualquer erro de único bit produz uma palavra de código com a paridade errada. Isso significa que ele pode detectar erros de um único bit.

Considere um canal no qual os erros são isolados e a taxa de erros é de 10^{-6} por bit. Essa pode parecer uma taxa de erros pequena, mas é no mínimo uma taxa justa para um cabo longo que esteja desafiando a detecção de erros. Os enlaces de LAN comuns oferecem taxas de erro de bit de 10^{-10}. Defina o tamanho do bloco como 1.000 bits. Para proporcionar a correção de erros de blocos de 1.000 bits, sabemos, pela Equação 3.1, que são necessários 10 bits de

verificação. Assim, um megabit de dados necessitaria de 10.000 bits de verificação. Para simplesmente detectar um bloco com um único erro de 1 bit, um bit de paridade por bloco seria suficiente. A cada 1.000 blocos, será descoberto que um bloco possui erro e um bloco extra (1.001 bits) terá de ser transmitido para repará-lo. O overhead total para o método de detecção de erros e retransmissão é de apenas 2.001 bits por megabit de dados, contra 10.000 bits para um código de Hamming.

Uma dificuldade com esse esquema é que um único bit de paridade só pode detectar, de maneira confiável, um erro de único bit no bloco. Se o bloco tiver um erro em rajada longo, a probabilidade de ele ser detectado é de apenas 0,5, o que não é muito aceitável. As disparidades poderão ser consideravelmente melhoradas se cada bloco for enviado como uma matriz retangular com n bits de largura e k bits de altura. Agora, se calcularmos e enviarmos um bit de paridade para cada linha, até k erros de bit serão confiantemente detectados, desde que haja no máximo um erro por linha.

Contudo, há outra coisa que podemos fazer que oferece melhor proteção contra erros em rajada: calcular os bits de paridade sobre os dados em uma ordem diferente daquela em que os bits de dados são transmitidos pelo canal de comunicação. Isso é chamado de **entrelaçamento**. Nesse caso, calcularemos um bit de paridade para cada uma das n colunas e enviaremos todos os bits de dados como k linhas, enviando as linhas de cima para baixo e os bits em cada linha da esquerda para a direita, da maneira normal. Na última linha, enviamos os n bits de paridade. Essa ordem de transmissão pode ser vista na Figura 3.8 para $n = 7$ e $k = 7$.

O entrelaçamento é uma técnica geral para converter um código que detecta (ou corrige) erros isolados em um código que detecta (ou corrige) erros em rajada. Na Figura 3.8, quando ocorre um erro em rajada de tamanho $n = 7$, os bits que contêm erro estão espalhados por diferentes colunas. (Um erro em rajada não implica que todos os bits estejam errados, mas sim que pelo menos o primeiro e o último estejam. Na Figura 3.8, 4 bits foram invertidos por uma faixa de 7 bits.) No máximo 1 bit em cada uma das n colunas será afetado, de modo que os bits de paridade nessas colunas detectarão o erro. Esse método usa n bits de paridade sobre blocos de kn bits de dados para detectar um único erro em rajada de comprimento n ou menor.

Contudo, uma rajada de comprimento $n + 1$ passará sem ser detectada se o primeiro e o último bits forem invertidos e todos os outros bits estiverem corretos. Se o bloco estiver bastante alterado por uma extensa rajada ou por várias rajadas mais curtas, a probabilidade de que qualquer uma das n colunas tenha a paridade correta por acidente é 0,5, de modo que a probabilidade de um bloco com problema ser aceito quando não deveria é de 2^{-n}.

O segundo tipo de código de correção de erros, o **checksum**, está bastante relacionado aos grupos de bits de paridade. O termo "checksum" normalmente é usado para indicar um grupo de bits de verificação associados a uma mensagem, independentemente de como são calculados. Um grupo de bits de paridade é um exemplo de checksum. Contudo, existem outros checksums, mais robustos, baseados na soma acumulada dos bits de dados da mensagem. O checksum normalmente é colocado no final da mensagem, como complemento da função soma e não de soma. Desse modo, os erros podem ser detectados somando a palavra inteira de código recebida, tanto bits de dados quanto o checksum. Se o resultado for zero, nenhum erro foi detectado.

Um exemplo de checksum é a soma de verificação de 16 bits da Internet usada em todos os pacotes da rede como parte do protocolo IP (Braden et al., 1988). Esse checksum é uma soma dos bits da mensagem dividida por palavras de 16 bits. Como esse método opera sobre palavras, em vez de bits, assim como na paridade, os erros que deixam a paridade inalterada ainda podem alterar a soma e ser detectados. Por exemplo, se o bit de mais baixa ordem em duas palavras diferentes for invertido de 0 para 1, uma verificação de paridade entre esses bits deixaria de detectar um erro. Contudo, dois 1s serão acrescentados ao checksum de 16 bits para produzir um resultado diferente. O erro pode, então, ser detectado.

O checksum da Internet é calculado com a aritmética de complemento de um, em vez da soma de módulo 2^{16}.

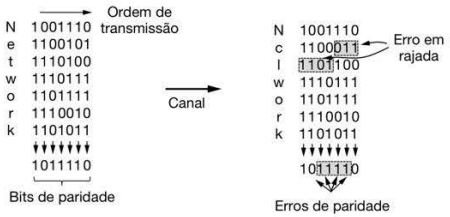

Figura 3.8 Entrelaçamento de bits de paridade para detectar um erro em rajada.

Na aritmética de complemento de um, um número negativo é o complemento bit a bit de seu correspondente positivo. Os computadores modernos trabalham na aritmética de complemento de dois, em que um número negativo é o complemento de um mais um. Em um computador com complemento de dois, a soma no complemento de um é equivalente a apanhar o total em módulo 2^{16} e somar qualquer overflow dos bits de alta ordem aos de baixa ordem. Esse algoritmo oferece uma cobertura mais uniforme dos dados pelos bits do checksum. De outra forma, dois bits de alta ordem podem ser somados, gerar overflow e serem perdidos sem alterar o total. Também existe outro benefício. O complemento de um tem duas representações de zero, todos os bits 0 e todos os bits 1. Isso permite um valor (p. ex., todos os bits 0) para indicar que não há um checksum, sem a necessidade de outro campo.

Ao longo de décadas, sempre foi considerado que os quadros a serem somados para verificação contêm bits aleatórios. Todas as análises dos algoritmos de checksum têm sido feitas sob essa hipótese. A inspeção de dados reais por Partridge et al. (1995) mostrou que essa suposição é bastante errada. Como consequência, os erros não detectados são, em alguns casos, muito mais comuns do que se havia imaginado.

O checksum da Internet em particular é eficiente e simples, mas oferece uma proteção fraca em alguns casos, exatamente porque essa é uma soma simples. Ele não detecta a exclusão ou o acréscimo de dados zero, nem a troca de partes da mensagem, e oferece pouca proteção contra pedaços da mensagem em que partes de dois pacotes são reunidas. A ocorrência desses erros pode parecer improvável por processos aleatórios, mas eles são simplesmente o tipo de erro que pode ocorrer com um hardware defeituoso.

Uma escolha melhor é o **checksum de Fletcher** (Fletcher, 1982). Ele inclui um componente posicional, somando o produto dos dados e sua posição à soma acumulada. Isso oferece melhor detecção das mudanças na posição dos dados.

Embora os dois esquemas anteriores às vezes possam ser adequados em camadas superiores, na prática, um terceiro tipo mais forte de código de detecção de erros tem o uso generalizado na camada de enlace: é o **código de redundância cíclica**, ou **CRC** (**Cyclic Redundancy Check**), também conhecido como **código polinomial**. Os códigos polinomiais são baseados no tratamento de sequências de bits como representações de polinômios com coeficientes de 0 e 1 apenas. Um quadro de k bits é considerado como a lista de coeficientes para um polinômio com k termos, variando de x^{k-1} a x^0. Dizemos que tal polinômio é de grau $k-1$. O bit de alta ordem (mais à esquerda) é o coeficiente de x^{k-1}, o próximo bit é o coeficiente de x^{k-2}, e assim por diante. Por exemplo, 110001 tem 6 bits e, portanto, representa um polinômio de seis termos com coeficientes 1, 1, 0, 0, 0 e 1: $1x^5 + 1x^4 + 0x^3 + 0x^2 + 0x^1 + 1x^0$.

A aritmética de polinômios é feita em módulo 2, de acordo com as regras da teoria algébrica. Ela não tem "vai uns" para a adição ou empréstimos para a subtração. Tanto a adição quanto a subtração são idênticos ao XOR. Por exemplo:

10011011	00110011	11110000	01010101
+ 11001010	+ 11001101	− 10100110	− 10101111
01010001	11111110	01010110	11111010

A divisão longa é efetuada do mesmo modo que em binário, exceto pelo fato de a subtração novamente ser de módulo 2. Diz-se que um divisor "cabe em" um dividendo se o dividendo tem a mesma quantidade de bits do divisor.

Quando o método do código polinomial é empregado, o transmissor e o receptor devem concordar em relação a um **polinômio gerador**, $G(x)$, antecipadamente. Tanto o bit de mais alta ordem quanto o de mais baixa ordem do polinômio gerador devem ser iguais a 1. Para calcular o CRC de um quadro com m bits, que corresponde ao polinômio $M(x)$, o quadro deve ter mais bits do que o polinômio gerador. A ideia é acrescentar um CRC ao final do quadro, de forma que o polinômio representado pelo quadro verificado pela soma seja divisível por $G(x)$. Quando obtiver o quadro verificado, o receptor tentará dividi-lo por $G(x)$. A existência de resto indica que houve um erro de transmissão.

O algoritmo para calcular o CRC é o seguinte:

1. Seja r o grau de $G(x)$. Acrescente r bits zero à extremidade de baixa ordem do quadro, de modo que ele passe a conter $m + r$ bits e corresponda ao polinômio $x^r M(x)$.
2. Divida a sequência de bits correspondente a $G(x)$ pela sequência de bits correspondente a $x^r M(x)$ utilizando a divisão de módulo 2.
3. Subtraia o resto (que tem sempre r ou menos bits) da sequência de bits correspondente a $x^r M(x)$ utilizando a subtração de módulo 2. O resultado é o quadro, com o checksum, que deverá ser transmitido. Chame o polinômio de $T(x)$.

A Figura 3.9 ilustra o cálculo referente a um quadro 1101011111, usando o gerador $G(x) = x^4 + x + 1$.

Deve ficar claro que $T(x)$ é divisível (em módulo 2) por $G(x)$. Em qualquer problema de divisão, se você subtrair o resto do dividendo, o resultado será divisível pelo divisor. Por exemplo, na base 10, se você dividir 210.278 por 10.941, o resto será 2.399. Subtraindo-se 2.399 de 210.278, o resultado final (207.879) será divisível por 10.941.

Agora vamos analisar o poder desse método. Que tipos de erro serão detectados? Imagine que ocorra um erro de transmissão de forma que, em lugar de chegar a sequência de bits correspondente a $T(x)$, seja recebida a soma $T(x) + E(x)$. Cada bit 1 em $E(x)$ corresponde a um bit que foi invertido. Se houver k bits 1 em $E(x)$, isso significa que ocorreram k erros de único bit. Um único erro em rajada é

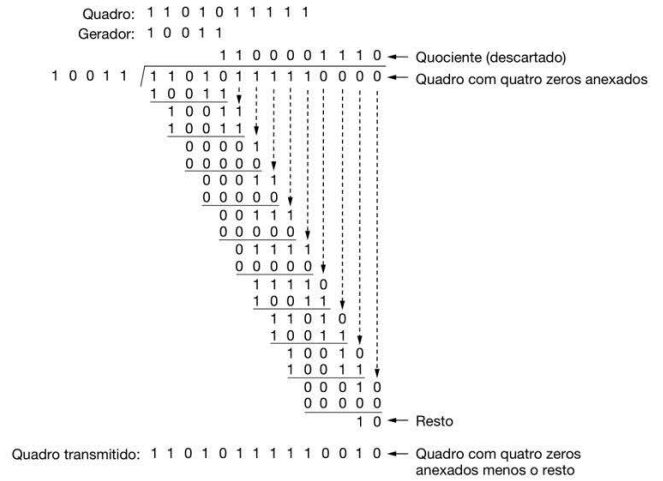

Figura 3.9 Exemplo de cálculo do CRC.

caracterizado por um bit 1 inicial, uma mistura de bits 0 e 1 e um bit 1 final, sendo todos os outros bits iguais a 0.

Ao receber o quadro com o checksum, o receptor o divide por $G(x)$; ou seja, ele calcula $[T(x) + E(x)]/G(x)$. $T(x)/G(x)$ é igual a 0; portanto, o resultado do cálculo é simplesmente $E(x)/G(x)$. Os erros que corresponderem a polinômios contendo $G(x)$ como fator serão simplesmente ignorados; todos os outros serão descobertos.

Se houver ocorrido um erro de único bit, $E(x) = x^i$, onde i determina o bit incorreto. Se contiver dois ou mais termos, $G(x)$ nunca dividirá $E(x)$; portanto, todos os erros de único bit serão detectados.

Se tiverem ocorrido dois erros isolados de único bit, $E(x) = x^i + x^j$, onde $i > j$. Como alternativa, esse cálculo pode ser representado como $E(x) = x^j(x^{i-j} + 1)$. Se considerarmos que $G(x)$ não é divisível por x, uma condição suficiente para todos os erros duplos serem detectados é que $G(x)$ não divida $x^k + 1$ para qualquer k até o valor máximo de $i - j$ (i.e., até o comprimento máximo do quadro). São conhecidos polinômios simples, de grau baixo, que protegem quadros longos. Por exemplo, $x^{15} + x^{14} + 1$ não dividirá $x^k + 1$ para qualquer valor de k abaixo de 32.768.

Se houver um número ímpar de bits com erros, $E(x)$ conterá um número ímpar de termos (p. ex., $x^5 + x^2 + 1$, mas não $x^2 + 1$). É interessante observar que nenhum polinômio com um número ímpar de termos tem $x + 1$ como fator no sistema de módulo 2. Ao tornar $x + 1$ um fator de $G(x)$, podemos detectar todos os erros que consistem em um número ímpar de bits invertidos. Estatisticamente, apenas isso já compreende metade dos casos.

Por último, e muito importante, um código polinomial com r bits de verificação detectará todos os erros em rajada que tiverem um tamanho $\leq r$. Um erro em rajada de tamanho k pode ser representado por $x^i(x^{k-1} + ... + 1)$, onde i determina a distância entre a rajada e a extremidade direita do quadro recebido. Se contiver um termo x^0, $G(x)$ não terá x^i como fator; portanto, se o grau da expressão entre parênteses for menor que o grau de $G(x)$, o resto nunca poderá ser igual a zero.

Se o tamanho da rajada for $r + 1$, o resto da divisão por $G(x)$ será zero se, e somente se, a rajada for idêntica a $G(x)$. Por definição de rajada, o primeiro e o último bits devem ser iguais a 1; assim, a correspondência entre os valores dependerá dos $r - 1$ bits intermediários. Se todas as combinações forem consideradas igualmente prováveis, a probabilidade de esse quadro incorreto ser aceito como válido será de $\frac{1}{2}^{r-1}$.

Também podemos mostrar que, ao ocorrer um erro em rajada com mais de $r + 1$ bits ou forem registradas várias rajadas mais curtas, a probabilidade de um quadro defeituoso passar despercebido poderá ser igual a $\frac{1}{2}^r$, supondo-se que todos os padrões de bits sejam igualmente prováveis.

Certos polinômios se tornaram padrões internacionais. O que é utilizado no IEEE 802 acompanhou o exemplo da Ethernet e é:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

Entre outras características interessantes, ele tem a propriedade de detectar todas as rajadas de comprimento 32 ou menores e todas as rajadas que afetam um número ímpar de bits. Tem sido muito usado desde a década de 1980, mas isso não significa que seja a melhor escolha. Usando uma busca computacional completa, Castagnoli et al. (1993) e Koopman (2002) encontraram os melhores CRCs. Esses CRCs têm uma distância de Hamming de 6 para os tamanhos de mensagem típicos, enquanto o padrão do IEEE CRC-32 tem uma distância de Hamming de apenas 4.

Apesar de o cálculo necessário para computar o checksum parecer complicado, Peterson e Brown (1961) mostraram que é possível criar um simples circuito shift register (registrador de deslocamento) para calcular e conferir os CRCs no hardware. Implementações mais recentes e mais rápidas são criadas regularmente (Mitra e Nyack, 2017). Na prática, esse hardware quase sempre é utilizado. Dezenas de padrões de rede compreendem diversos CRCs, incluindo praticamente todas as LANs (p. ex., Ethernet, 802.11) e enlaces ponto a ponto (p. ex., pacotes sobre SONET).

3.3 PROTOCOLOS BÁSICOS DE ENLACE DE DADOS

Como uma introdução ao estudo dos protocolos, vamos começar examinando três protocolos com graus de complexidade crescentes. Antes disso, é útil esclarecer algumas das suposições nas quais se baseia o modelo de comunicação.

3.3.1 Premissas básicas para simplificação

Processos independentes. Para começar, supomos que, na camada física, na camada de enlace de dados e na camada de rede existem processos independentes que se comunicam pelo envio de mensagens de um lado para outro. Uma implementação comum aparece na Figura 3.10. O processo da camada física e parte do processo da camada de enlace de dados funcionam em hardware dedicado, chamado **placa de interface de rede**, ou **NIC** (**Network Interface Card**). O restante do processo da camada de enlace e o processo da camada de rede atuam sobre a CPU principal como parte do sistema operacional, com o software para o processo da camada de enlace normalmente tomando a forma de um **driver de dispositivo**. No entanto, outras implementações também são possíveis (p. ex., três processos transferidos para um hardware dedicado, chamado **acelerador de rede**, ou três processos rodando na CPU principal a uma razão definida pelo software). Na realidade, a implementação preferida muda de uma década para a outra, com as mudanças de tecnologia. De qualquer forma, tratar as

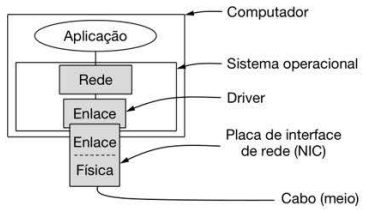

Figura 3.10 Implementação das camadas física, de enlace de dados e de rede.

três camadas como processos separados torna a discussão conceitualmente mais clara e também enfatiza a independência delas.

Comunicação unidirecional. Outra suposição de extrema importância é de que a máquina *A* deseja enviar um longo fluxo de dados à máquina *B* utilizando um serviço confiável e orientado a conexões. Mais adiante, consideraremos a situação em que *B* também deseja enviar dados a *A* simultaneamente. Supõe-se que *A* tenha um suprimento infinito de dados prontos para ser enviados, e nunca terá de esperar até que eles sejam produzidos. Quando a camada de dados de *A* solicitar dados, a camada de rede sempre será capaz de obedecer de imediato. (Mais adiante essa restrição também será superada.)

Máquinas e processos confiáveis. Também supomos que as máquinas não sofrerão panes. Isto é, esses protocolos lidam com erros de comunicação, mas não com os problemas causados por computadores que sofrem panes e são reiniciados.

No que se refere à camada de enlace de dados, o pacote repassado a ela pela camada de rede através da interface consiste em dados puros, em que cada bit deve ser entregue à camada de rede de destino. O fato de a camada de rede de destino interpretar parte do pacote como um cabeçalho não tem nenhum interesse para a camada de enlace de dados.

3.3.2 Noções básicas de transmissão e recepção

Quando a camada de enlace de dados aceita um pacote da camada de rede do transmissor, ela o encapsula em um quadro, acrescentando-lhe um cabeçalho e um final de enlace de dados (veja a Figura 3.1). Portanto, um quadro consiste em um pacote incorporado, algumas informações de controle (no cabeçalho) e um checksum (no final). Em seguida, o quadro é transmitido à camada de enlace de dados da outra máquina. Presumiremos que existem funções de biblioteca adequadas, *to_physical_layer* para enviar um quadro e *from_physical_layer* para receber um quadro. Essas

funções calculam ou acrescentam o checksum (o que normalmente é feito no hardware), de forma que os protocolos que desenvolvemos nesta seção não precisam se preocupar com isso. Por exemplo, eles poderiam usar o algoritmo de CRC discutido na seção anterior.

Inicialmente, o receptor nada tem a fazer. Ele apenas fica à espera de que algo aconteça. Nos exemplos de protocolos apresentados neste capítulo, indicaremos que a camada de enlace de dados está esperando que algo aconteça por meio da chamada função *wait_for_event*(&*event*). Essa função só retorna quando acontece algo (p. ex., quando chega um quadro). Ao retornar, a variável *event* informa o que aconteceu. O conjunto de eventos possíveis é diferente para os diversos protocolos a serem descritos e será definido separadamente para cada protocolo. Observe que, em uma situação mais realista, a camada de enlace de dados não ficará em um loop estrito à espera de um evento, como sugerimos, mas receberá uma interrupção, o que a fará interromper o que quer que esteja fazendo para manipular o quadro recebido. Apesar disso, por simplicidade, ignoraremos todos os detalhes de atividades paralelas na camada de enlace de dados, e presumiremos que ela se dedica em tempo integral apenas ao tratamento do nosso canal.

Quando um quadro chega ao receptor, este recalcula o checksum. Se este estiver incorreto (ou seja, se houve um erro de transmissão), a camada de enlace de dados será informada (*event* = *cksum_err*). Se o quadro recebido tiver chegado intacto, a camada de enlace de dados também será informada (*event* = *frame_arrival*), para que ela possa receber o quadro para inspeção usando *from_physical_layer*. Assim que recebe um quadro sem danos, a camada de enlace de dados verifica as informações de controle contidas no cabeçalho e, se tudo estiver correto, repassa a porção do pacote à camada de rede. Em nenhuma circunstância o cabeçalho do quadro será entregue à camada de rede.

Há uma boa razão para que a camada de rede nunca receba nenhuma parte do cabeçalho do quadro: manter os protocolos de rede e de enlace de dados completamente separados. Desde que a camada de rede não saiba absolutamente nada sobre o protocolo de enlace de dados ou sobre o formato do quadro, esses itens poderão ser alterados sem exigir mudanças no software da camada de rede. Isso acontece sempre que uma nova NIC é instalada em um computador. A utilização de uma interface rígida entre a camada de rede e a de enlace de dados simplifica bastante o projeto do software, pois os protocolos de comunicação das diferentes camadas podem evoluir de forma independente.

A Figura 3.11 mostra algumas declarações (na linguagem C) comuns a muitos dos protocolos que serão discutidos mais adiante. Cinco estruturas de dados são definidas no código: *boolean*, *seq_nr*, *packet*, *frame_kind* e *frame*. Um *boolean* é do tipo enumerado e pode assumir os valores verdadeiro (*true*) e falso (*false*). Um *seq_nr* é um inteiro pequeno usado para numerar os quadros, para facilitar sua distinção. Esses números de sequência variam de 0 até *MAX_SEQ* (inclusive), que é definido em cada protocolo que tenha necessidade. Um *packet* é a unidade de informação trocada entre as camadas de rede e de enlace de dados da mesma máquina, ou entre pares da camada de rede. Em nosso modelo, ele sempre contém *MAX_PKT* bytes; no entanto, de modo mais realista, ele teria comprimento variável.

A estrutura *frame* é composta por quatro campos: *kind*, *seq*, *ack* e *info*; os três primeiros contêm informações de controle, e o último pode conter os dados reais a serem transferidos. Esses campos de controle são chamados coletivamente de **cabeçalho do quadro**.

O campo *kind* indica se há dados no quadro, pois alguns protocolos distinguem quadros que contêm exclusivamente informações de controle daqueles que armazenam dados além dessas informações. Os campos *seq* e *ack* são usados para números de sequência e confirmações, respectivamente (seu uso será descrito detalhadamente mais adiante). O campo *info* de um quadro de dados contém um único pacote; o campo *info* de um quadro de controle não é usado. Uma implementação mais realista utilizaria um campo *info* de comprimento variável; nos quadros de controle, esse campo seria completamente omitido.

Novamente, é importante compreender o relacionamento entre um pacote e um quadro (veja a Figura 3.1). A camada de rede monta um pacote tomando uma mensagem da camada de transporte e acrescentando a ela o cabeçalho da camada de rede. Esse pacote é repassado à camada de enlace de dados para inclusão no campo *info* de um quadro que esteja sendo enviado. Quando o quadro chega ao destino, a camada de enlace de dados extrai o pacote do quadro e o envia à camada de rede. Dessa forma, a camada de rede pode atuar como se as máquinas pudessem trocar pacotes diretamente.

Na Figura 3.11 também estão listadas diversas funções, que são rotinas de biblioteca cujos detalhes são dependentes da implementação e cujo funcionamento interno não será discutido aqui. A função *wait_for_event* permanece à espera de que algo aconteça, como mencionamos anteriormente. As funções *to_network_layer* e *from_network _layer* são usadas pela camada de enlace de dados para enviar pacotes à camada de rede e aceitar pacotes dela, respectivamente. Observe que *from_physical_layer* e *to_physical_layer* repassam quadros entre a camada de enlace de dados e a camada física. Em outras palavras, *to_network_layer* e *from_network_layer* lidam com a interface entre as camadas 2 e 3, enquanto *from_physical_layer* e *to_physical_layer* lidam com a interface entre as camadas 1 e 2.

Na maioria dos protocolos, supomos o uso de um canal não confiável que perde quadros inteiros ocasionalmente. Para se recuperar dessas calamidades, sempre que envia um quadro, a camada de enlace de dados transmissora tem de inicializar um timer interno. Se nenhuma confirmação tiver sido recebida dentro de um intervalo predeterminado, o timer expirará por timeout e a camada de enlace de dados receberá um sinal de interrupção.

```
#define MAX_PKT 1024                                  /* determina tamanho do pacote em bytes */
typedef enum {false, true} boolean;                   /* tipo boolean */
typedef unsigned int seq_nr;                          /* números de sequência ou ack */
typedef struct {unsigned char data[MAX_PKT];}packet;  /* definição do pacote */
typedef enum {data, ack, nak} frame_kind;             /* definição de frame_kind */

typedef struct {                                      /* quadros são transportados nesta camada */
   frame_kind kind;                                   /* que tipo de quadro é este? */
   seq_nr seq;                                        /* número de sequência */
   seq_nr ack;                                        /* número de confirmação */
   packet info;                                       /* o pacote da camada de rede */
} frame;
/* Espera que um evento aconteça; retorna o tipo de evento em event. */
void wait_for_event(event_type *event);

/* Busca um pacote da camada de rede para transmissão pelo canal. */
void from_network_layer(packet *p);

/* Entrega informação de um quadro que chega à camada de rede. */
void to_network_layer(packet *p);

/* Recebe um quadro de entrada da camada física e o copia para r. */
void from_physical_layer(frame *r);

/* Passa o quadro à camada física para transmissão. */
void to_physical_layer(frame *s);

/* Inicia o timer e habilita o evento timeout. */
void start_timer(seq_nr k);

/* Termina o timer e desativa o evento timeout. */
void stop_timer(seq_nr k);

/* Inicia um timer auxiliar e habilita o ack_timeout event. */
void start_ack_timer(void);

/* Encerra o timer auxiliar e desabilita o evento ack_timeout. */
void stop_ack_timer(void);

/* Permite que a camada de rede gere um evento network_layer_ready. */
void enable_network_layer(void);

/* Proíbe a camada de rede de um evento network_layer_ready. */
void disable_network_layer(void);

/* Macro inc é expandido em linha: incrementa k de modo circular. */
#define inc(k) if (k < MAX_SEQ)k = k + 1; else k = 0
```

Figura 3.11 Algumas definições utilizadas nos protocolos apresentados a seguir. Essas definições estão armazenadas no arquivo *protocol.h*.

Em nossos protocolos, isso é tratado permitindo-se à função *wait_for_event* retornar *event = timeout*. As funções *start_timer* e *stop_timer* ativam e desativam o timer, respectivamente. Os eventos de timeout só são possíveis quando o timer está funcionando e antes que *stop_timer* seja chamado. É explicitamente permitido chamar *start_timer* enquanto o timer está funcionando; esse tipo de chamada simplesmente reinicializa o timer para provocar o próximo timeout, depois de decorrer um intervalo do timer (a menos que ele seja reiniciado ou desativado).

As funções *start_ack_timer* e *stop_ack_timer* controlam um timer auxiliar cuja finalidade é gerar confirmações sob determinadas condições.

As funções *enable_network_layer* e *disable_network_layer* são usadas nos protocolos mais sofisticados, para os quais não mais supomos que a camada de rede sempre terá pacotes a serem enviados. Quando a camada de enlace de dados habilita a camada de rede, esta passa a ter permissão para causar uma interrupção sempre que tiver um pacote para enviar. Isso é indicado por *event = network_layer_ready*. Quando a camada de rede está inativa, ela não pode causar tais eventos. Definindo com cuidado os momentos em que ativa e desativa a camada de rede, a camada de enlace de dados pode impedir que a camada de rede fique sobrecarregada com pacotes para os quais não dispõe de espaço no buffer.

Os números de sequência dos quadros estão sempre na faixa de 0 a *MAX_SEQ* (inclusive), onde *MAX_SEQ* tem um valor diferente para os diversos protocolos. Com frequência, é necessário aumentar um número de sequência em uma unidade, de forma circular (i.e., *MAX_SEQ* é seguido por 0). A macro *inc* cuida desse incremento. Ela é definida como uma macro porque é usada em linha no caminho crítico. Como veremos adiante, com frequência o processamento de protocolos é o fator que limita o desempenho da rede; portanto, a definição de operações simples como macros (em vez de funções) não afeta a legibilidade do código, mas melhora o desempenho.

As declarações da Figura 3.11 fazem parte de cada um dos protocolos brevemente apresentados a seguir. Para economizar espaço e facilitar a consulta, essas declarações foram extraídas dos protocolos e são apresentadas juntas, mas conceitualmente elas devem estar integradas aos protocolos. Na linguagem C, essa integração é feita inserindo-se as definições em um arquivo de cabeçalho especial, neste caso *protocol.h*, e utilizando-se o recurso #include do pré-processador C, que inclui essas definições nos arquivos de protocolo.

3.3.3 Protocolo simplex da camada de enlace de dados

Nesta seção, examinaremos três protocolos simples, cada um capaz de lidar com uma situação mais realista que a anterior.

Utopia: sem controle de fluxo ou correção de erros

Como primeiro exemplo, consideraremos o protocolo mais simples possível, pois não se preocupa com a possibilidade de algo sair errado. Os dados são transmitidos em apenas um sentido. As camadas de rede do transmissor e do receptor estão sempre prontas. O tempo de processamento pode ser ignorado. O espaço disponível em buffer é infinito. E o melhor de tudo é que o canal de comunicação entre as camadas de enlace de dados nunca é danificado nem perde quadros. Esse protocolo absolutamente imaginário, que denominaremos "utopia", é simplesmente para mostrar a estrutura básica com a qual trabalharemos. Sua implementação aparece na Figura 3.12.

O protocolo consiste em dois procedimentos distintos, um que envia informações e outro que as recebe. O procedimento no transmissor é executado na camada de enlace de dados da máquina de origem, e no receptor é executado na camada de enlace de dados da máquina de destino. Não são usados números de sequência ou de confirmação; portanto, *MAX_SEQ* não é necessário. O único tipo de evento possível é *frame_arrival* (ou seja, a chegada de um quadro não danificado).

No transmissor há um loop while infinito que envia os dados o mais rápido possível. O corpo do loop é formado por três ações: buscar um pacote da (sempre prestativa) camada de rede, criar um quadro de saída utilizando a variável *s* e transmitir o quadro ao destino. Apenas o campo *info* do quadro é usado por esse protocolo, pois os outros campos se referem ao controle de fluxo e de erros e, nesse caso, não há erros nem restrições de controle de fluxo.

O receptor é igualmente simples. No início, ele espera que algo aconteça, e a única possibilidade é a chegada de um quadro não danificado. Finalmente, o quadro chega e a função *wait_for_event* retorna, com *event* definido como *frame_arrival* (o que, de qualquer forma, é ignorado). A chamada *from_physical_layer* remove o quadro recém-chegado do buffer do hardware e o coloca na variável *r*, onde o código receptor poderá buscá-lo quando necessário. Por fim, a parte referente aos dados é repassada à camada de rede, e a camada de enlace de dados volta a esperar pelo próximo quadro, ficando efetivamente em suspenso até a sua chegada.

O protocolo utópico (sem restrições) é imaginário porque não trata nem do controle de fluxo nem da correção de erros. Seu processamento é próximo ao de um serviço não confirmado não orientado a conexões, que conta com as camadas mais altas para resolver esses problemas, embora até mesmo um serviço desse tipo realize alguma detecção de erros.

Acrescentando controle de fluxo: stop-and-wait

Agora, trataremos do problema de impedir que o transmissor sobrecarregue o receptor com quadros mais rapidamente do que ele consegue processá-los. Essa situação pode facilmente acontecer na prática, de modo que é muito importante poder impedi-la. No entanto, continuamos supondo que o canal de comunicação não apresenta erros e que o tráfego de dados ainda é do tipo simplex.

Uma solução é montar o receptor para que seja poderoso o bastante para processar um fluxo contínuo de quadros de ponta a ponta (ou, de modo equivalente, definir a camada de enlace para que seja lenta o bastante para que o receptor possa acompanhar). Deve ter buffer e capacidade de processamento suficientes para atuar na velocidade da linha e deve ser capaz de passar os quadros recebidos à camada de rede com rapidez suficiente. Contudo, essa é uma solução no pior dos casos. Ela exige hardware dedicado e pode desperdiçar recursos se a utilização do enlace for quase sempre baixa. Além do mais, apenas passa o problema de lidar com um emissor muito rápido para outro lugar; nesse caso, para a camada de rede.

Uma solução mais geral para esse problema é fazer o receptor oferecer feedback ao transmissor. Depois de enviar um pacote à sua camada de rede, o receptor envia um pequeno quadro fictício de volta ao transmissor, permitindo a transmissão do próximo quadro. Após o envio de um quadro, o protocolo exige que o transmissor espere sua vez,

```
/* O protocolo 1 (utopia) oferece transmissão de dados em um único sentido, do transmissor
   para o receptor. Pressupõe-se que o canal de comunicação é livre de erros e que
   o receptor é capaz de processar toda a entrada de uma forma infinitamente rápida.
   Consequentemente, o transmissor permanece em um loop enviando os dados com a maior
   rapidez possível. */
typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender1(void)
{
   frame s;                          /* buffer para um quadro de saída */
   packet buffer;                    /* buffer para um pacote de saída */
   while (true) {
       from_network_layer(&buffer);  /* pega algo para enviar */
       s.info = buffer;              /* copia para s, para transmissão */
       to_physical_layer(&s);        /* envia-o pelo caminho */
   }                                 /* O amanhã, o amanhã, o amanhã,
                                        avança em pequenos passos, dia após dia,
                                        até a última sílaba da recordação.
                                          – Macbeth, V, v */
}
void receiver1(void)
{
   frame r;
   event_type event;                 /* preenchido ao se esperar ou aguardar, mas não usado aqui */
   while (true) {
       wait_for_event(&event);       /* única possibilidade é frame_arrival */
       from_physical_layer(&r);      /* recebe o quadro que chega */
       to_network_layer(&r.info);    /* passa os dados à camada de rede */
   }
}
```

Figura 3.12 Um protocolo simplex utópico.

até a chegada de um pequeno quadro fictício (i.e., de confirmação). Esse atraso é um exemplo simples de protocolo de controle de fluxo.

Os protocolos nos quais o transmissor envia um quadro e em seguida espera por uma confirmação antes de continuar sua operação são chamados **stop-and-wait** (**pare e espere**). A Figura 3.13 mostra um exemplo de protocolo simplex stop-and-wait.

Apesar de o tráfego de dados nesse exemplo ser simplex, indo apenas do transmissor ao receptor, os quadros são enviados em ambas as direções. Consequentemente, o canal de comunicação entre as duas camadas de enlace de dados deve ser capaz de realizar a transferência bidirecional de informações. No entanto, esse protocolo acarreta uma rígida alternância de fluxo: primeiro o transmissor envia um quadro, depois o receptor envia outro; em seguida, o transmissor envia mais um quadro, e então o receptor envia outro, e assim por diante. Um canal físico half-duplex seria suficiente nesse caso.

A exemplo do protocolo 1, o transmissor começa extraindo um pacote da camada de rede, utilizando-o para criar um quadro que em seguida é transmitido a seu destino. Todavia, agora, ao contrário do que ocorre no protocolo 1, o transmissor deve aguardar a chegada de um quadro de confirmação antes de tornar a entrar em loop e buscar o próximo pacote da camada de rede. A camada de enlace de dados do transmissor não precisa sequer inspecionar o quadro recebido, pois só há uma possibilidade: ele é sempre uma confirmação.

A única diferença entre *receiver1* e *receiver2* é que, após entregar um pacote à camada de rede, o *receiver2* envia um quadro de confirmação de volta ao transmissor, antes de entrar mais uma vez no loop de espera. Como apenas a chegada do quadro de volta ao transmissor é importante, e não seu conteúdo, o receptor não precisa incluir qualquer informação específica no quadro.

Acrescentando correção de erros: números de sequência e ARQ

Agora, vamos considerar a situação normal de um canal de comunicação no qual ocorrem erros. Os quadros podem ser danificados ou completamente perdidos. No entanto, supomos que, se um quadro for danificado em trânsito, o hardware receptor detectará essa ocorrência ao calcular o checksum. Se o quadro for danificado de tal forma que o checksum nunca esteja correto – uma possibilidade muito improvável –, o protocolo em questão (e todos os outros protocolos) poderá apresentar falhas (i.e., poderá entregar um pacote incorreto à camada de rede).


```
/* O protocolo 2 (stop-and-wait) também implementa um fluxo de dados unidirecional entre
   o transmissor e o receptor. Presume-se mais uma vez que o canal de comunicação é
   totalmente livre de erros, como no protocolo 1. No entanto, dessa vez, o receptor tem buffer
   e velocidade de processamento finitos; portanto, o protocolo deverá impedir explicitamente
   que o transmissor sobrecarregue o receptor enviando dados mais rapidamente do que ele é
   capaz de processar. */
typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender2(void)
{
   frame s;                          /* buffer para um quadro de saída */
   packet buffer;                    /* buffer para um pacote de saída */
   event_type event;                 /* frame_arrival é a única possibilidade */
   while (true) {
       from_network_layer(&buffer);  /* apanha algo para enviar */
       s.info = buffer;              /* copia para s, para transmissão */
       to_physical_layer(&s);        /* pequeno quadro de adeus */
       wait_for_event(&event);       /* não avança até um sinal verde */
   }
}
void receiver2(void)
{
   frame r, s;                       /* buffers para quadros */
   event_type event;                 /* frame_arrival é a única possibilidade */
   while (true) {
       wait_for_event(&event);       /* a única possibilidade é frame_arrival */
       from_physical_layer(&r);      /* apanha o quadro de entrada */
       to_network_layer(&r.info);    /* passa os dados para a camada de rede */
       to_physical_layer(&s);        /* envia quadro fictício para acordar o transmissor */
   }
}
```

Figura 3.13 Um protocolo simplex stop-and-wait.

À primeira vista, pode parecer que uma variação do protocolo 2 seria viável com a inclusão de um timer. O transmissor poderia enviar um quadro, mas o receptor só enviaria um quadro de confirmação se os dados fossem recebidos corretamente. Se um quadro danificado chegasse ao receptor, ele seria descartado. Após certo tempo, o transmissor alcançaria seu timeout e enviaria o quadro mais uma vez. Esse processo seria repetido até que o quadro finalmente chegasse intacto.

Esse esquema tem uma falha fatal. Pense no problema e tente descobrir o que poderia estar errado antes de continuar a leitura.

Para verificar o que poderia estar errado, lembre-se de que a função dos processos da camada de enlace de dados é oferecer comunicações transparentes e livres de erros entre os processos da camada de rede. A camada de rede da máquina *A* envia uma série de pacotes à camada de enlace de dados da mesma máquina. Esta, por sua vez, deve se certificar de que a camada de enlace de dados da máquina *B* enviará uma série idêntica de pacotes à camada de rede da mesma máquina. Em particular, a camada de rede da máquina *B* não tem como saber se um pacote foi perdido ou duplicado; portanto, a camada de enlace de dados deve garantir que nenhuma combinação de erros de transmissão, mesmo improvável, possa fazer um pacote duplicado ser entregue à camada de rede.

Considere a seguinte situação:

1. A camada de rede de *A* envia o pacote 1 à sua camada de enlace de dados. O pacote é corretamente recebido em *B* e repassado à camada de rede de *B*. *B* envia um quadro de confirmação de volta a *A*.
2. O quadro de confirmação se perde por completo. Ele simplesmente nunca chega ao destino. Tudo seria muito mais simples se o canal tivesse adulterado e perdido apenas quadros de dados, não quadros de controle. No entanto, para nossa tristeza, o canal não faz distinção entre quadros.
3. Finalmente, a camada de enlace de dados de *A* tem seu limite de tempo esgotado. Como não recebeu uma confirmação, ela presume (incorretamente) que seu quadro de dados se perdeu ou foi danificado e envia mais uma vez o quadro contendo o pacote 1.
4. O quadro duplicado também chega perfeitamente à camada de enlace de dados de *B* e é repassado de imediato, sem maiores problemas, à sua camada de rede. Caso *A* esteja enviando um arquivo a *B*, uma parte do

arquivo será duplicada (i.e., a cópia do arquivo criado por *B* estará incorreta e o erro não será detectado). Em outras palavras, o protocolo falhará.

Certamente, precisamos proporcionar ao receptor alguma forma de distinguir entre um quadro que ele está recebendo pela primeira vez e uma retransmissão. A maneira mais óbvia de conseguir isso é fazer o transmissor incluir um número de sequência no cabeçalho de cada quadro enviado. Dessa forma, o receptor poderá verificar o número de sequência de cada quadro recebido para confirmar se esse é um novo quadro ou se é uma cópia a ser descartada.

Como o protocolo deve ser correto e o campo de número de sequência no cabeçalho provavelmente é pequeno para usar o enlace de modo eficiente, surge a seguinte pergunta: qual é a quantidade mínima de bits necessária para o número de sequência? O cabeçalho poderia oferecer 1 bit, alguns bits, um byte ou múltiplos bytes para um número de sequência, dependendo do protocolo. O importante é que ele deve transportar números de sequência grandes o suficientes para que o protocolo funcione corretamente, ou o protocolo não terá valor.

A única ambiguidade nesse protocolo ocorre entre um quadro, m, e seu sucessor direto, $m + 1$. Se o quadro m tiver sido perdido ou danificado, o receptor não o confirmará; portanto, o transmissor continuará tentando enviá-lo. Uma vez que o quadro tenha sido corretamente recebido, o receptor enviará uma confirmação de volta ao transmissor. E aqui surge um problema potencial. Dependendo do fato de o quadro de confirmação voltar ao transmissor corretamente ou não, o transmissor poderá tentar enviar m ou $m + 1$.

No transmissor, o evento que dispara a transmissão do quadro $m + 1$ é a chegada de uma confirmação para o quadro m. Mas essa situação implica que $m - 1$ foi recebido corretamente e, além disso, que sua confirmação também foi recebida corretamente pelo transmissor. Caso contrário, o transmissor não teria iniciado com m, muito menos estaria considerando $m + 1$. Por conseguinte, a única ambiguidade é entre um quadro e seu predecessor ou sucessor imediato, não entre os próprios predecessores ou sucessores.

Um número de sequência de 1 bit (0 ou 1) é, portanto, suficiente. A cada instante, o receptor espera o próximo número de sequência. Quando chega um quadro contendo um número de sequência correto, ele é aceito e repassado à camada de rede, e depois confirmado. Em seguida, o número de sequência esperado é incrementado na base 2 (ou seja, 0 passa a ser 1 e 1 passa a ser 0). Qualquer quadro recebido que contenha o número de sequência errado será rejeitado por ser considerado uma cópia. Contudo, a última confirmação válida é repetida, de forma que o transmissor finalmente pode descobrir que o quadro foi recebido.

Um exemplo desse tipo de protocolo é mostrado na Figura 3.14. Os protocolos nos quais o transmissor espera por uma confirmação positiva antes de passar para o próximo item de dados frequentemente são chamados de **solicitação de repetição automática**, ou **ARQ** (**Automatic Repeat reQuest**), ou **confirmação positiva com retransmissão**, ou **PAR** (**Positive Acknowledgement with Retransmission**). A exemplo do protocolo 2, ele também transmite dados em apenas um sentido.

O protocolo 3 difere de seus predecessores pelo fato de tanto o transmissor quanto o receptor terem uma variável cujo valor é memorizado enquanto a camada de enlace de dados se encontra em estado de espera. Em *next_frame_to_send*, o transmissor memoriza o número de sequência do próximo quadro a ser enviado; em *frame_expected*, o receptor memoriza o número de sequência do próximo quadro esperado. Cada protocolo tem uma breve fase de inicialização antes de entrar no loop infinito.

Após enviar um quadro, o transmissor ativa o timer. Caso já esteja ativado, ele será reiniciado para permitir a contagem de outro intervalo, o qual deve ser escolhido de forma que haja tempo suficiente para o quadro chegar ao receptor, para o receptor processá-lo na pior das hipóteses e para o quadro de confirmação ser enviado de volta ao transmissor. Somente quando o intervalo tiver se esgotado, poderemos supor com segurança que o quadro transmitido ou sua confirmação se perdeu, e que será necessário enviar uma cópia. Se o intervalo de timeout for definido com um valor curto demais, o transmissor enviará quadros desnecessários. Embora não afetem a exatidão do protocolo, esses quadros extras prejudicarão o desempenho.

Depois de transmitir um quadro e ativar o timer, o transmissor espera que algo interessante aconteça. Existem apenas três possibilidades: o quadro de confirmação chegar sem danos, o quadro de confirmação chegar com erro ou o timer expirar. Se uma confirmação válida for recebida, o transmissor buscará o próximo pacote em sua camada de rede e o colocará no buffer, substituindo o pacote anterior. Ele também aumentará o número de sequência. Se for recebido um quadro com erro ou se o timer expirar, o buffer e o número de sequência permanecerão inalterados, de modo que uma cópia do quadro poderá ser enviada. De qualquer forma, o conteúdo do buffer (tanto o próximo pacote como uma cópia) é enviado em seguida.

Quando um quadro válido chega ao receptor, seu número de sequência é conferido, para verificar se ele é uma cópia. Se não for, o quadro será aceito, enviado à camada de rede, e uma confirmação será gerada. Cópias e quadros danificados não serão repassados à camada de rede, mas eles fazem o último quadro recebido corretamente ser confirmado para sinalizar ao transmissor para avançar ao próximo quadro ou retransmitir um quadro danificado.

3.4 MELHORANDO A EFICIÊNCIA

Nos protocolos apresentados anteriormente, os quadros de dados eram transmitidos em apenas um sentido. Em situações


```
/* O protocolo 3 (PAR) permite que dados unidirecionais fluam por um canal não confiável. */

#define MAX_SEQ 1                       /* deve ser 1 para o protocolo 3 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;            /* número seq do próximo quadro de saída */
  frame s;                              /* variável auxiliar */
  packet buffer;                        /* buffer para pacote de saída */
  event_type event;

  next_frame_to_send = 0;               /* inicia números de sequência de saída */
  from_network_layer(&buffer);          /* busca primeiro pacote */
  while (true) {
    s.info = buffer;                    /* monta um quadro para transmissão */
    s.seq = next_frame_to_send;         /* insere número de sequência no quadro */
    to_physical_layer(&s);              /* envia o quadro */
    start_timer(s.seq);                 /* se a resposta levar muito tempo, timeout */
    wait_for_event(&event);             /* frame_arrival, cksum_err, timeout */
    if (event == frame_arrival) {
      from_physical_layer(&s);          /* obtém a confirmação */
      if (s.ack == next_frame_to_send) {
        stop_timer(s.ack);              /* desliga o timer */
        from_network_layer(&buffer);    /* pega o próximo quadro a enviar */
        inc(next_frame_to_send);        /* inverte next_frame_to_send */
      }
    }
  }
}
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
    wait_for_event(&event);             /* possibilidades: frame_arrival, cksum_err */
    if (event == frame_arrival) {       /* chegou um quadro válido */
      from_physical_layer(&r);          /* pega quadro recém-chegado */
      if (r.seq == frame_expected) {    /* é isso que estávamos esperando */
        to_network_layer(&r.info);      /* passa os dados para a camada de rede */
        inc(frame_expected);            /* da próxima vez, espera outro número de sequência nr */
      }
      s.ack = 1 - frame_expected;       /* diz qual quadro está sendo confirmado */
      to_physical_layer(&s);            /* envia confirmação */
    }
  }
}
```

Figura 3.14 Uma confirmação positiva com protocolo de retransmissão.

mais práticas, há necessidade de transmitir dados em ambos os sentidos. Além disso, a camada de enlace pode ser mais eficiente se puder enviar vários quadros ao mesmo tempo antes de receber uma confirmação. Vamos explorar esses dois conceitos em seguida, para depois oferecer vários exemplos de protocolos que alcançam esses objetivos.

3.4.1 Objetivo: transmissão bidirecional, múltiplos quadros em andamento

Em seguida, vamos explicar um conceito chamado piggybacking, que pode ajudar um protocolo da camada de enlace a alcançar transmissão bidirecional, e um conceito

chamado janela deslizante, que pode melhorar a eficiência da transmissão ao permitir que o transmissor tenha vários bytes em andamento.

Transmissão bidirecional: piggybacking

Você pode obter uma transmissão de dados full-duplex definindo dois canais de comunicação distintos e cada um deles usando um enlace separado para um tráfego de dados simplex (em diferentes sentidos). Cada enlace é composto de um canal "direto" (para dados) e de um canal "reverso" (para confirmações). Em ambos os casos, a capacidade do canal reverso é quase totalmente desperdiçada.

Uma ideia melhor é usar o mesmo circuito para dados em ambos os sentidos. Afinal, nos protocolos 2 e 3 ele já estava sendo usado para transmitir quadros em ambos os sentidos, e o canal reverso normalmente tem a mesma capacidade do canal direto. Nesse modelo, os quadros de dados enviados de *A* para *B* são misturados com os quadros de confirmação enviados de *A* para *B*. Ao verificar o campo *kind* no cabeçalho de um quadro recebido, o receptor pode identificar se o quadro é de dados ou de confirmação.

Apesar de o entrelaçamento de quadros de dados e de controle no mesmo circuito representar um grande avanço em relação ao uso de dois enlaces físicos separados, ainda é possível introduzir outra melhoria. Quando um quadro de dados chega a seu destino, em vez de enviar imediatamente um quadro de controle separado, o receptor se contém e espera até a camada de rede enviar o próximo pacote. A confirmação é acrescentada ao quadro de dados que está sendo enviado (por meio do campo *ack* do cabeçalho do quadro). Na verdade, a confirmação pega carona no próximo quadro de dados que estiver sendo enviado. A técnica de retardar temporariamente as confirmações e enviá-las com o próximo quadro de dados é conhecida pelo nome de **piggybacking** (**pegar carona**).

A principal vantagem do piggybacking em relação ao envio de quadros de confirmação distintos é a melhor utilização da largura de banda disponível para o canal. O campo *ack* do cabeçalho do quadro precisa de apenas alguns bits, enquanto um quadro separado precisaria do cabeçalho, da confirmação e do checksum. Além disso, um número menor de quadros enviados significa uma carga de processamento menor no receptor. No próximo protocolo a ser examinado, o campo de piggyback necessita apenas de um bit no cabeçalho do quadro. Em geral, ele raramente precisa de mais que alguns bits.

No entanto, o piggybacking introduz uma complicação que não existe em confirmações separadas. Quanto tempo a camada de enlace de dados deve esperar por um pacote ao qual deverá acrescentar a confirmação? Se a camada de enlace de dados esperar durante um intervalo maior que o permitido pelo timeout do transmissor, o quadro será retransmitido, o que invalidará todo o processo de confirmação. Se a camada de enlace de dados fosse um oráculo e pudesse prever o futuro, ela saberia quando o próximo pacote da camada de rede estivesse chegando e poderia decidir entre esperar por ele e enviar imediatamente uma confirmação separada, dependendo da duração prevista do tempo de espera. É óbvio que a camada de enlace de dados não é capaz de prever o futuro; portanto, ela deve recorrer a algum esquema ad hoc, como esperar durante um número fixo de milissegundos. Se um novo pacote chegar rapidamente, a confirmação será acrescentada a ele; caso contrário, se nenhum pacote tiver chegado até o final desse intervalo, a camada de enlace de dados simplesmente enviará um quadro de confirmação separado.

Janelas deslizantes

Os três protocolos seguintes são bidirecionais e pertencem a uma classe identificada como protocolos de **janela deslizante**. Eles apresentam diferenças em termos de eficiência, complexidade e requisitos de buffer, como discutiremos adiante. Neles, assim como em todos os protocolos de janela deslizante, cada quadro enviado contém um número de sequência, variando de 0 até algum valor máximo. Em geral, o valor máximo é $2^n - 1$, de forma que o número de sequência caiba exatamente em um campo de n bits. O protocolo de janela deslizante stop-and-wait utiliza $n = 1$, restringindo os números de sequência a 0 e 1; no entanto, versões mais sofisticadas podem usar um valor arbitrário de n.

A essência de todos os protocolos de janela deslizante é o fato de que, em qualquer instante, o transmissor mantém um conjunto de números de sequência correspondentes a quadros que ele pode enviar. Dizemos que esses quadros estão reunidos na **janela de transmissão**. Da mesma forma, o receptor mantém uma **janela de recepção** correspondente ao conjunto de quadros que está apto a aceitar. As janelas do transmissor e do receptor não precisam ter os mesmos limites superior e inferior ou o mesmo tamanho. Em alguns protocolos, essas janelas têm tamanho fixo, mas em outros elas podem aumentar ou diminuir à medida que os quadros são enviados e recebidos.

Apesar de esses protocolos permitirem que a camada de enlace de dados tenha mais liberdade em relação à ordem em que pode enviar e receber quadros, definitivamente não descartamos a exigência de o protocolo entregar os pacotes à camada de rede na mesma ordem em que eles foram repassados à camada de enlace de dados na máquina transmissora. Outra exigência que não mudou é que o canal de comunicação física seja "como nos fios", ou seja, que entregue todos os quadros na ordem em que eles são enviados.

Os números de sequência contidos na janela do transmissor representam quadros que foram ou que podem ser enviados, mas que ainda não foram confirmados. Sempre que chega um novo pacote da camada de rede, ele recebe o próximo número de sequência mais alto, e o limite superior da janela é incrementado em uma unidade. Quando uma confirmação é recebida, o limite inferior é incrementado em uma unidade. Dessa forma, a janela mantém continuamente uma lista de quadros não confirmados. A Figura 3.15 mostra um exemplo.

Tendo em vista que os quadros presentes atualmente na janela do transmissor podem ser perdidos ou danificados em trânsito, o transmissor deve mantê-los em sua memória para que a retransmissão seja possível. Assim, se o tamanho máximo da janela for n, o transmissor precisará de n buffers para armazenar os quadros não confirmados. Se a janela chegar a seu tamanho máximo, a camada de enlace de dados do transmissor será obrigada a desativar a camada de rede até que outro buffer esteja livre.

O tamanho da janela da camada de enlace de dados receptora corresponde aos quadros que ela é capaz de aceitar. Qualquer quadro que ficar dentro da janela é colocado no buffer do receptor. Quando for recebido um quadro cujo número de sequência for igual ao limite inferior da janela, ele será repassado à camada de rede, será gerada uma confirmação e a janela será incrementada em uma unidade. Qualquer quadro fora da janela é descartado. Em todos esses casos, uma confirmação subsequente é gerada para que o transmissor descubra como proceder. Observe que um tamanho de janela igual a 1 significa que a camada de enlace de dados só aceita quadros em ordem, mas para janelas maiores isso não é verdade. A camada de rede, ao contrário, sempre recebe dados na ordem adequada, independentemente do tamanho da janela da camada de enlace de dados.

A Figura 3.15 mostra um exemplo com um tamanho máximo de janela igual a 1. Inicialmente, não há quadros pendentes; portanto, os limites inferior e superior da janela do transmissor são iguais, mas, à medida que o tempo passa, a situação se desenvolve da maneira mostrada. Diferentemente da janela do transmissor, a janela do receptor sempre permanece em seu tamanho inicial, incrementando-se à medida que o próximo quadro é aceito e entregue à camada de rede.

3.4.2 Exemplos de protocolos full-duplex de janela deslizante

Agora, vejamos alguns exemplos de um protocolo de janela deslizante simples, de um único bit, além de protocolos que podem tratar da retransmissão de quadros com erro quando vários quadros estão a caminho.

Janela deslizante de um bit

Antes de abordarmos o caso geral, vamos examinar primeiro um protocolo de janela deslizante com um tamanho máximo de janela igual a 1. Esse tipo de protocolo utiliza o stop-and-wait, pois o transmissor envia um quadro e aguarda sua confirmação antes de enviar o quadro seguinte.

A Figura 3.16 representa esse tipo de protocolo. Assim como os demais, ele começa definindo algumas variáveis: *next_frame_to_send* informa qual quadro o transmissor está tentando enviar. De modo semelhante, *frame_expected* informa que quadro o receptor está esperando. Nos dois casos, 0 e 1 são as únicas possibilidades.

Normalmente, uma das duas camadas de enlace de dados, do transmissor ou do receptor, inicia e envia o primeiro quadro. Em outras palavras, apenas um dos programas da camada de enlace de dados pode conter as chamadas das

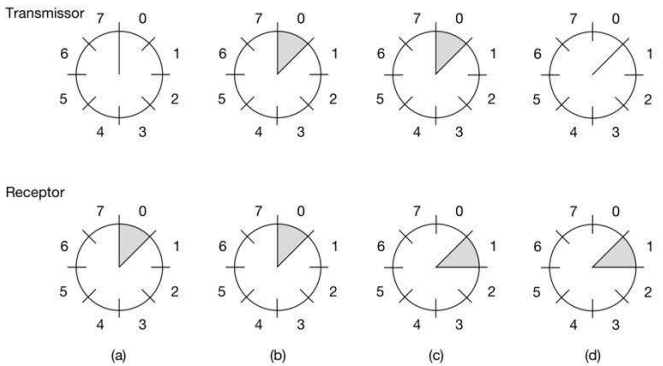

Figura 3.15 Uma janela deslizante de tamanho 1, com um número de sequência de 3 bits. (a) Inicialmente. (b) Depois que o primeiro quadro é enviado. (c) Depois que o primeiro quadro é recebido. (d) Depois que a primeira confirmação é recebida.

```
/* O protocolo 4 (janela deslizante) é bidirecional. */

#define MAX_SEQ 1                       /* deve ser 1 para protocolo 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send;            /* 0 ou 1 apenas */
  seq_nr frame_expected                 /* 0 ou 1 apenas */
  frame r, s;                           /* variáveis auxiliares */
  packet buffer;                        /* pacote atual sendo enviado */
  event_type event;
  next_frame_to_send = 0;               /* próximo quadro no fluxo de saída */
  frame_expected = 0;                   /* quadro esperado em seguida */
  from_network_layer(&buffer);          /* busca um pacote da camada de rede */
  s.info = buffer;                      /* prepara para enviar quadro inicial */
  s.seq = next_frame_to_send;           /* insere número de sequência no quadro */
  s.ack = 1 – frame_expected;           /* confirmação acrescentada */
  to_physical_layer(&s);                /* transmite o quadro */
  start_timer(s.seq);                   /* inicia a execução do timer */
  while (true) {
    wait_for_event(&event);             /* frame_arrival, cksum_err ou timeout */
    if (event == frame_arrival) {       /* um quadro chegou intacto */
      from_physical_layer(&r);          /* vai pegá-lo */
      if (r.seq == frame_expected) {    /* trata do fluxo de quadros que chega */
          to_network_layer(&r.info);    /* passa pacote à camada de rede */
          inc(frame_expected);          /* inverte número de sequência esperado em seguida */
      }
      if (r.ack == next_frame_to_send) {/* trata do fluxo de quadros que sai */
          stop_timer(r.ack);            /* desliga o timer */
          from_network_layer(&buffer);  /* busca novo pac. da camada de rede */
          inc(next_frame_to_send);      /* inverte número de sequência do transmissor */
      }
    }
    s.info = buffer;                    /* constrói quadro de saída */
    s.seq = next_frame_to_send;         /* insere número de sequência nele */
    s.ack = 1 – frame_expected;         /* número de sequência do último quadro recebido */
    to_physical_layer(&s);              /* transmite um quadro */
    start_timer(s.seq);                 /* inicia a execução do timer */
  }
}
```

Figura 3.16 Um protocolo de janela deslizante de um bit.

funções *to_physical_layer* e *start_timer* fora do loop principal. A máquina que inicia busca o primeiro pacote em sua camada de rede, constrói um quadro a partir dele e o envia. Quando esse (ou qualquer) quadro chega ao destino, a camada de enlace de dados receptora verifica se ele é uma cópia, como ocorreu no protocolo 3. Se o quadro for o esperado, ele será repassado à camada de rede e a janela do receptor será deslocada para cima.

O campo de confirmação contém o número do último quadro recebido sem erro. Se esse número estiver de acordo com o número de sequência do quadro que o transmissor está tentando enviar, o transmissor saberá que já cuidou do quadro armazenado em *buffer* e poderá buscar o pacote seguinte em sua camada de rede. Se o número de sequência for discordante, o transmissor deverá continuar tentando enviar o mesmo quadro. Sempre que um quadro é recebido, outro quadro também é enviado de volta.

Agora, vamos examinar o protocolo 4 para ver quanto ele é flexível em relação a situações patológicas. Suponha que o computador A esteja tentando enviar seu quadro 0 ao computador B e que B esteja tentando enviar seu quadro 0 ao computador A. Imagine que A envia um quadro a B, mas o intervalo de timeout de A é curto demais. Consequentemente, A pode completar o timeout repetidas vezes, enviando uma série de quadros idênticos, todos com $seq = 0$ e $ack = 1$.

Quando o primeiro quadro válido chegar a B, ele será aceito e *frame_expected* será definido como 1. Todos os quadros subsequentes serão rejeitados, porque B agora está esperando quadros com número de sequência 1, e não 0.

Além disso, como todas as cópias têm $ack = 1$ e B ainda está aguardando uma confirmação de 0, B não buscará um novo pacote em sua camada de rede.

Após a chegada de todas as cópias rejeitadas, B enviará um quadro para A contendo $seq = 0$ e $ack = 0$. Por fim, um desses quadros chegará sem erros à máquina A, fazendo-a começar a enviar o próximo pacote. Nenhuma combinação de quadros perdidos ou timeouts prematuros pode fazer o protocolo entregar pacotes duplicados à camada de rede, ignorar um pacote ou chegar a um impasse. O protocolo está correto.

Entretanto, para mostrar quão sutis as interações entre protocolos podem ser, surgirá uma situação peculiar se os dois lados enviarem simultaneamente um pacote inicial. Essa dificuldade de sincronização está ilustrada na Figura 3.17. Na parte (a), é exibida a operação normal do protocolo. Na parte (b), observamos a peculiaridade. Se B esperar pelo primeiro quadro de A antes de enviar um de seus quadros, a sequência será a da parte (a) e todos os quadros serão aceitos.

Contudo, se A e B iniciarem a comunicação ao mesmo tempo, seus primeiros quadros se cruzarão e as camadas de enlace de dados entrarão na situação (b). Em (a), cada quadro recebido traz um novo pacote para a camada de rede; não há cópias. Em (b), metade dos quadros contém cópias, embora não haja erros de transmissão. Situações similares podem ocorrer como resultado de timeouts prematuros, mesmo quando está claro que um lado começa primeiro. Na verdade, se ocorrerem vários timeouts prematuros, os quadros poderão ser enviados três ou mais vezes, desperdiçando uma largura de banda valiosa.

Go-back-n

Até agora estávamos supondo implicitamente que era insignificante o tempo de transmissão necessário para a chegada de um quadro até o receptor, somado ao tempo de transmissão para o retorno da confirmação. Às vezes, essa suposição é nitidamente falsa. Nessas situações, o longo tempo de viagem de ida e volta pode ter implicações importantes para a eficiência da utilização da largura de banda. Como exemplo, considere um canal de satélite de 50 kbps com um atraso de propagação de ida e volta de 500 ms. Vamos imaginar uma tentativa de usar o protocolo 4 para enviar quadros de 1.000 bits pelo satélite. Em $t = 0$, o transmissor começa a enviar o primeiro quadro. Em $t = 20$ ms, o quadro já foi completamente enviado. Até $t = 270$ ms, o quadro ainda não chegou completamente ao receptor, e até $t = 520$ ms, na melhor das hipóteses, a confirmação não terá voltado ao transmissor (sem nenhum tempo de espera no receptor e com um quadro de confirmação curto). Isso significa que o transmissor esteve bloqueado durante 500/520 ou 96% (i.e., apenas 4% da largura de banda disponível foi utilizada). É claro que a combinação de um longo tempo de trânsito, alta largura de banda e pequeno comprimento de quadro é desastrosa em termos de eficiência.

O problema descrito pode ser visto como uma consequência da regra que exige que um transmissor espere por uma confirmação antes de enviar outro quadro. Se essa restrição não for rigorosa, podemos obter uma eficiência muito melhor. Basicamente, a solução está em permitir que o transmissor envie até w quadros antes do bloqueio, e não apenas 1. Com uma escolha apropriada de w, o transmissor será capaz de transmitir quadros continuamente, pois as confirmações chegarão aos quadros anteriores antes que a janela se complete, impedindo o bloqueio do transmissor.

Para achar um valor apropriado para w, precisamos saber quantos quadros cabem dentro do canal à medida que se propagam do transmissor ao receptor. Essa capacidade é

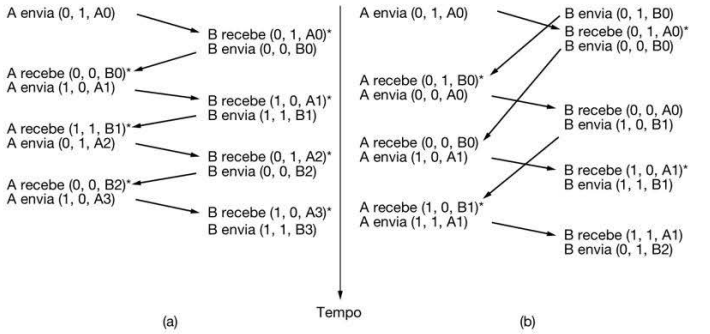

Figura 3.17 Dois cenários referentes ao protocolo 4. (a) Caso normal. (b) Caso anormal. A notação é sequência, confirmação, número do pacote. Um asterisco indica onde uma camada de rede aceita um pacote.

determinada pela largura de banda em bits/s, multiplicada pelo tempo de trânsito em mão única, ou **produto largura de banda-atraso** do enlace. Podemos dividir essa quantidade pelo número de bits em um quadro para expressá-lo como um número de quadros. Chame essa quantidade de *BD*. Então, *w* deve ser definido como $2BD + 1$. O dobro da largura de banda-atraso é o número de quadros que podem estar pendentes se o transmissor enviar quadros continuamente quando o tempo de ida e volta para receber uma confirmação for considerado. O "+1" é porque um quadro de confirmação não será enviado antes que um quadro completo seja recebido.

Para o enlace do exemplo com uma largura de banda de 50 kbps e um tempo de trânsito unidirecional de 250 ms, o produto largura de banda-atraso é de 12,5 kbits ou 12,5 quadros de 1.000 bits cada um. $2BD + 1$ significa, então, 26 quadros. Suponha que o transmissor comece enviando o quadro 0, como antes, e transmita um novo quadro a cada 20 ms. Quando ele tiver terminado de enviar 26 quadros, em $t = 520$ ms, a confirmação para o quadro 0 terá acabado de chegar. Depois disso, as confirmações chegarão a cada 20 ms, de modo que o transmissor sempre terá permissão para continuar assim que precisar. A partir desse ponto, 25 ou 26 quadros não confirmados sempre estarão pendentes. Em outras palavras, o tamanho de janela máxima do transmissor é 26.

Para tamanhos de janela menores, a utilização do enlace será menor que 100%, pois o transmissor às vezes será bloqueado. Podemos escrever a utilização como a fração de tempo em que o transmissor não está bloqueado:

$$\text{utilização do enlace} \leq \frac{w}{1 + 2BD}$$

Esse valor é um limite superior, pois não leva em consideração nenhum tempo de processamento de quadro e trata o quadro de confirmação como tendo tamanho zero, pois ele normalmente é curto. A equação mostra a necessidade de ter uma janela *w* grande sempre que o produto de largura de banda-atraso for grande. Se o atraso for alto, o transmissor rapidamente esgotará sua janela, mesmo para uma largura de banda moderada, como no exemplo do satélite. Se a largura de banda for alta, mesmo para um atraso moderado, o transmissor esgotará sua janela rapidamente, a menos que tenha uma janela grande (p. ex., um enlace de 1 Gbps com atraso de 1 ms mantém 1 megabit). Com o stop-and-wait, para o qual $w = 1$, se houver um atraso de propagação de até mesmo um quadro, a eficiência será menor que 50%.

Essa técnica de manter vários quadros pendentes é um exemplo de **pipelining**. O pipelining de quadros em um canal de comunicação não confiável faz surgir algumas questões muito sérias. Primeiro, o que acontecerá se um quadro em meio a um longo fluxo for danificado ou perdido? Um grande número de quadros sucessivos chegará ao receptor antes mesmo de o transmissor descobrir que algo está errado. Quando um quadro danificado chega ao receptor, sem dúvida ele deve ser descartado. No entanto, o que o receptor deve fazer com todos os quadros corretos que o seguem? Lembre-se de que a camada de enlace de dados receptora é obrigada a entregar pacotes à camada de rede em sequência.

Há duas estratégias básicas para lidar com erros na presença do pipelining, ambas mostradas na Figura 3.18.

Em uma opção denominada **go-back-n**, o receptor simplesmente descarta todos os quadros subsequentes e não envia nenhuma confirmação referente a eles. Essa estratégia corresponde a uma janela de recepção de tamanho 1. Em outras palavras, a camada de enlace de dados se recusa a aceitar qualquer quadro, exceto o próximo, que ela tem de entregar à camada de rede. Se a janela do transmissor for totalmente preenchida antes de o timer encerrar a contagem, o pipeline começará a se esvaziar. Consequentemente, o transmissor interromperá a transmissão e retransmitirá todos os quadros não confirmados em ordem, começando pelo quadro danificado ou perdido. Essa abordagem poderá desperdiçar uma grande quantidade de largura de banda se a taxa de erros for alta.

Na Figura 3.18(a), vemos o go-back-n para o caso em que a janela do receptor é 1. Os quadros 0 e 1 são corretamente recebidos e confirmados. Contudo, o quadro 2 está danificado ou perdido. O transmissor, desavisado desse problema, continua a enviar quadros até expirar o timer correspondente ao quadro 2. Em seguida, ele volta até o quadro 2 e começa tudo de novo a partir dele, enviando, mais uma vez, os quadros 2, 3, 4, e assim por diante.

Retransmissão seletiva

O protocolo go-back-n funciona bem se os erros forem raros, mas, se a linha for ruim, isso desperdiçará muita largura de banda nos quadros retransmitidos. Precisamos fazer algo melhor – e isso é possível. Uma alternativa é o protocolo de **retransmissão seletiva**, que permite que o receptor aceite e mantenha os quadros corretos em um buffer após um quadro danificado ou perdido.

Quando ela é utilizada, um quadro incorreto recebido é descartado, mas os quadros sem defeitos posteriores são aceitos e mantidos no buffer. Quando o transmissor chega ao timeout, apenas o quadro não confirmado mais antigo é retransmitido. Se ele chegar corretamente, o receptor poderá entregar à camada de rede, em sequência, todos os quadros que armazenou no buffer. A estratégia de retransmissão seletiva corresponde a uma janela receptora maior que 1. Caso a janela seja muito grande, essa abordagem poderá exigir um volume de memória muito grande da camada de enlace de dados.

Com frequência, a retransmissão seletiva é combinada com a ação de fazer o receptor enviar uma confirmação negativa, ou NAK (negative acknowledgement), ao detectar

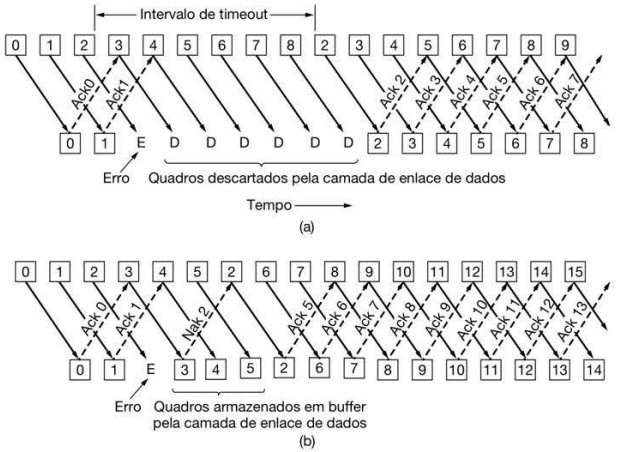

Figura 3.18 Pipelining e recuperação de erros. Efeito de um erro quando (a) o tamanho da janela receptora é igual a 1 e (b) quando o tamanho da janela receptora é grande.

um erro, por exemplo, quando receber um erro de checksum ou um quadro fora de sequência. As NAKs estimulam a retransmissão antes de expirar o timer correspondente e, desse modo, melhoram o desempenho.

Na Figura 3.18(b), os quadros 0 e 1 são mais uma vez recebidos e confirmados corretamente, e o quadro 2 é perdido. Quando o quadro 3 chega ao receptor, a camada de enlace de dados do receptor percebe que perdeu um quadro e, assim, envia de volta uma NAK correspondente ao quadro 2, mas armazena o quadro 3 no buffer. Quando os quadros 4 e 5 chegam, eles também são inseridos no buffer pela camada de enlace de dados, em vez de serem repassados à camada de rede. Por fim, a NAK do quadro 2 volta ao transmissor, que o retransmite de imediato. Quando esse quadro chega, a camada de enlace de dados fica com os quadros 2, 3, 4 e 5, e pode repassar todos eles à camada de rede na ordem correta. Ela também pode confirmar todos os quadros, inclusive até o 5, como mostra a figura. Se a NAK se perder, o transmissor chegará ao timeout correspondente ao quadro 2 e o enviará (e apenas esse quadro) por sua própria iniciativa, mas isso pode acontecer um pouco mais tarde.

Esses dois enfoques alternativos são dilemas entre uso eficiente de largura de banda e espaço no buffer da camada de enlace de dados. Dependendo de qual recurso é mais escasso, um ou outro poderá ser usado. A Figura 3.19 mostra um protocolo go-back-n no qual a camada de enlace de dados receptora aceita apenas quadros em ordem; os quadros que vierem depois de um quadro com erro serão descartados. Nesse protocolo, pela primeira vez abandonamos a suposição de que a camada de rede sempre tem um suprimento infinito de pacotes a enviar. Quando a camada de rede tem um pacote que deseja enviar, ela pode provocar a ocorrência de um evento *network_layer_ready*. Entretanto, para reforçar o limite de controle de fluxo sobre a janela do transmissor ou o número de quadros não confirmados pendentes em qualquer instante, a camada de enlace de dados deve ser capaz de proibir a camada de rede de sobrecarregá-la com mais trabalho. As funções de biblioteca *enable_network_layer* e *disable_network_layer* executam essa funcionalidade.

Observe que o número máximo de quadros que podem estar pendentes em qualquer instante não é o mesmo que o tamanho do espaço do número de sequência. Para go-back-n, *MAX_SEQ* quadros podem estar pendentes em qualquer instante, mesmo que haja $MAX\_SEQ + 1$ números de sequência distintos (que são 0, 1, 2,..., *MAX_SEQ*). Veremos uma restrição ainda maior para o protocolo seguinte, a retransmissão seletiva. Para saber por que essa restrição é necessária, considere a situação a seguir, com $MAX\_SEQ = 7$:

1. O transmissor envia quadros de 0 a 7.
2. Uma confirmação para o quadro 7 volta de carona (piggyback) ao transmissor.
3. O transmissor envia mais oito quadros, novamente com números de sequência de 0 a 7.
4. Agora chega outra confirmação de carona correspondente ao quadro 7.

```
/* O protocolo 5 (go-back-n) permite a existência de muitos quadros pendentes. O transmissor poderá transmitir até
   MAX_SEQ quadros sem a necessidade de esperar por uma confirmação. Além disso, ao contrário dos protocolos
   anteriores, não presumimos que a camada de rede esteja sempre recebendo um novo pacote. Em vez disso, a camada
   de rede provoca o evento network_layer_ready quando há um pacote a ser enviado. */

#define MAX_SEQ 7

typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;

#include "protocol.h"


static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Retorna true se a <= b < c de modo circular; caso contrário, false. */
   if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
        return(true);
      else
        return(false);
}

static void send_data(seq_nr frame_nr,seq_nr frame_expected,packet buffer[ ])
{
/* Constrói e envia um quadro de dados. */
   frame s;                                  /* variável auxiliar */
   s.info = buffer[frame_nr];                /* insere pacote no quadro */
   s.seq = frame_nr;                         /* insere número de sequência no quadro */
   s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);   /* confirma piggyback */
   to_physical_layer(&s);                    /* transmite o quadro */
   start_timer(frame_nr);                    /* inicia o timer */
}

void protocol5(void)
{
   seq_nr next_frame_to_send;                /* MAX_SEQ > 1; usado para fluxo de saída */
   seq_nr ack_expected;                      /* quadro mais antigo como ainda não confirmado */
   seq_nr frame_expected;                    /* próximo quadro esperado no fluxo de entrada */
   frame r;                                  /* variável auxiliar */
   packet buffer[MAX_SEQ + 1];               /* buffers para o fluxo de saída */
   seq_nr nbuffered;                         /* número de buffers de saída atualmente em uso */
   seq_nr i;                                 /* usado para indexar array no buffer */
   event_type event;
   enable_network_layer();                   /* permite network_layer_ready events */
   ack_expected = 0;                         /* próximo ack esperado na entrada */
   next_frame_to_send = 0;                   /* próximo quadro saindo */
   frame_expected= 0;                        /* número do quadro esperado na entrada */
   nbuffered = 0;                            /* inicialmente, nenhum pacote no buffer */
   while (true) {
      wait_for_event(&event);                /* quatro possibilidades: ver event_type acima */
```

Figura 3.19 Um protocolo de janela deslizante que utiliza go-back-n. *(Continua)*


```
      switch(event) {
        case network_layer_ready:                /* a camada de rede tem um pacote a enviar */
          /* Aceita, salva e transmite um novo quadro. */
          from_network_layer(&buffer[next_frame_to_send]); /* busca novo pacote */
          nbuffered = nbuffered + 1;             /* expande a janela do transmissor */
          send_data(next_frame_to_send, frame_expected, buffer); /* transmite o quadro */
          inc(next_frame_to_send);               /* avança o limite da janela superior do transmissor */
          break;
        case frame_arrival:                      /* um quadro de dados ou controle chegou */
          from_physical_layer(&r);               /* pega quadro que chegou da camada física */
          if (r.seq == frame_expected) {
          /* Quadros são aceitos apenas em ordem. */
            to_network_layer(&r.info);           /* passa pacote à camada de rede */
            inc(frame_expected);                 /* avança o limite inferior da janela do receptor */
          }
          /* Ack n implica n – 1, n – 2, etc. Verifica isso. */
          while (between(ack_expected, r.ack, next_frame_to_send)) {
            /* Trata o ack piggyback. */
            nbuffered = nbuffered – 1;           /* um quadro a menos no buffer */
            stop_timer(ack_expected);            /* quadro chegou intacto; interrompe timer */
            inc(ack_expected);                   /* ajusta a janela do transmissor */
          }
          break;
        case cksum_err: break;                   /* apenas ignora quadros ruins */
        case timeout:                            /* problema; retransmite todos os quadros pendentes */
          next_frame_to_send = ack_expected;     /* inicia retransmissão aqui */
          for (i = 1; i <= nbuffered; i++) {
              send_data(next_frame_to_send, frame_expected, buffer);   /* reenvia quadro */
              inc(next_frame_to_send);           /* prepara para enviar o quadro seguinte */
          }
      }
      if (nbuffered < MAX_SEQ)
            enable_network_layer();
      else
            disable_network_layer();
   }
}
```

Figura 3.19 *(Continuação)* Um protocolo de janela deslizante que utiliza go-back-n.

A questão é: os oito quadros pertencentes ao segundo lote chegaram com sucesso, ou todos eles se perderam (a contagem descarta os quadros posteriores a um erro, considerando-os perdidos)? Nos dois casos, o receptor enviaria o quadro 7 como confirmação. O transmissor não tem como saber disso. Por essa razão, o número máximo de quadros pendentes deve ser restrito à *MAX_SEQ* (e não *MAX_SEQ* + 1).

Apesar de não armazenar no buffer os quadros recebidos após um quadro com erro, o protocolo 5 não evita totalmente o problema do armazenamento em buffer. Tendo em vista que um transmissor talvez seja obrigado a retransmitir todos os quadros não confirmados em determinado momento no futuro, ele deverá reter todos os quadros transmitidos até ter certeza de que eles foram aceitos pelo receptor.

Quando uma confirmação chega para o quadro n, os quadros $n - 1$, $n - 2$, e assim por diante, também são confirmados de forma automática. Esse tipo de confirmação é chamado de **confirmação cumulativa**. Essa propriedade é especialmente importante nos casos em que alguns dos quadros anteriores que representavam confirmações se perderam ou foram adulterados. Sempre que chega uma confirmação, a camada de enlace de dados verifica se algum buffer pode ser liberado. Se os buffers puderem ser liberados (i.e., se houver espaço disponível na janela), uma camada de rede bloqueada anteriormente poderá ter permissão para provocar mais eventos *network_layer_ready*.

Para esse protocolo, supomos que sempre existe tráfego no sentido inverso, para que as confirmações possam

ser transportadas por piggyback. O protocolo 4 não precisa dessa suposição, pois ele envia um quadro de volta toda vez que recebe um quadro, mesmo que já o tenha enviado. No próximo protocolo, resolveremos de modo elegante o problema do tráfego de mão única.

Por ter vários quadros pendentes, é claro que o protocolo 5 necessita de vários timers, um para cada quadro pendente. Cada quadro tem um timeout independente de todos os demais. Contudo, todos esses timers podem ser facilmente simulados por software, usando-se um único timer do hardware para provocar interrupções periódicas. Os timeouts pendentes formam uma lista encadeada, com cada nó da lista contendo a quantidade de pulsos do timer até que ele expire, o quadro sendo sincronizado e um ponteiro para o nó seguinte.

Para ilustrar como os timers poderiam ser implementados, considere o exemplo da Figura 3.20(a). Suponha que o timer pulse uma vez a cada 1 ms. Inicialmente, o tempo real é 10:00:00.000 e há três timeouts pendentes, em 10:00:00.005, 10:00:00.013 e 10:00:00.019. Toda vez que o timer do hardware pulsar, o tempo real será atualizado e o contador de pulsos no início da lista será decrementado. Quando o contador de pulsos for igual a zero, ocorrerá um timeout e o nó será removido da lista, como mostra a Figura 3.20(b). Embora essa organização exija que a lista seja examinada quando *start_timer* ou *stop_timer* são chamadas, ela não requer muito trabalho por pulso. No protocolo 5, essas duas funções receberam um parâmetro, que indica o quadro a ser sincronizado.

Nesse protocolo, tanto o transmissor quanto o receptor mantêm uma janela de números de sequência pendentes e aceitáveis, respectivamente. O tamanho da janela do transmissor é medido a partir de 0 e atinge um número máximo predefinido. Por sua vez, a janela do receptor tem sempre um tamanho fixo e igual ao máximo predeterminado. O receptor tem um buffer reservado para cada número de sequência dentro de sua janela fixa. Associado a cada buffer há um bit (*arrived*) que informa se o buffer está cheio ou vazio. Sempre que um quadro chega, seu número de sequência é verificado pela função *between*, para confirmar se ele se enquadra na janela. Se isso ocorrer e o quadro ainda não tiver sido recebido, ele será aceito e armazenado. Essa ação é executada sem levar em conta se o quadro contém ou não o próximo pacote esperado pela camada de rede. É obvio que ele deve ser mantido dentro da camada de enlace de dados e não deve ser repassado à camada de rede até que todos os quadros de números mais baixos já tenham sido entregues a ela na ordem correta. Um protocolo que utiliza esse algoritmo é apresentado na Figura 3.21.

A recepção não sequencial introduz outras restrições sobre os números de sequência em comparação a protocolos nos quais os quadros só são aceitos em ordem. Podemos ilustrar melhor o problema com um exemplo. Imagine que haja um número de sequência de 3 bits, de modo que o transmissor tenha permissão para transmitir até sete quadros antes de ser obrigado a esperar por uma confirmação.

Inicialmente, as janelas do transmissor e do receptor são semelhantes às da Figura 3.22(a). No momento, o transmissor envia os quadros de 0 a 6. A janela do receptor permite que ele aceite qualquer quadro com número de sequência entre 0 e 6, ambos inclusive. Todos os sete quadros chegam corretamente; assim, o receptor os confirma e avança a janela para permitir a recepção de 7, 0, 1, 2, 3, 4 ou 5, conforme mostra a Figura 3.22(b). Todos os sete buffers são marcados como vazios.

Nesse ponto ocorre um desastre, na forma de um raio que atinge a central telefônica e apaga todas as confirmações. O protocolo deve operar corretamente apesar disso. Mais tarde, o transmissor entra em timeout e retransmite o quadro 0. Quando esse quadro chega ao receptor, é feita uma conferência para ver se ele se ajusta à janela do receptor. Infelizmente, na Figura 3.22(b), o quadro 0 está dentro da nova janela e, assim, ele será aceito como um novo quadro. O receptor também envia uma confirmação com piggyback para o quadro 6, pois os quadros de 0 a 6 foram recebidos.

O transmissor fica feliz em saber que todos os quadros transmitidos realmente chegaram de forma correta,

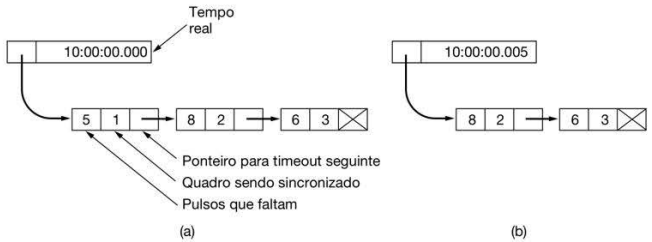

Figura 3.20 Simulação de vários timers por software. (a) Os timeouts enfileirados. (b) A situação após o primeiro timeout expirar.


```
/* O protocolo 6 (retransmissão seletiva) aceita quadros fora de ordem, mas repassa pacotes para a camada
   de rede obedecendo à ordem de transmissão. Há um timer associado a cada quadro pendente. Quando
   o timer expira, apenas o quadro que o contém é retransmitido, e não todos os quadros pendentes, como
   ocorria no protocolo 5. */
#define MAX_SEQ 7                                        /* deve ser 2^n – 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
#include "protocol.h"
boolean no_nak = true;                                   /* nenhuma nak foi enviada */
seq_nr oldest_frame = MAX_SEQ + 1;                       /* valor inicial é apenas para o simulador */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* O mesmo que no protocolo 5, porém mais curto e mais confuso. */
   return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}

static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
/* Constrói e envia um quadro de dados, ack ou nak. */
   frame s;                                              /* variável auxiliar */
   s.kind = fk;                                          /* kind == data, ack, ou nak */
   if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
   s.seq = frame_nr;                                     /* significativo apenas para quadros de dados */
   s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
   if (fk == nak) no_nak = false;                        /* um nak por quadro, por favor */
   to_physical_layer(&s);                                /* transmite o quadro */
   if (fk == data) start_timer(frame_nr % NR_BUFS);
   stop_ack_timer();                                     /* não necessário para quadro ack separado */
}
void protocol6(void)
{
   seq_nr ack_expected;                                  /* limite inferior da janela do transmissor */
   seq_nr next_frame_to_send;                            /* limite superior da janela do transmissor + 1 */
   seq_nr frame_expected;                                /* limite inferior da janela do receptor */
   seq_nr too_far;                                       /* limite superior da janela do receptor + 1 */
   int i;                                                /* índice para os buffers de entrada/saída */
   frame r;                                              /* variável auxiliar */
   packet out_buf[NR_BUFS];                              /* buffers para o fluxo de saída */
   packet in_buf[NR_BUFS];                               /* buffers para o fluxo de entrada */
   boolean arrived[NR_BUFS];                             /* mapa de bits de entrada */
   seq_nr nbuffered;                                     /* quantidade de buffers de saída usados atualmente */
   event_type event;
   enable_network_layer();                               /* inicializa */
   ack_expected = 0;                                     /* próximo ack esperado no fluxo de entrada */
   next_frame_to_send = 0;                               /* número do próximo quadro de saída */
   frame_expected = 0;
   too_far = NR_BUFS;
   nbuffered = 0;                                        /* inicialmente nenhum pacote em buffer */
   for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
   while (true) {
      wait_for_event(&event);                            /* cinco possibilidades: ver event_type acima */
```

Figura 3.21 Um protocolo de janela deslizante que utiliza a retransmissão seletiva. *(Continua)*

```
    switch(event) {
      case network_layer_ready:                                  /* aceita, salva e transmite um novo quadro */
          nbuffered = nbuffered + 1;                             /* estende a janela */
          from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* busca novo pacote */
          send_frame(data, next_frame_to_send, frame_expected, out_buf); /* transmite o quadro */
          inc(next_frame_to_send);                               /* avança o limite da janela superior */
          break;

      case frame_arrival:                                        /* um quadro de dados ou de controle chegou */
          from_physical_layer(&r);                               /* busca quadro que chega da camada física */
          if (r.kind == data) {
              /* Um quadro correto chegou. */
              if ((r.seq != frame_expected) && no_nak)
                 send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
              if (between(frame_expected,r.seq,too_far) && (arrived[r.seq%NR_BUFS]==false)) {
                  /* Quadros podem ser aceitos em qualquer ordem. */
                  arrived[r.seq % NR_BUFS] = true;               /* marca o buffer como cheio */
                  in_buf[r.seq % NR_BUFS] = r.info;              /* insere dados no buffer */
                  while (arrived[frame_expected % NR_BUFS]) {
                      /* Passa quadros e avança a janela. */
                      to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                      no_nak = true;
                      arrived[frame_expected % NR_BUFS] = false;
                      inc(frame_expected);                       /* avança o limite inferior da janela do receptor */
                      inc(too_far);                              /* avança o limite superior da janela do receptor */
                      start_ack_timer();                         /* para ver se precisa de ack separado */
                  }
              }
          }
          if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
              send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
          while (between(ack_expected, r.ack, next_frame_to_send)) {
              nbuffered = nbuffered – 1;                         /* trata do ack piggyback */
              stop_timer(ack_expected % NR_BUFS);                /* quadro chegou intacto */
              inc(ack_expected);                                 /* avança o limite inferior da janela do transmissor */
          }
          break;
      case cksum_err:
          if (no_nak) send_frame(nak, 0, frame_expected, out_buf);  /* quadro danificado */
          break;
      case timeout:
          send_frame(data, oldest_frame, frame_expected, out_buf);  /* timeout */
          break;
      case ack_timeout:
          send_frame(ack,0,frame_expected, out_buf);             /* timer de ack expirou; envia ack */
      }
      if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
    }
}
```

Figura 3.21 *(Continuação)* Um protocolo de janela deslizante que utiliza a retransmissão seletiva.

portanto, ele avança sua janela e envia imediatamente os quadros 7, 0, 1, 2, 3, 4 e 5. O quadro 7 será aceito pelo receptor e seu pacote será repassado diretamente à camada de rede. Logo depois, a camada de enlace de dados receptora verifica se já tem um quadro 0 válido, descobre que sim e repassa o pacote incorporado à camada de rede como se fosse um novo. Consequentemente, a camada de rede recebe um pacote incorreto e o protocolo falha.

A essência do problema é que, depois que o receptor avançou a janela, a nova faixa de números de sequência válidos substituiu a antiga. Consequentemente, o próximo lote de quadros poderia ser formado por cópias (se todas as confirmações se perderam) ou por novos quadros (se todas as confirmações foram recebidas). O receptor não tem como distinguir esses dois casos.

A saída desse dilema reside em ter certeza de que, depois que o receptor avançou sua janela, não há sobreposição entre essa janela e a original. Para assegurar que não há sobreposição, o tamanho máximo da janela deve ser igual à metade do intervalo dos números de sequência, como ocorre na Figura 3.22(c) e (d). Com 3 bits, os números de sequência vão variar de 0 a 7. Apenas quatro quadros não confirmados devem estar pendentes em qualquer instante. Dessa forma, se o receptor só tiver aceito os quadros de 0 a 3 e avançado sua janela para aceitar os quadros de 4 a 7, ele poderá saber, sem nenhuma dúvida, se os quadros subsequentes são retransmissões (0 a 3) ou novos quadros (4 a 7). Em geral, o tamanho da janela para o protocolo 6 será $(MAX\_SEQ + 1)/2$.

Uma pergunta interessante é: quantos buffers o receptor deverá ter? De maneira alguma ele aceitará quadros cujos números de sequência estejam abaixo do limite inferior da janela ou acima do limite superior. Consequentemente, o número de buffers necessário é igual ao tamanho da janela, e não ao intervalo dos números de sequência. No exemplo anterior de um número de sequência de 3 bits, são necessários quatro buffers, numerados de 0 a 3. Quando o quadro i chega, ele é colocado no buffer i mod 4. Observe que, apesar de i e $(i + 4)$ mod 4 estarem "competindo" pelo mesmo buffer, eles nunca estão dentro da janela ao mesmo tempo, pois isso implicaria um tamanho de janela de no mínimo 5.

Pela mesma razão, o número de timers necessários é igual ao número de buffers, e não ao tamanho do espaço de sequência. Efetivamente, um timer é associado a cada buffer. Quando o timer chega a seu timeout, o conteúdo do buffer é retransmitido.

O protocolo 6 também alivia a suposição implícita de que o canal está muito carregado. Fizemos essa suposição no protocolo 5, quando contamos com quadros enviados na direção inversa das confirmações com piggyback. Se o tráfego inverso for leve, a confirmação será retida por um longo período, o que poderá causar problemas. No limite, se houver um tráfego intenso em um sentido e nenhum tráfego no outro, então o protocolo será bloqueado quando a janela transmissora atingir seu máximo.

Para aliviar essa suposição, após um quadro de dados sequencial ser recebido, um timer auxiliar é iniciado por *start_ack_timer*. Se nenhum tráfego inverso tiver se apresentado antes desse timer expirar, é enviado um quadro de confirmação separado. Uma interrupção provocada pelo timer auxiliar é chamada evento *ack_timeout*. Diante dessa organização, o fluxo de tráfego unidirecional passa a ser possível nesse momento, pois a falta de quadros de dados inversos, nos quais as confirmações possam ser transportadas, não representa mais um obstáculo. Existe apenas um timer auxiliar e, se *start_ack_timer* for chamada durante o intervalo em que o timer estiver funcionando, ela não terá efeito. O timer não é reiniciado ou estendido, pois sua finalidade é oferecer uma taxa mínima de confirmações.

É essencial que o timeout associado ao timer auxiliar seja ligeiramente mais curto que o timer utilizado para sincronizar quadros de dados. Essa condição é necessária para assegurar que a confirmação de um quadro corretamente recebido chegue antes de o timer de retransmissão do quadro expirar, de modo que o transmissor não tenha de retransmiti-lo.

O protocolo 6 utiliza uma estratégia mais eficiente que o protocolo 5 para tratamento de erros. Sempre que há motivos para suspeitar da ocorrência de um erro, o receptor envia um quadro de confirmação negativa (NAK) de volta ao transmissor. Ele é um pedido de retransmissão do quadro especificado na NAK. Existem dois casos que podem

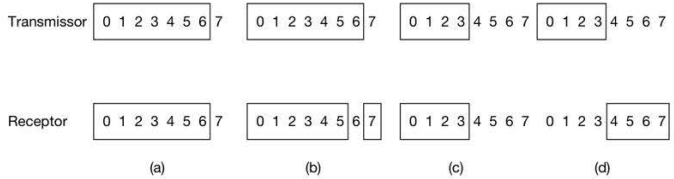

Figura 3.22 (a) Situação inicial com uma janela de tamanho 7. (b) Após sete quadros serem transmitidos e recebidos, mas não confirmados. (c) Situação inicial com uma janela de tamanho 4. (d) Após quatro quadros serem transmitidos e recebidos, mas não confirmados.

provocar a suspeita do receptor: a chegada de um quadro danificado ou de um quadro diferente do esperado (quadro potencialmente perdido). Para impedir que sejam feitas várias solicitações de retransmissão do mesmo quadro perdido, o receptor deve controlar se já foi enviada uma NAK correspondente a determinado quadro. A variável *no_nak* no protocolo 6 será verdadeira (*true*) se nenhuma NAK tiver sido enviada para *frame_expected*. Se a NAK for danificada ou perdida, não haverá qualquer prejuízo real, pois, com o término do intervalo de timeout, de qualquer forma, o transmissor retransmitirá o quadro ausente. Se o quadro errado chegar depois que uma NAK tiver sido enviada e perdida, *no_nak* será verdadeira e o timer auxiliar será inicializado. Quando o timer expirar, uma ACK será enviada para ressincronizar o transmissor com o status atual do receptor.

Em algumas situações, o tempo necessário para que um quadro se propague até o destino, seja processado lá e tenha a confirmação retornada é (praticamente) constante. Nessas situações, o transmissor pode ajustar seu timer para ser "estreito", apenas ligeiramente maior que o intervalo normal esperado entre o envio de um quadro e a recepção de sua confirmação. As NAKs não são úteis nesse caso.

Todavia, em outras situações o tempo deve ser bastante variável. Por exemplo, se o tráfego inverso for esporádico, o tempo antes da confirmação será mais curto quando houver tráfego inverso, e mais longo quando não houver. O transmissor enfrenta a escolha de definir o intervalo para um valor pequeno (arriscando retransmissões desnecessárias) ou defini-lo para um valor grande (e ficar ocioso por longo período após um erro). As duas opções desperdiçam largura de banda. Em geral, sempre que o desvio padrão do intervalo de confirmação é grande em comparação com o próprio intervalo, o timer pode ser ajustado "mais livremente" para ser conservador. As NAKs podem, então, acelerar bastante a retransmissão de quadros perdidos ou danificados.

Um problema intimamente relacionado com o uso de timeouts e NAKs é determinar o quadro que provocou um timeout. No protocolo 5, ele é sempre *ack_expected*, porque é sempre o mais antigo. No protocolo 6, não há uma forma trivial para determinar o quadro que chegou em timeout. Imagine que os quadros de 0 a 4 tenham sido transmitidos, significando que a lista de quadros pendentes é 01234, na ordem do mais antigo para o mais recente. Agora, imagine que o quadro 0 chegue após o timeout, que 5 (um novo quadro) seja transmitido, 1 e 2 cheguem após o timeout e 6 (outro quadro novo) seja transmitido. Nesse ponto, a lista de quadros pendentes será 3405126, do mais antigo para o mais recente. Se todo o tráfego de chegada (i.e., quadros que transportam confirmações) for perdido durante algum tempo, esses sete quadros pendentes chegarão em timeout nessa ordem.

Para evitar que o exemplo fique ainda mais complicado do que já está, não mostramos a administração do timer. Em vez disso, consideramos apenas que a variável *oldest_frame* é ativada no momento de timeout para indicar o quadro que chegou após o timeout.

3.5 EXEMPLOS DE PROTOCOLOS DE ENLACE DE DADOS

Dentro de um prédio, as LANs são muito utilizadas para interconexão, mas a maior parte da infraestrutura de rede a longa distância é montada a partir de linhas ponto a ponto. No Capítulo 4, examinaremos melhor as LANs. Aqui, em três situações comuns, veremos os protocolos de enlace de dados nas linhas ponto a ponto da Internet. A primeira situação é quando os pacotes são enviados por enlaces de fibra óptica SONET, nas redes a longas distâncias. Esses enlaces são muito utilizados, por exemplo, para conectar roteadores nos diferentes locais da rede de um ISP. A segunda situação é a dos enlaces ADSL que se localizam no circuito terminal da rede telefônica, na borda da Internet. A terceira situação é para enlaces DOCSIS no circuito terminal de uma rede a cabo. Tanto ADSL quanto DOCSIS conectam milhões de indivíduos e empresas à Internet.

A Internet precisa de enlaces ponto a ponto para esses fins, bem como modems de linha discada, linhas alugadas, modems a cabo, e assim por diante. Um protocolo padrão, chamado **protocolo ponto a ponto**, ou **PPP** (**Point-to-Point Protocol**), é usado para enviar pacotes por esses enlaces. O PPP é definido na RFC 1661 e mais elaborado na RFC 1662 e em outras RFCs (Simpson, 1994a, 1994b). Enlaces SONET, ADSL e DOCSIS aplicam PPP, mas de maneiras diferentes.

3.5.1 Pacotes sobre SONET

O SONET, que analisamos na Seção 2.5.3, é o protocolo da camada física mais utilizado pelos enlaces de fibra óptica que compõem o backbone das redes de comunicação, incluindo o sistema telefônico. Ele oferece um fluxo de bits que trabalha em uma taxa bem definida, por exemplo, 2,4 Gbps para um enlace OC-48. Esse fluxo de bits é organizado como cargas úteis de bytes, com tamanho fixo, que se repetem a cada 125 μs, havendo ou não dados do usuário para transmitir.

Para transportar pacotes por esses enlaces, é preciso que haja algum mecanismo de enquadramento para distinguir pacotes ocasionais do fluxo de bits contínuo no qual são transportados. O PPP atua sobre roteadores IP para fornecer esse mecanismo, como mostra a Figura 3.23.

O PPP é uma melhoria de um protocolo mais antigo e mais simples, chamado **SLIP** (**Serial Line Internet Protocol**), e é usado para lidar com a configuração do enlace de detecção de erros, dar suporte a múltiplos protocolos,

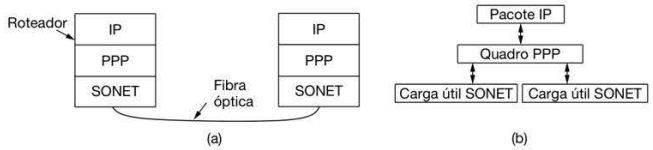

Figura 3.23 Pacotes sobre SONET. (a) Uma pilha de protocolos. (b) Relacionamentos de quadros.

permitir autenticação, entre outros. Com um grande conjunto de opções, o PPP dispõe de três recursos principais:

1. Um método de enquadramento que delineia de forma não ambígua o final de um quadro e o início do seguinte. O formato do quadro também lida com a detecção de erros.
2. Um protocolo de controle de enlace usado para ativar linhas, testá-las, negociar opções e desativá-las novamente de forma controlada quando não forem mais necessárias. Esse protocolo é denominado protocolo de controle de enlace, ou **LCP** (**Link Control Protocol**).
3. Uma maneira de negociar as opções da camada de rede de modo independente do protocolo dessa camada a ser utilizado. O método escolhido deve ter um protocolo de controle de rede, ou **NCP** (**Network Control Protocol**), diferente para cada camada de rede aceita.

O formato de quadro PPP foi definido de modo a ter uma aparência semelhante ao formato do quadro **HDLC** (**High-level Data Link Control**), uma instância bastante utilizada de uma família de protocolos mais antiga, pois não há motivo algum para a definição de um novo padrão.

A principal diferença entre o PPP e o HDLC é que o primeiro é orientado a caracteres, e não a bits. Especificamente, o PPP utiliza a técnica de inserção de bytes e todos os quadros representam um número inteiro de bytes. Já o HDLC usa a inserção de bits, e permite quadros de, digamos, 30,25 bytes.

Contudo, na prática, há uma segunda diferença importante. O HDLC oferece transmissão confiável com uma janela deslizante, confirmações e timeouts da forma como estudamos. O PPP também pode oferecer transmissão confiável em ambientes com ruído, como redes sem fio; os detalhes exatos são definidos na RFC 1663. Entretanto, isso raramente é feito na prática. Em vez disso, um padrão adotado "ainda sem número" quase sempre é usado na Internet para oferecer serviço não orientado a conexão sem confirmação.

A Figura 3.24 mostra o formato do quadro PPP. Todos os quadros PPP começam com o byte de flag padrão do HDLC, 0x7E (01111110). O byte de flag é inserido se ocorrer dentro do campo de *Carga útil* usando o byte de escape 0x7D. O byte seguinte é o XOR entre o byte de escape e 0x20, que inverte o quinto bit. Por exemplo, 0x7D 0x5E é a sequência de escape para o byte de flag 0x7E. Isso significa que o início e o final dos quadros podem ser pesquisados simplesmente procurando-se pelo byte 0x7E, pois ele não ocorrerá em nenhum outro lugar. A regra de retirada na recepção de um quadro é procurar por 0x7D, removê-lo e realizar o XOR do byte seguinte com 0x20. Além disso, apenas um byte de flag é necessário entre os quadros. Vários bytes de flag podem ser usados para preencher o enlace quando não existem quadros para transmitir.

Após o byte de flag de início do quadro, temos o campo *Endereço*, que sempre é definido como o valor binário 11111111, indicando que todas as estações devem aceitar o quadro. A utilização desse valor evita o problema da necessidade de atribuição de endereços de enlace de dados.

O campo *Controle* vem depois do campo *Endereço* e seu valor padrão é 00000011. Esse valor indica um quadro não numerado.

Como os campos *Endereço* e *Controle* sempre são constantes na configuração padrão, o LCP oferece o mecanismo necessário para as duas partes negociarem uma opção que os omita completamente e economize 2 bytes por quadro.

O quarto campo do quadro PPP é o *Protocolo*. Sua tarefa é informar o tipo de pacote que se encontra no campo *Carga útil*. Os códigos que começam com o bit 0 são definidos para o IP versão 4, IP versão 6 e outros protocolos da camada de rede que poderiam ser usados, como IPX e AppleTalk. Os códigos que começam com o bit 1 são utilizados para protocolos de configuração do PPP, incluindo

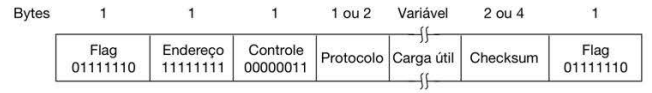

Figura 3.24 O formato completo do quadro PPP para a operação no modo não numerado.

LCP e um NCP diferente para cada protocolo da camada de rede admitido. O tamanho padrão do campo *Protocolo* é 2 bytes, mas é possível negociar uma redução para 1 byte, utilizando-se o LCP. Os projetistas talvez tenham sido demasiadamente cuidadosos, achando que algum dia poderia haver mais de 256 protocolos em uso.

O campo *Carga útil* tem comprimento variável, podendo se estender até o tamanho máximo negociado. Se o comprimento não for negociado com o uso do LCP durante a configuração da linha, será empregado um comprimento padrão de 1.500 bytes. Poderá haver um preenchimento logo após a carga útil, caso seja necessário.

Depois do campo *Carga útil*, temos o campo *Checksum*, que normalmente tem 2 bytes, embora seja possível negociar um checksum de 4 bytes, o qual, na verdade, é o mesmo CRC de 32 bits cujo polinômio gerador é mostrado no final da Seção 3.2.2. O checksum de 2 bytes também é um CRC padrão.

O PPP é um mecanismo de enquadramento que pode transportar os pacotes de vários protocolos por muitos tipos de camadas físicas. Para usá-lo sobre SONET, as escolhas a fazer estão descritas na RFC 2615 (Malis e Simpson, 1999). É utilizado um checksum de 4 bytes, pois esse é o meio principal de detectar erros de transmissão pelas camadas física, de enlace e de rede. Recomenda-se que os campos *Endereço*, *Controle* e *Protocolo* não sejam compactados, pois os enlaces SONET já trabalham com velocidades relativamente altas.

Também há um recurso incomum. A carga útil PPP é misturada (conforme descrevemos na Seção 2.4.3) antes de ser inserida na carga útil SONET. A mistura realiza a operação XOR da carga útil com uma longa sequência pseudo-aleatória antes de ser transmitida. O problema é que o fluxo de bits SONET precisa de transições de bits frequentes para realizar a sincronização. Essas transições vêm naturalmente com a variação em sinais de voz, mas na comunicação de dados o usuário escolhe a informação enviada e poderia enviar um pacote com uma longa sequência de 0s. Com a mistura, a probabilidade de um usuário causar problemas enviando uma longa sequência de 0s se torna extremamente baixa.

Antes que os quadros PPP possam ser transportados por linhas SONET, o enlace PPP precisa ser estabelecido e configurado. As fases pelas quais o enlace passa quando é criado, usado e removido novamente aparecem na Figura 3.25.

O enlace começa com a linha no estado *DEAD*, o que significa que não há nenhuma conexão na camada física. Depois de estabelecida a conexão na camada física, o enlace passa para a fase *ESTABLISH*. Nesse ponto, as partes do PPP trocam uma série de pacotes LCP, cada um transportado no campo *Carga útil* de um quadro PPP, para selecionar as opções do PPP para o enlace a partir das possibilidades mencionadas anteriormente. A parte que inicia propõe opções e a parte que responde as aceita ou rejeita, total ou

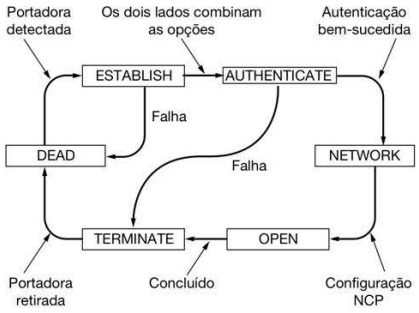

Figura 3.25 Diagrama de estado para ativar e desativar um enlace PPP.

parcialmente. A parte que responde também pode fazer propostas alternativas.

Se a negociação de opções do LCP for bem-sucedida, o enlace chega à fase *AUTHENTICATE*. Agora, as duas partes poderão verificar suas identidades mutuamente, se desejarem. Se a autenticação tiver sucesso, o estado *NETWORK* é alcançado e diversos pacotes NCP são enviados para configurar a camada de rede. É difícil generalizar a respeito dos protocolos NCP, pois cada um é específico a algum protocolo da camada de rede e permite que sejam feitas solicitações de configuração específicas a esse protocolo. Para o IP, por exemplo, a atribuição de endereços IP às duas extremidades do enlace é a possibilidade mais importante.

Quando a fase *OPEN* é alcançada, o transporte de dados pode ser feito. É nesse estado que os pacotes IP são transportados em quadros PPP pela linha SONET. Quando o transporte de dados é concluído, o enlace entra na fase *TERMINATE* e, de lá, volta ao estado *DEAD* quando a conexão da camada física é desativada.

3.5.2 ADSL (Asymmetric Digital Subscriber Line)

A ADSL conecta milhões de assinantes domésticos à Internet em taxas de Mbits/s pelo mesmo circuito terminal de telefone usado para o serviço telefônico tradicional. Na Seção 2.5.2, descrevemos como um dispositivo chamado modem DSL é colocado em uma residência. Ele envia bits do circuito terminal para um dispositivo chamado DSLAM (DSL Access Multiplexer), na estação local da companhia telefônica. Agora, vamos examinar com mais detalhes como os pacotes são transportados por enlaces ADSL.

A imagem geral para os protocolos e dispositivos usados com ADSL aparece na Figura 3.26. Diferentes

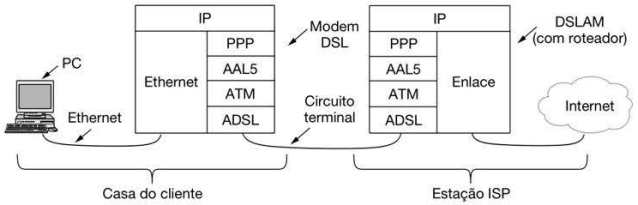

Figura 3.26 Pilhas de protocolos ADSL.

protocolos são empregados em diferentes redes, de modo que escolhemos mostrar o caso mais popular. Dentro da casa, um computador envia pacotes IP ao modem DSL usando uma camada de enlace como o padrão Ethernet. O modem DSL, então, envia os pacotes IP do circuito terminal para o DSLAM usando os protocolos que estamos prestes a estudar. No DSLAM (ou um roteador conectado a ele, dependendo da implementação), os pacotes IP são extraídos e entram em uma rede do ISP, de modo que possam alcançar qualquer destino na Internet.

Os protocolos mostrados pelo enlace ADSL na Figura 3.26 começam de baixo, com a camada física ADSL. Eles são baseados em um esquema de modulação digital, chamado multiplexação por divisão de frequência ortogonal (também conhecido como multitom discreto), como vimos na Seção 2.5.2. Perto do topo da pilha, logo abaixo da camada de rede IP, está o PPP. Esse protocolo é o mesmo PPP que estudamos para transportes de pacote sobre SONET. Ele funciona da mesma maneira para estabelecer e configurar o enlace e transportar pacotes IP.

Entre ADSL e PPP estão ATM e AAL5. Estes são protocolos novos, que ainda não estudamos. O **modo de transferência assíncrono**, ou **ATM** (**Asynchronous Transfer Mode**), foi elaborado no início da década de 1990 e lançado com incrível entusiasmo. Ele prometia uma tecnologia de rede que resolveria os problemas de telecomunicações do mundo, reunindo voz, dados, televisão a cabo, telégrafo, pombo-correio, latinhas conectadas por barbante, tambores e tudo o mais em um sistema integrado, que poderia fazer tudo para todos. Isso não aconteceu. Em grande parte, os problemas do ATM eram semelhantes aos que já descrevemos com relação aos protocolos OSI, ou seja, má sincronização, tecnologia, implementação e política. Apesar disso, o ATM teve muito mais sucesso do que o padrão OSI. Embora não tenha ganho o mundo, ele continua sendo muito usado em nichos, incluindo linhas de acesso de banda larga, como DSL, e enlaces de WAN dentro das redes telefônicas.

O ATM é uma camada de enlace baseada na transmissão de **células** de informação de tamanho fixo. O "assíncrono" em seu nome significa que as células nem sempre precisam ser enviadas da maneira como os bits são continuamente enviados por linhas síncronas, como SONET. As células só precisam ser enviadas quando existe informação para transportar. O ATM é uma tecnologia orientada à conexão. Cada célula transporta um identificador do **circuito virtual** em seu cabeçalho, e os dispositivos usam esse identificador para encaminhar células ao longo dos caminhos das conexões estabelecidas.

Cada célula tem 53 bytes de extensão, consistindo em uma carga útil de 48 bytes mais um cabeçalho de 5 bytes. Usando células pequenas, o ATM pode dividir a largura de banda de um enlace de camada física entre diferentes usuários, de forma flexível, em fatias finas. Esse recurso é útil quando, por exemplo, são enviados voz e dados por um enlace sem ter longos pacotes de dados, que causariam enormes atrasos nas amostras de voz. A escolha incomum para o tamanho da célula (p. ex., em comparação com a escolha mais natural de uma potência de 2) é uma indicação de como o projeto do ATM foi político. O tamanho de 48 bytes para a carga útil foi um meio-termo para resolver um impasse entre a Europa, que queria células de 32 bytes, e os Estados Unidos, que queriam células de 64 bytes. Uma breve visão geral do ATM pode ser encontrada em Siu e Jain (1995).

Para enviar dados por uma rede ATM, eles precisam ser mapeados para uma sequência de células. Esse mapeamento é feito com uma camada de adaptação ATM em um processo chamado segmentação e remontagem. Várias camadas de adaptação foram definidas para diferentes serviços, variando de amostras periódicas de voz até pacotes de dados. A principal usada para os pacotes de dados é a **camada de adaptação ATM 5**, ou **AAL5** (**ATM Adaptation Layer 5**).

Um quadro AAL5 aparece na Figura 3.27. Em vez de um cabeçalho, ele tem um final que oferece o comprimento e um CRC de 4 bytes para detecção de erros. Naturalmente, o CRC é o mesmo usado para redes locais PPP e IEEE 802, como Ethernet. Wang e Crowcroft (1992) mostraram que ele é forte o bastante para detectar erros não tradicionais, como reordenação de célula. Assim como uma carga útil, o quadro AAL5 tem preenchimento. Isso encerra o tamanho geral como um múltiplo de 48 bytes, de modo que o quadro pode ser dividido igualmente em células. Não são necessários endereços no quadro, pois o identificador de

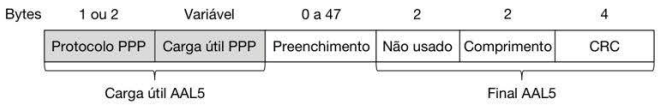

Figura 3.27 Quadro AAL5 transportando dados PPP.

circuito virtual transportado em cada célula o levará ao destino correto.

Agora que já descrevemos o ATM, só precisamos descrever como o PPP o utiliza no caso da ADSL. Isso é feito com outro padrão, chamado **PPPoA** (**PPP over ATM**). Esse padrão não é realmente um protocolo (de modo que não aparece na Figura 3.26), e sim mais uma especificação de como trabalhar com quadros PPP e AAL5. Ele é descrito na RFC 2364 (Gross et al., 1998).

Somente os campos de protocolo e carga útil do PPP são colocados na carga útil do AAL5, como vemos na Figura 3.27. O campo de protocolo indica ao DSLAM no outro extremo se a carga útil é um pacote IP ou um pacote de outro protocolo, como LCP. O canto extremo sabe que as células contêm informações do PPP porque um circuito virtual ATM é estabelecido para essa finalidade.

Dentro do quadro AAL5, o enquadramento PPP não é necessário, pois não teria nenhuma finalidade; ATM e AAL5 já oferecem o enquadramento, e mais seria inútil. O CRC do PPP também não é necessário, pois o AAL5 já inclui o mesmo CRC. Esse mecanismo de detecção de erros suplementa a codificação da camada física da ADSL de um código de Reed-Solomon para correção de erros e um CRC de 1 byte para a detecção de quaisquer erros restantes, não descobertos de outra maneira. Esse esquema tem um mecanismo de recuperação de erros muito mais sofisticado do que quando os pacotes são enviados por uma linha SONET, pois a ADSL é um canal com muito mais ruído.

3.5.3 DOCSIS (Data Over Cable Service Interface Specification)

O protocolo **DOCSIS** (**Data Over Cable Service Interface Specification**) geralmente é descrito como tendo dois componentes: a camada física (PHY), conforme descrito no capítulo anterior (às vezes chamada de PMD ou subcamada dependente do meio), e a camada de controle de acesso ao meio (MAC), que abordaremos com mais detalhes no Capítulo 4. Acima da camada física, o DOCSIS deve lidar com uma série de tarefas para a camada de rede, incluindo a alocação de largura de banda nas direções upstream e downstream (controle de fluxo), enquadramento e correção de erros (é claro que, às vezes, a correção de erros é vista como uma construção da camada física). Descrevemos cada um desses conceitos anteriormente neste capítulo. Nesta seção, exploraremos como o DOCSIS trata de cada um desses problemas.

Um quadro DOCSIS contém várias informações, incluindo indicadores de qualidade de serviço e suporte para fragmentação ou concatenação de quadros. Cada sequência unidirecional de quadros é chamada de **fluxo de serviço**. Os fluxos de serviço principais permitem que o CMTS (sistema de terminação de modem a cabo, na central da empresa de cabo) comunique mensagens de gerenciamento a cada modem a cabo. Cada fluxo de serviço tem um identificador único e normalmente está associado a uma classe de serviço, que pode ser o melhor esforço, pesquisa (em que um modem a cabo faz solicitações explícitas de largura de banda) e serviço de concessão (em que um modem a cabo transmite rajadas de dados a um taxa de dados garantida). Um fluxo de serviço primário é o fluxo de serviço padrão que transporta todos os quadros que não são classificados para outro serviço. Em muitos arranjos de serviço de banda larga, há apenas um fluxo de serviço upstream e downstream padrão entre o CM e o CMTS, que transporta todo o tráfego do usuário, bem como todas as mensagens de gerenciamento. As redes DOCSIS foram historicamente projetadas assumindo que a maioria dos dados é transmitida no sentido downstream. Certos aplicativos, como videoconferência, vão contra essas tendências, embora serviços de jogos em nuvem anunciados recentemente (p. ex., Stadia, GeForce Now, xCloud) possam resultar em uma utilização ainda mais downstream, já que essas aplicações têm como objetivo taxas de streaming contínuo de 30 a 35 Mbps.

Depois que um modem a cabo é ligado, ele estabelece uma conexão com o CMTS, o que normalmente permite que ele se conecte ao resto da rede. Ao se registrar no CMTS, ele adquire canais de comunicação upstream e downstream para usar, bem como chaves de criptografia do CMTS. As operadoras upstream e downstream oferecem dois canais compartilhados para todos os modems a cabo. Na direção downstream, todos os modems a cabo conectados ao CMTS recebem todos os pacotes transmitidos. Na direção upstream, muitos modems a cabo transmitem e o CMTS é o único receptor. Pode haver vários caminhos físicos entre o CMTS e cada modem a cabo.

Antes do DOCSIS 3.1, os pacotes no sentido downstream eram divididos em quadros MPEG de 188 bytes, cada um com um cabeçalho de 4 bytes e uma carga útil de 184 bytes (a chamada camada de convergência de transmissão MPEG). Além dos dados em si, o CMTS envia periodicamente informações de gerenciamento para o modem a cabo, incluindo informações sobre alcance, atribuição de

canal e outras tarefas relacionadas à alocação de canal que são realizadas pela camada MAC (que abordaremos com mais detalhes no Capítulo 4). Embora o DOCSIS 3.1 ainda tenha suporte para essa camada de convergência para propósitos legados, ele não depende mais dela para a comunicação downstream.

A camada de enlace DOCSIS organiza a transmissão de acordo com **perfis de modulação**. Um perfil de modulação é uma lista de ordens de modulação (i.e., carregamentos de bits) que correspondem às subportadoras OFDM. No sentido downstream, o CMTS pode usar perfis diferentes para modems a cabo diferentes, mas normalmente um grupo de modems a cabo que têm o mesmo desempenho ou desempenho semelhante será agrupado no mesmo perfil. Com base na identificação do fluxo de serviço e nos parâmetros de QoS, a camada de enlace (no DOCSIS 3.1), agora chamada **camada de convergência**, agrupa pacotes que têm o mesmo perfil no mesmo buffer de envio; normalmente, há um buffer de envio por perfil, cada um dos quais é superficial, para evitar latência significativa. O construtor de palavras-código então mapeia cada quadro DOCSIS para as palavras-código FEC correspondentes, puxando pacotes dos buffers apenas perfis diferentes a cada limite de palavra-código. A codificação FEC vê o quadro DOCSIS como um fluxo de bits, não como uma sequência de bytes. DOCSIS depende de uma palavra-código LDPC. No sentido downstream, uma palavra-código completa tem até 2027 bytes, dos quais até 1799 bytes são dados e 225 são paridade. Dentro de cada byte de um quadro DOCSIS, o bit menos significativo é transferido primeiro; quando um valor superior a um byte é transmitido, os bytes são ordenados do mais significativo para o menos significativo, uma sequência às vezes chamada de **ordem de rede**. O CMTS também adota o preenchimento de bytes: se não houver um quadro DOCSIS no sentido downstream, o CMTS insere subportadoras preenchidas com bits zero nos símbolos OFDM ou simplesmente armazena sequências de 1s em palavras-código, como mostra a Figura 3.28.

Desde a versão 3.0, o DOCSIS oferece suporte a uma tecnologia chamada **ligação de canal**, que permite a um único assinante usar vários canais upstream e downstream simultaneamente. Essa tecnologia é uma forma de **agregação de enlace**, que pode combinar vários enlaces físicos, ou portas, para criar uma única conexão lógica. O DOCSIS 3.0 permite que até 32 canais downstream e 8 canais upstream sejam ligados, onde cada canal pode ter de 6 a 8 MHz de largura. A ligação de canal no DOCSIS 3.1 é a mesma do DOCSIS 3.0, embora o DOCSIS 3.1 suporte canais mais largos para upstream e downstream: a diferença é que esses canais podem ser muito mais largos (até 192 MHz em downstream, 96 MHz em upstream, em comparação com 6 ou 8 MHz downstream e até 6,4 MHz upstream, no caso do DOCSIS 3.0). Por outro lado, um modem DOCSIS 3.1 pode ligar canais de vários tipos (p. ex., um modem DOCSIS 3.1 pode ligar um canal OFDM de 192 MHz e quatro canais SC-QAM de 6 MHZ).

3.6 RESUMO

A tarefa da camada de enlace de dados é converter o fluxo de dados brutos fornecido pela camada física em um fluxo de quadros a ser utilizado pela camada de rede. A camada de enlace pode apresentar esse fluxo com níveis de confiabilidade variados, desde o serviço não orientado à conexão sem confirmação até o serviço confiável, orientado à conexão.

Diversos métodos de enquadramento são utilizados, inclusive a contagem e a inserção de bytes e de bits. Os protocolos de enlace de dados podem oferecer recursos de controle de erros para a detecção ou correção de quadros danificados e a retransmissão de quadros perdidos. Para evitar que um transmissor rápido sobrecarregue um receptor lento, o protocolo de enlace de dados também pode fornecer controle de fluxo. O mecanismo de janela deslizante é bastante utilizado para integrar os controles de erros e de fluxo de maneira simples. Quando o tamanho da janela é de 1 pacote, o protocolo é stop-and-wait.

Os códigos para correção e detecção de erros acrescentam informações redundantes às mensagens usando diversas técnicas matemáticas. Os códigos de convolução e de Reed-Solomon são muito utilizados para correção de

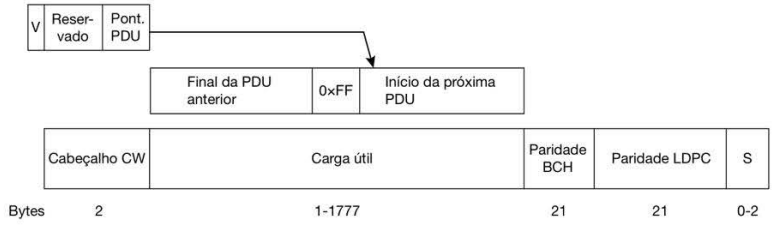

Figura 3.28 Quadro DOCSIS para mapeamento da palavra-código.

erros, mas os códigos de verificação de paridade de baixa densidade estão ganhando popularidade. Na prática, os códigos utilizados para detecção de erros incluem verificações de redundância cíclica e checksums. Todos podem ser aplicados na camada de enlace, bem como na camada física e em camadas mais altas.

Examinamos diversos protocolos que oferecem uma camada de enlace confiável usando confirmações e retransmissões, ou ARQ, sob suposições mais realistas. Começando a partir de um ambiente livre de erros, em que o receptor pode tratar de qualquer quadro que lhe é enviado, apresentamos o controle de fluxo, seguido pelo controle de erros com números de sequência e o algoritmo stop-and-wait. Depois, usamos o algoritmo de janela deslizante para permitir a comunicação bidirecional e apresentamos o conceito de piggybacking. Os dois últimos protocolos realizam o pipeline da transmissão de vários quadros para impedir que o transmissor bloqueie um enlace com um longo atraso de propagação. O receptor pode descartar todos os quadros menos o próximo na sequência, ou então armazenar em buffer os quadros fora de ordem e enviar confirmações negativas, aumentando a eficiência da largura de banda. A primeira estratégia é um protocolo go-back-n, e a segunda é um protocolo de repetição seletiva.

A Internet utiliza o PPP como principal protocolo de enlace de dados em linhas ponto a ponto. Ele oferece um serviço não orientado a conexões sem confirmação, usando bytes de flag para delimitar quadros e um CRC para detectar erros. É usado para transportar pacotes por uma série de enlaces, incluindo enlaces SONET em redes a longas distâncias e enlaces ADSL para residências. DOCSIS é usado quando o serviço de Internet é fornecido pela rede de TV a cabo existente.

PROBLEMAS

1. A rede Ethernet utiliza um preâmbulo em combinação com um contador de bytes para separar os quadros. O que acontecerá se um usuário tentar enviar dados que contêm esse preâmbulo?
2. O fragmento de dados a seguir ocorre no meio de um fluxo de dados para o qual é usado o algoritmo de inserção de bytes descrito no texto: A B ESC C ESC FLAG FLAG D. Qual será a saída após a inserção?
3. Qual é o overhead máximo no algoritmo de inserção de bytes?
4. Você recebe o seguinte fragmento de dados: 0110 0111 1100 1111 0111 1101 e sabe que o protocolo utiliza a inserção de bits. Mostre os dados após a retirada dos bits.
5. Quando a inserção de bits é utilizada, é possível que a perda, inserção ou modificação de um único bit cause um erro não detectado pelo checksum? Se não, por quê? Se sim, como isso pode acontecer? O comprimento do checksum influencia em alguma coisa aqui?
6. Um pacote de uma camada superior está dividido em 10 quadros, cada um deles com 80% de chance de chegar sem danos. Se o protocolo de enlace de dados não fizer nenhum controle de erros, quantas vezes em média a mensagem deverá ser enviada para que o processo inteiro seja concluído?
7. Você consegue imaginar alguma circunstância em que seria preferível um protocolo de loop aberto (p. ex., um código de Hamming) aos protocolos de feedback discutidos neste capítulo?
8. Para proporcionar maior confiabilidade que a obtida com um único bit de paridade, um esquema de codificação para detecção de erros utiliza um bit de paridade para verificar todos os bits de numeração ímpar e um segundo para todos os bits de numeração par. Qual é a distância de Hamming desse código?
9. Depois de notar que o serviço de mensagens instantâneas que você usa todos os dias não oferece detecção de erros, você decide adotar, por conta própria, um mecanismo de detecção de erros simples: envia todas as suas mensagens duas vezes. Qual é a distância de Hamming correspondente e a taxa de código? Qual é a semelhança entre isso e a inclusão de um bit de paridade?
10. Imagine um mecanismo de detecção de erros que envia cada mensagem duas vezes. Supondo que ocorram exatamente dois erros de único bit, qual é a probabilidade de os erros passarem sem ser detectados? Qual é a probabilidade de quando se usa um bit de paridade? Qual método detecta mais erros?
11. Um byte de 8 bits com valor binário 10101111 deve ser codificado usando um código de Hamming com paridade par. Qual é o valor binário após a codificação?
12. Um byte de 8 bits com valor binário 10011010 deve ser codificado usando um código de Hamming com paridade *ímpar*. Qual é o valor binário após a codificação?
13. Um código de Hamming de 12 bits com paridade *ímpar*, cujo valor hexadecimal é 0xB4D, chega a um receptor. Qual era o valor original em hexadecimal? Suponha que não exista mais de 1 bit com erro.
14. Os códigos de Hamming têm uma distância de três e podem ser usados para corrigir um único erro ou detectar um erro duplo. Eles podem ser usados para realizar ambos ao mesmo tempo? Explique sua resposta. Em geral, se a distância de Hamming for n, quantos erros podem ser corrigidos? Quantos erros podem ser detectados?
15. Considere um protocolo que, para cada 16 bytes de dados de mensagem, seja acrescentado 1 byte de dados redundantes. Esse protocolo pode utilizar um código de Hamming para corrigir erros isolados?
16. Uma forma de detectar erros é transmitir dados como um bloco de n linhas com k bits por linha e acrescentar bits de paridade a cada linha e a cada coluna. O canto inferior direito é um bit de paridade que verifica sua linha e sua coluna. Esse esquema detectará todos os erros simples (isolados)? E os erros duplos? E os erros triplos? Mostre que esse esquema não pode detectar alguns erros de quatro bits.

17. No problema anterior, quantos erros podem ser detectados e corrigidos?
18. Indique uma fórmula para o limite inferior do número de bits redundantes r que precisam ser acrescentados a uma mensagem m, a fim de corrigir todos os erros simples *e duplos*.
19. Dada a resposta da questão anterior, explique a popularidade dos complexos mecanismos probabilísticos de correção de erros, como os códigos de convolução e a verificação de paridade de baixa densidade, discutidos neste capítulo.
20. Suponha que sejam transmitidos dados em blocos com tamanhos de 1.000 bits. Qual é a taxa máxima de erro sob a qual o mecanismo de detecção de erros e retransmissão (1 bit de paridade por bloco) é melhor do que usar o código de Hamming? Suponha que os erros de bit sejam independentes um do outro e nenhum erro de bit ocorra durante a retransmissão.
21. Um bloco de bits com n linhas e k colunas utiliza bits de paridade horizontais e verticais para a detecção de erros. Imagine que exatamente 4 bits sejam invertidos em virtude de erros de transmissão. Derive uma expressão para a probabilidade de que o erro não seja detectado.
22. Suponha que uma mensagem 1001 1100 1010 0011 seja transmitida usando o checksum da Internet (palavra de 4 bits). Qual é o valor do checksum?
23. Qual é o resto obtido pela divisão de $x^7 + x^5 + 1$ pelo polinômio gerador $x^3 + 1$?
24. Um fluxo de bits 10011101 é transmitido com a utilização do método CRC padrão descrito no texto. O polinômio gerador é $x^3 + 1$. Mostre a sequência de bits real transmitida. Suponha que o terceiro bit a partir da esquerda seja invertido durante a transmissão. Mostre que esse erro é detectado na extremidade receptora. Dê um exemplo de erro de bit, na sequência de bits transmitida, que não será detectado pelo receptor.
25. Um fluxo de bits 11100110 é transmitido com a utilização do método CRC-padrão descrito no texto. O polinômio gerador é $x^4 + x^3 + 1$. Mostre a sequência de bits real transmitida. Suponha que o terceiro bit a partir da esquerda seja invertido durante a transmissão. Mostre que esse erro é detectado na extremidade receptora. Dê um exemplo de erro de bit, na sequência de bits transmitida, que não será detectado pelo receptor.
26. Os protocolos de enlace de dados sempre colocam o CRC em um final (trailer), em vez de um cabeçalho. Por quê?
27. Na discussão do protocolo ARQ na Seção 3.3.3, esboçamos uma situação que resultou no receptor aceitando duas cópias do mesmo quadro, em decorrência de uma perda do quadro de confirmação. É possível que um receptor aceite várias cópias do mesmo quadro quando nenhum dos quadros (mensagem ou confirmação) foi perdido?
28. Um canal tem uma taxa de bits de 4 kbps e um atraso de propagação de 20 ms. Para que faixa de variação de tamanhos de quadros a técnica stop-and-wait proporciona uma eficiência de pelo menos 50%?
29. Dois protocolos, A e B, diferem apenas nos tamanhos de janela de transmissão. O protocolo A usa uma janela de 20 quadros. O protocolo B é um protocolo stop-and-wait. Os dois protocolos são transportados por dois canais idênticos. Se o protocolo A atinge quase 100% de eficiência da largura de banda, qual é a eficiência de largura de banda do protocolo B?
30. Um protocolo stop-and-wait alcança 25% de eficiência da largura de banda usando quadros de 900 bits por meio de um canal com 50 ms de atraso de propagação de mão única. Em bits por segundo, qual é o atraso de propagação desse canal?
31. Um protocolo stop-and-wait alcança 60% de eficiência da largura de banda usando quadros de 300 bits por meio de um canal com 50 kbps de largura de banda. Qual é o atraso de propagação em mão única desse canal?
32. Um protocolo stop-and-wait que usa quadros de 800 bits percorre um canal com um atraso de propagação de mão única de 8 ms e uma largura de banda de 1200 kbps. Qual é a eficiência de largura de banda que esse protocolo atinge nesse canal?
33. Um protocolo de janela deslizante usa quadros de 1.000 bits e um tamanho de janela de transmissão fixo de 3. Ele atinge quase 100% de eficiência da largura de banda em um canal de 250 kbps. O mesmo protocolo é usado em um canal aprimorado, que possui o mesmo atraso mas dobra a largura de banda. Qual é a eficiência da largura de banda do protocolo no novo canal?
34. No protocolo 3, é possível que o transmissor inicialize o timer quando ele já estiver funcionando? Nesse caso, como isso poderia acontecer? Se não, por que não é possível?
35. Um tronco T1 com o comprimento de 3.000 Km é utilizado para transmitir quadros de 64 bytes usando o protocolo 5. Se a velocidade de propagação for de 6 μs/Km, quantos bits deverão ter os números de sequência?
36. Imagine que um protocolo de janela deslizante utilize tantos bits para números de sequência que nunca ocorra sobreposição. Que relações devem ser mantidas entre os quatros limitadores da janela e o tamanho da janela, que é constante e idêntica para o transmissor e o receptor?
37. No protocolo 6, quando um quadro de dados chega, é feita uma verificação para confirmar se o número de sequência é diferente do esperado, e se *no_nak* é verdadeira. Se as duas condições forem verdadeiras, será enviada uma NAK. Caso contrário, o timer auxiliar será iniciado. Imagine que o comando `else` tenha sido omitido. Essa alteração afetaria a exatidão do protocolo?
38. Imagine que o loop `while` de três instruções próximo ao fim do protocolo 6 fosse removido do código. Isso afetaria a exatidão do protocolo ou apenas o desempenho? Explique sua resposta.
39. No problema anterior, considere que um protocolo de janela deslizante seja usado em seu lugar. Para qual tamanho da janela de transmissão a utilização do enlace será de 100%? Você pode ignorar os tempos de processamento do protocolo no transmissor e no receptor.
40. Suponha que o caso dos erros de checksum fosse removido da instrução `switch` do protocolo 6. Como essa mudança afetaria a operação do protocolo?

41. No protocolo 6, o código de *frame_arrival* tem uma seção utilizada para NAKs. Essa seção será chamada se o quadro recebido for uma NAK e outra condição for satisfeita. Crie uma situação em que a presença dessa outra condição seja essencial.

42. Considere a operação do protocolo 6 sobre uma linha perfeita (i.e., livre de erros) de 1 Mbps. O tamanho máximo de quadro é de 1.000 bits. Novos pacotes são gerados a cada segundo. O intervalo de timeout é de 10 ms. Se o timer especial de confirmação fosse eliminado, ocorreriam timeouts desnecessários. Quantas vezes em média a mensagem seria transmitida?

43. No protocolo 6, $MAX\_SEQ = 2^n - 1$. Embora essa condição seja evidentemente desejável para tornar a utilização dos bits de cabeçalho mais eficiente, não demonstramos que ela é essencial. Por exemplo, o protocolo funciona corretamente para $MAX\_SEQ = 4$?

44. Quadros de 1.000 bits são enviados por um canal de 1 Mbps usando um satélite geoestacionário cujo tempo de propagação a partir da Terra é de 270 ms. As confirmações sempre são transportadas por piggyback em quadros de dados. Os cabeçalhos são muito curtos. São utilizados números de sequência de 3 bits. Qual é a utilização máxima do canal que é possível alcançar para:

(a) Stop-and-wait?
(b) Protocolo 5?
(c) Protocolo 6?

45. Confirmações negativas disparam diretamente uma resposta no transmissor, enquanto a falta de confirmações positivas só dispara uma ação após um timeout. Seria possível criar um canal de comunicação confiável usando apenas confirmações negativas e nenhuma confirmação positiva? Se for possível, mostre um exemplo. Se for impossível, explique o motivo.

46. Considere um canal de satélite de 64 kbps livre de erros utilizado para enviar quadros de dados de 512 bytes em um sentido, com confirmações muito curtas voltando no outro sentido. Qual é o throughput máximo para os tamanhos de janelas iguais a 1, 7, 15 e 127? O tempo de propagação entre a Terra e o satélite é de 270 ms.

47. Um cabo com 100 km de comprimento funciona na taxa de dados T1. A velocidade de propagação no cabo é igual a 2/3 da velocidade da luz no vácuo. Quantos bits o cabo pode conter?

48. Cite pelo menos um motivo pelo qual o PPP utiliza a inserção de bytes e não a inserção de bits para evitar que bytes de flag acidentais na carga útil causem confusão.

49. Qual é o overhead mínimo para o envio de um pacote IP usando o PPP? Leve em consideração apenas o overhead introduzido pelo próprio PPP, e não o overhead do cabeçalho IP. Qual é o overhead máximo?

50. O fluxo de dados a seguir é enviado por meio de um quadro PPP sobre SONET: ESC FLAG FLAG ESC. Qual é a sequência de bytes transmitida na carga útil? Escreva sua resposta como uma sequência de bytes, cada um representado por oito uns ou zeros. A sequência de bits usada para representar ESC é 01111101 e a sequência de bits para FLAG é 01111110.

51. Um pacote IP de 100 bytes é transmitido por um circuito terminal usando a pilha de protocolos ADSL. Quantas células ATM serão transmitidas? Descreva seu conteúdo resumidamente.

52. O objetivo deste exercício de laboratório é implementar um mecanismo de detecção de erros usando o algoritmo CRC padrão descrito no texto. Escreva dois programas, um *gerador* e um *verificador*. O programa *gerador* lê na entrada padrão uma mensagem de n bits que tem a forma de uma sequência de valores 0 e 1 como uma linha de texto ASCII. A segunda linha é o polinômio de k bits, também em ASCII. A saída padrão é uma linha de texto ASCII com $n + k$ valores, e 0 e 1 representam a mensagem a ser transmitida. Em seguida, são atribuídos valores de saída ao polinômio, exatamente como ele foi lido na entrada. O programa verificador lê a saída no programa gerador e transmite uma mensagem indicando se ela está correta ou não. Por fim, escreva um programa, chamado *alterar*, que inverta 1 bit na primeira linha, dependendo de seu argumento (o número do bit, considerando o bit mais à esquerda como igual a 1), mas copie o restante das duas linhas de forma correta. Digitando:

```
gerador <arquivo | verificador
```

você deverá ver que a mensagem está correta; porém, digitando

```
gerador <arquivo | alterar arg | verificador
```

você deverá obter a mensagem de erro.