Capítulo 5: Protocolos de ponta a ponta

Problema: Fazer com que os processos se comuniquem

Muitas tecnologias podem ser usadas para conectar um conjunto de computadores, desde simples redes Ethernet e sem fio até interconexões de rede em escala global. Uma vez interconectados, o próximo problema é transformar esse serviço de entrega de pacotes de host para host em um canal de comunicação entre processos. Esse é o papel desempenhado pelo nível de *transporte* da arquitetura de rede, que, por suportar a comunicação entre programas aplicativos executados em nós finais, às vezes é chamado de protocolo *ponta a ponta* .

Duas forças moldam o protocolo de ponta a ponta. Visto acima, os processos em nível de aplicação que utilizam seus serviços têm certos requisitos. A lista a seguir detalha algumas das propriedades comuns que um protocolo de transporte deve fornecer:

- Garante a entrega da mensagem
- Entrega mensagens na mesma ordem em que são enviadas
- Entrega no máximo uma cópia de cada mensagem
- Suporta mensagens arbitrariamente grandes
- Suporta sincronização entre o remetente e o destinatário
- Permite que o receptor aplique controle de fluxo ao remetente
- Suporta múltiplos processos de aplicação em cada host

Observe que esta lista não inclui todas as funcionalidades que os processos de aplicação podem desejar da rede. Por exemplo, ela não inclui recursos de segurança

como autenticação ou criptografia, que normalmente são fornecidos por protocolos acima do nível de transporte. (Discutiremos tópicos relacionados à segurança em um capítulo posterior.)

Visto de baixo, a rede subjacente sobre a qual o protocolo de transporte opera apresenta certas limitações no nível de serviço que pode fornecer. Algumas das limitações mais típicas da rede são:

- Soltar mensagens
- Reordenar mensagens
- Entregar cópias duplicadas de uma determinada mensagem
- Limitar mensagens a um tamanho finito
- Entregar mensagens após um atraso arbitrariamente longo

Diz-se que essa rede fornece um nível de serviço *de melhor esforço* , como exemplificado pela Internet.

O desafio, portanto, é desenvolver algoritmos que transformem as propriedades imperfeitas da rede subjacente no alto nível de serviço exigido pelos programas aplicativos. Diferentes protocolos de transporte empregam diferentes combinações desses algoritmos. Este capítulo analisa esses algoritmos no contexto de quatro serviços representativos: um serviço simples de demultiplexação assíncrona, um serviço confiável de fluxo de bytes, um serviço de solicitação/resposta e um serviço para aplicações em tempo real.

No caso dos serviços de demultiplexação e fluxo de bytes, utilizamos o Protocolo de Datagrama de Usuário (UDP) e o Protocolo de Controle de Transmissão (TCP) da Internet, respectivamente, para ilustrar como esses serviços são fornecidos na prática. No caso de um serviço de solicitação/resposta, discutimos o papel que ele desempenha em um serviço de Chamada de Procedimento Remoto (RPC) e quais

recursos ele oferece. A Internet não possui um único protocolo RPC, portanto, encerramos esta discussão com uma descrição de três protocolos RPC amplamente utilizados: SunRPC, DCE-RPC e gRPC.

Por fim, aplicações em tempo real impõem exigências específicas ao protocolo de transporte, como a necessidade de transportar informações de tempo que permitam a reprodução de amostras de áudio ou vídeo no momento apropriado. Analisaremos os requisitos impostos pelas aplicações a esse protocolo e ao exemplo mais utilizado, o Protocolo de Transporte em Tempo Real (RTP).

5.1 Demultiplexador Simples (UDP)

O protocolo de transporte mais simples possível é aquele que estende o serviço de entrega host-a-host da rede subjacente para um serviço de comunicação processo-a-processo. É provável que haja muitos processos em execução em qualquer host, portanto, o protocolo precisa adicionar um nível de demultiplexação, permitindo que múltiplos processos de aplicação em cada host compartilhem a rede. Além desse requisito, o protocolo de transporte não adiciona nenhuma outra funcionalidade ao serviço de melhor esforço fornecido pela rede subjacente. O Protocolo de Datagrama de Usuário da Internet é um exemplo desse protocolo de transporte.

A única questão interessante em tal protocolo é a forma do endereço usado para identificar o processo de destino. Embora seja possível que os processos se identifiquem *diretamente* com um ID de processo (pid) atribuído pelo sistema operacional, tal abordagem só é prática em um sistema distribuído fechado, no qual um único sistema operacional é executado em todos os hosts e atribui a cada processo um ID exclusivo. Uma abordagem mais comum, e a usada pelo UDP, é que os processos se identifiquem *indiretamente* usando um localizador abstrato, geralmente chamado de

porta . A ideia básica é que um processo de origem envie uma mensagem para uma porta e que o processo de destino receba a mensagem de uma porta.

O cabeçalho de um protocolo ponta a ponta que implementa essa função de demultiplexação normalmente contém um identificador (porta) para o remetente (origem) e o destinatário (destino) da mensagem. Por exemplo, o cabeçalho UDP é apresentado na Figura 125. Observe que o campo de porta UDP tem apenas 16 bits. Isso significa que há até 64 mil portas possíveis, claramente insuficientes para identificar todos os processos em todos os hosts da Internet. Felizmente, as portas não são interpretadas em toda a Internet, mas apenas em um único host. Ou seja, um processo é realmente identificado por uma porta em algum host específico: um par (porta, host). Esse par constitui a chave de demultiplexação para o protocolo UDP.

A próxima questão é como um processo aprende a porta do processo para o qual deseja enviar uma mensagem. Normalmente, um processo cliente inicia uma troca de mensagens com um processo servidor. Uma vez que um cliente tenha contatado um servidor, o servidor sabe a porta do cliente (a partir do `SrcPrt`campo contido no cabeçalho da mensagem) e pode responder a ela. O verdadeiro problema, portanto, é como o cliente aprende a porta do servidor em primeiro lugar. Uma abordagem comum é o servidor aceitar mensagens em uma *porta bem conhecida* . Ou seja, cada servidor recebe suas mensagens em alguma porta fixa amplamente divulgada, muito semelhante ao serviço telefônico de emergência disponível nos Estados Unidos no conhecido número de telefone 911. Na Internet, por exemplo, o Servidor de Nomes de Domínio (DNS) recebe mensagens na conhecida porta 53 em cada host, o serviço de e-mail escuta mensagens na porta 25 e o `talk`programa Unix aceita mensagens na conhecida porta 517, e assim por diante. Esse mapeamento é publicado periodicamente em um RFC e está disponível na maioria dos sistemas Unix no arquivo `/etc/services`. Às vezes, uma porta conhecida é apenas o ponto de partida para a comunicação: o cliente e o servidor usam a porta conhecida para concordar com outra

porta que usarão para comunicação subsequente, deixando a porta conhecida livre para outros clientes.

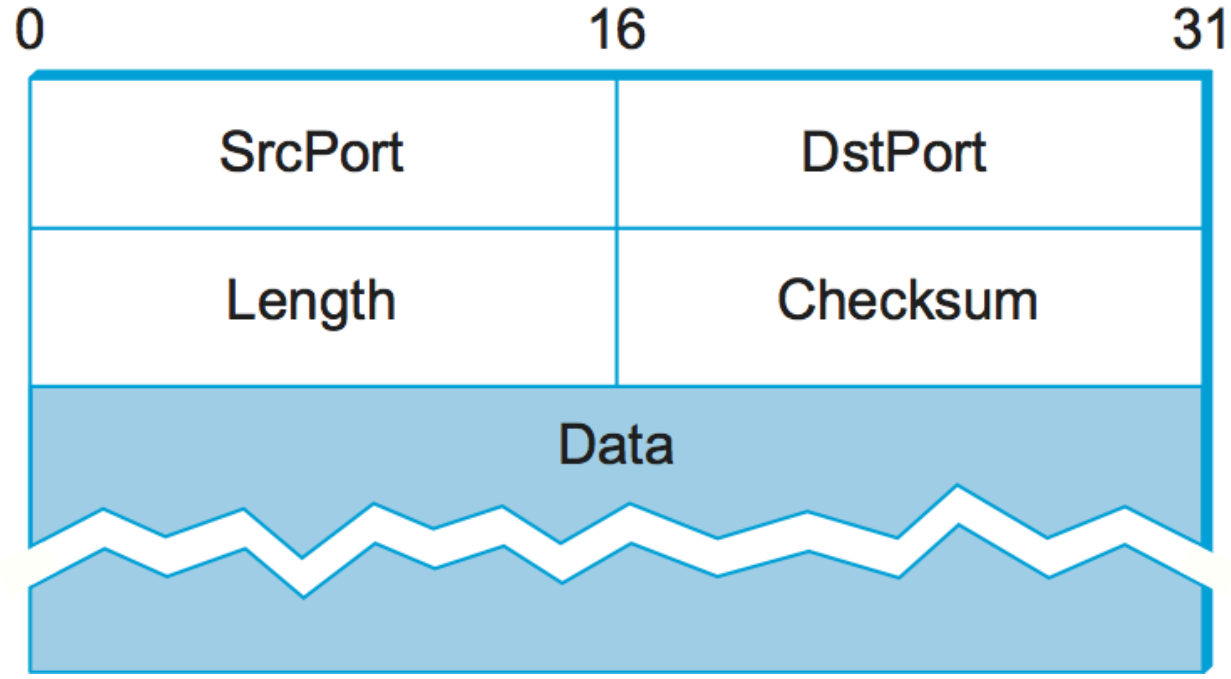

Figura 125. *Formato para cabeçalho UDP.*

Uma estratégia alternativa é generalizar essa ideia, de modo que haja apenas uma porta conhecida — aquela na qual o serviço *de mapeamento de portas* aceita mensagens. Um cliente enviaria uma mensagem para a porta conhecida do mapeador de portas solicitando a porta que deveria usar para se comunicar com o serviço "qualquer que seja", e o mapeador de portas retornaria a porta apropriada. Essa estratégia facilita a alteração da porta associada a diferentes serviços ao longo do tempo e permite que cada host use uma porta diferente para o mesmo serviço.

Como mencionado, uma porta é puramente uma abstração. A forma exata como ela é implementada difere de sistema para sistema, ou mais precisamente, de sistema operacional para sistema operacional. Por exemplo, a API de soquete descrita no

Capítulo 1 é um exemplo de implementação de portas. Normalmente, uma porta é implementada por uma fila de mensagens, conforme ilustrado na Figura 126. Quando uma mensagem chega, o protocolo (por exemplo, UDP) anexa a mensagem ao final da fila. Se a fila estiver cheia, a mensagem é descartada. Não há mecanismo de controle de fluxo no UDP para dizer ao remetente para diminuir a velocidade. Quando um processo de aplicativo deseja receber uma mensagem, uma é removida da frente da fila. Se a fila estiver vazia, o processo bloqueia até que uma mensagem fique disponível.

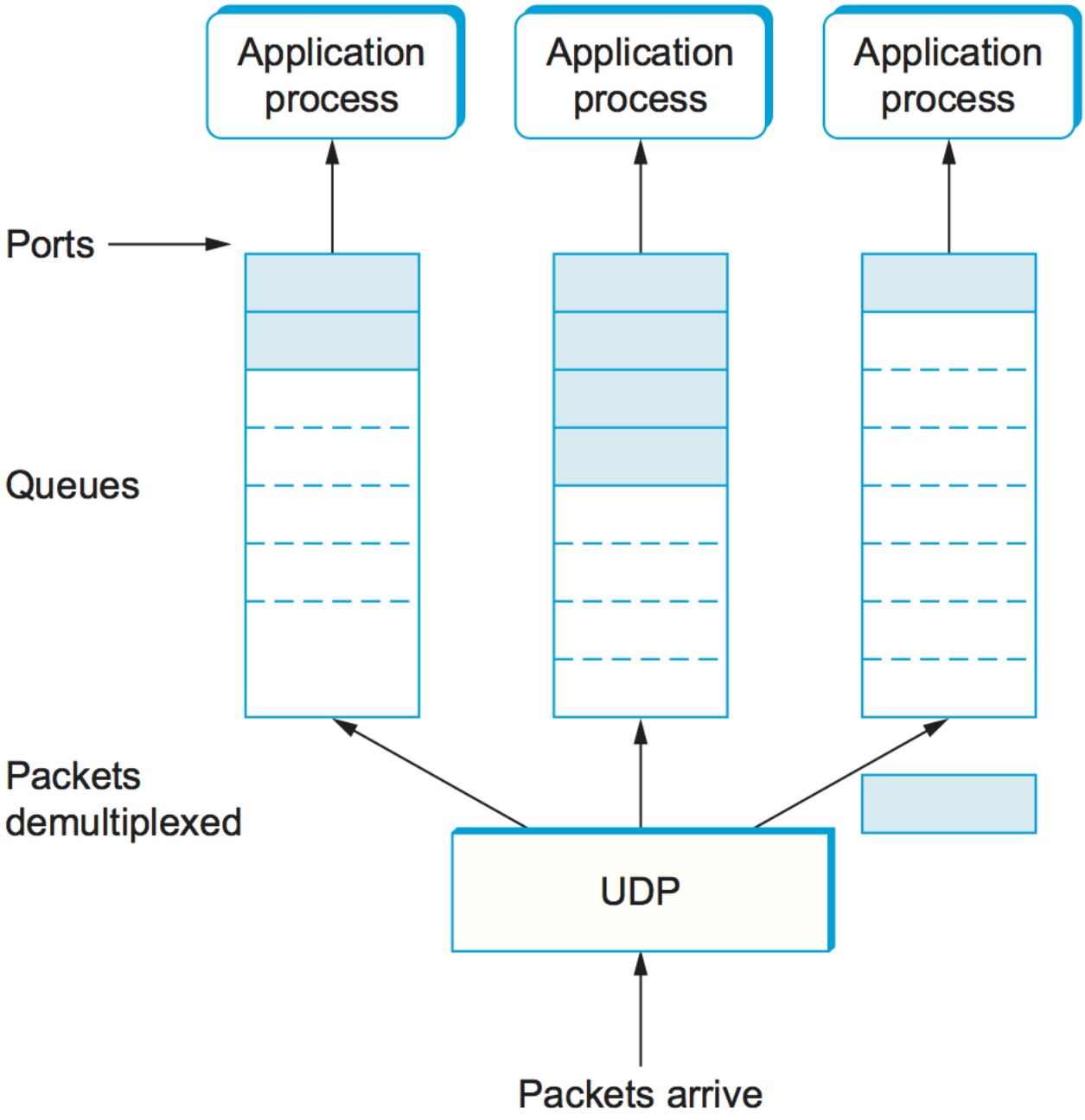

Figura 126. *Fila de mensagens UDP.*

Por fim, embora o UDP não implemente controle de fluxo ou entrega confiável/ordenada, ele oferece mais uma função além de desmultiplexar mensagens para algum processo de aplicação — ele também garante a exatidão da mensagem por meio de uma soma de verificação. (A soma de verificação do UDP é opcional no IPv4,

mas obrigatória no IPv6.) O algoritmo básico de soma de verificação do UDP é o mesmo usado para IP — ou seja, ele soma um conjunto de palavras de 16 bits usando aritmética de complemento de um e toma o complemento de um do resultado. Mas os dados de entrada usados para a soma de verificação são um pouco contraintuitivos.

A soma de verificação UDP recebe como entrada o cabeçalho UDP, o conteúdo do corpo da mensagem e algo chamado *pseudocabeçalho* . O pseudocabeçalho consiste em três campos do cabeçalho IP — número do protocolo, endereço IP de origem e endereço IP de destino — além do campo de comprimento UDP. (Sim, o campo de comprimento UDP é incluído duas vezes no cálculo da soma de verificação.) A motivação por trás do pseudocabeçalho é verificar se a mensagem foi entregue entre os dois pontos de extremidade corretos. Por exemplo, se o endereço IP de destino foi modificado enquanto o pacote estava em trânsito, causando sua entrega incorreta, esse fato seria detectado pela soma de verificação UDP.

5.2 Fluxo de Bytes Confiável (TCP)

Em contraste com um protocolo de demultiplexação simples como o UDP, um protocolo de transporte mais sofisticado é aquele que oferece um serviço de fluxo de bytes confiável, orientado à conexão. Esse serviço tem se mostrado útil para uma ampla gama de aplicações, pois libera a aplicação da preocupação com dados perdidos ou reordenados. O Protocolo de Controle de Transmissão da Internet é provavelmente o protocolo mais utilizado desse tipo; é também o mais cuidadosamente ajustado. É por essas duas razões que esta seção estuda o TCP em detalhes, embora identifiquemos e discutamos opções alternativas de projeto no final da seção.

Em termos das propriedades dos protocolos de transporte apresentadas na descrição do problema no início deste capítulo, o TCP garante a entrega confiável e ordenada de um fluxo de bytes. É um protocolo full-duplex, o que significa que cada conexão TCP

suporta um par de fluxos de bytes, um fluindo em cada direção. Ele também inclui um mecanismo de controle de fluxo para cada um desses fluxos de bytes, que permite ao receptor limitar a quantidade de dados que o remetente pode transmitir em um determinado momento. Por fim, assim como o UDP, o TCP suporta um mecanismo de demultiplexação que permite que vários programas aplicativos em qualquer host mantenham uma conversação simultânea com seus pares.

Além dos recursos acima, o TCP também implementa um mecanismo de controle de congestionamento altamente ajustado. A ideia desse mecanismo é controlar a velocidade de envio de dados pelo TCP, não para evitar que o remetente sobrecarregue o destinatário, mas para evitar que o remetente sobrecarregue a rede. A descrição do mecanismo de controle de congestionamento do TCP será adiada para o próximo capítulo, onde o discutiremos no contexto mais amplo de como os recursos de rede são alocados de forma justa.

Como muitas pessoas confundem controle de congestionamento com controle de fluxo, reiteramos a diferença. *O controle de fluxo* envolve evitar que os remetentes excedam a capacidade dos destinatários. *O controle de congestionamento* envolve evitar que muitos dados sejam injetados na rede, causando sobrecarga de switches ou links. Portanto, o controle de fluxo é uma questão de ponta a ponta, enquanto o controle de congestionamento se preocupa com a forma como os hosts e as redes interagem.

5.2.1 Problemas de ponta a ponta

No cerne do TCP está o algoritmo de janela deslizante. Embora este seja o mesmo algoritmo básico frequentemente usado no nível do enlace, como o TCP opera pela internet em vez de um enlace físico ponto a ponto, existem muitas diferenças importantes. Esta subseção identifica essas diferenças e explica como elas complicam o TCP. As subseções seguintes descrevem como o TCP aborda essas e outras complicações.

Primeiro, enquanto o algoritmo de janela deslizante em nível de link apresentado é executado em um único link físico que sempre conecta os mesmos dois computadores, o TCP suporta conexões lógicas entre processos que estão sendo executados em quaisquer dois computadores na Internet. Isso significa que o TCP precisa de uma fase explícita de estabelecimento de conexão, durante a qual os dois lados da conexão concordam em trocar dados entre si. Essa diferença é análoga a ter que discar para a outra parte, em vez de ter uma linha telefônica dedicada. O TCP também tem uma fase explícita de desconexão da conexão. Uma das coisas que acontecem durante o estabelecimento da conexão é que as duas partes estabelecem algum estado compartilhado para permitir que o algoritmo de janela deslizante comece. A desconexão da conexão é necessária para que cada host saiba que está tudo bem liberar esse estado.

Em segundo lugar, enquanto um único link físico que sempre conecta os mesmos dois computadores tem um tempo de ida e volta (RTT) fixo, as conexões TCP provavelmente têm tempos de ida e volta muito diferentes. Por exemplo, uma conexão TCP entre um host em São Francisco e um host em Boston, que estão separados por vários milhares de quilômetros, pode ter um RTT de 100 ms, enquanto uma conexão TCP entre dois hosts na mesma sala, a apenas alguns metros de distância, pode ter um RTT de apenas 1 ms. O mesmo protocolo TCP deve ser capaz de suportar ambas as conexões. Para piorar a situação, a conexão TCP entre hosts em São Francisco e Boston pode ter um RTT de 100 ms às 3 da manhã, mas um RTT de 500 ms às 15 da tarde. Variações no RTT são possíveis até mesmo durante uma única conexão TCP que dura apenas alguns minutos. O que isso significa para o algoritmo de janela deslizante é que o mecanismo de tempo limite que aciona as retransmissões deve ser adaptativo. (Certamente, o tempo limite para um link ponto a ponto deve ser um parâmetro configurável, mas não é necessário adaptar esse temporizador para um par específico de nós.)

Uma terceira diferença é que os pacotes podem ser reordenados à medida que cruzam a Internet, mas isso não é possível em um link ponto a ponto, onde o primeiro pacote

colocado em uma extremidade do link deve ser o primeiro a aparecer na outra extremidade. Pacotes ligeiramente fora de ordem não causam problemas, pois o algoritmo de janela deslizante pode reordenar os pacotes corretamente usando o número de sequência. A verdadeira questão é o quão fora de ordem os pacotes podem chegar ou, dito de outra forma, o quão atrasado um pacote pode chegar ao destino. Na pior das hipóteses, um pacote pode ser atrasado na Internet até que o `TTL`campo time to live ( ) do IP expire, momento em que o pacote é descartado (e, portanto, não há perigo de chegar atrasado). Sabendo que o IP descarta pacotes após sua `TTL` expiração, o TCP assume que cada pacote tem um tempo de vida máximo. O tempo de vida exato, conhecido como *tempo de vida máximo do segmento* (MSL), é uma escolha de engenharia. A configuração recomendada atualmente é de 120 segundos. Lembre-se de que o IP não impõe diretamente esse valor de 120 segundos; Trata-se simplesmente de uma estimativa conservadora que o TCP faz de quanto tempo um pacote pode permanecer na internet. A implicação é significativa — o TCP precisa estar preparado para o caso de pacotes muito antigos chegarem repentinamente ao receptor, potencialmente confundindo o algoritmo de janela deslizante.

Em quarto lugar, os computadores conectados a um enlace ponto a ponto são geralmente projetados para suportar o enlace. Por exemplo, se o produto atraso × largura de banda de um enlace for calculado como 8 KB — o que significa que um tamanho de janela é selecionado para permitir que até 8 KB de dados não sejam reconhecidos em um determinado momento — então é provável que os computadores em cada extremidade do enlace tenham a capacidade de armazenar em buffer até 8 KB de dados. Projetar o sistema de outra forma seria tolice. Por outro lado, quase qualquer tipo de computador pode ser conectado à Internet, tornando a quantidade de recursos dedicada a qualquer conexão TCP altamente variável, especialmente considerando que qualquer host pode potencialmente suportar centenas de conexões TCP ao mesmo tempo. Isso significa que o TCP deve incluir um mecanismo que cada lado use para "aprender" quais recursos (por exemplo, quanto espaço de buffer) o outro lado é capaz de aplicar à conexão. Essa é a questão do controle de fluxo.

Quinto, como o lado transmissor de um link conectado diretamente não pode enviar mais rápido do que a largura de banda do link permite, e apenas um host está enviando dados para o link, não é possível congestionar o link sem saber. Em outras palavras, a carga no link é visível na forma de uma fila de pacotes no remetente. Em contraste, o lado remetente de uma conexão TCP não tem ideia de quais links serão percorridos para chegar ao destino. Por exemplo, a máquina remetente pode estar conectada diretamente a uma Ethernet relativamente rápida — e capaz de enviar dados a uma taxa de 10 Gbps — mas em algum lugar no meio da rede, um link de 1,5 Mbps precisa ser percorrido. E, para piorar a situação, dados gerados por muitas fontes diferentes podem estar tentando atravessar esse mesmo link lento. Isso leva ao problema de congestionamento da rede. A discussão deste tópico foi adiada para o próximo capítulo.

Concluímos esta discussão sobre questões de ponta a ponta comparando a abordagem do TCP para fornecer um serviço de entrega confiável/ordenado com a abordagem usada por redes baseadas em circuitos virtuais, como a historicamente importante rede X.25. No TCP, a rede IP subjacente é considerada não confiável e entrega mensagens fora de ordem; o TCP usa o algoritmo de janela deslizante de ponta a ponta para fornecer entrega confiável/ordenada. Em contraste, as redes X.25 usam o protocolo de janela deslizante dentro da rede, salto a salto. A suposição por trás dessa abordagem é que, se as mensagens são entregues de forma confiável e ordenada entre cada par de nós ao longo do caminho entre o host de origem e o host de destino, o serviço de ponta a ponta também garante uma entrega confiável/ordenada.

O problema com esta última abordagem é que uma sequência de garantias salto a salto não necessariamente se soma a uma garantia de ponta a ponta. Primeiro, se um link heterogêneo (digamos, uma Ethernet) for adicionado a uma extremidade do caminho, não há garantia de que este salto preservará o mesmo serviço que os outros saltos. Segundo, só porque o protocolo de janela deslizante garante que as mensagens sejam entregues corretamente do nó A para o nó B e, em seguida, do nó B para o nó

C, ele não garante que o nó B se comporte perfeitamente. Por exemplo, sabe-se que nós de rede introduzem erros em mensagens ao transferi-las de um buffer de entrada para um buffer de saída. Eles também são conhecidos por reordenar mensagens acidentalmente. Como consequência dessas pequenas janelas de vulnerabilidade, ainda é necessário fornecer verdadeiras verificações de ponta a ponta para garantir um serviço confiável/ordenado, mesmo que os níveis mais baixos do sistema também implementem essa funcionalidade.

Conclusão principal

Esta discussão serve para ilustrar um dos princípios mais importantes no projeto de sistemas — o *argumento de ponta a ponta* . Em resumo, o argumento de ponta a ponta afirma que uma função (em nosso exemplo, fornecer entrega confiável/ordenada) não deve ser fornecida nos níveis mais baixos do sistema, a menos que possa ser implementada completa e corretamente nesse nível. Portanto, essa regra argumenta a favor da abordagem TCP/IP. No entanto, essa regra não é absoluta. Ela permite que funções sejam fornecidas de forma incompleta em um nível baixo como uma otimização de desempenho. É por isso que é perfeitamente consistente com o argumento de ponta a ponta realizar a detecção de erros (por exemplo, CRC) salto a salto; detectar e retransmitir um único pacote corrompido em um salto é preferível a ter que retransmitir um arquivo inteiro de ponta a ponta. [Próximo]

5.2.2 Formato do Segmento

O TCP é um protocolo orientado a bytes, o que significa que o remetente grava bytes em uma conexão TCP e o destinatário lê bytes da conexão TCP. Embora “fluxo de bytes” descreva o serviço que o TCP oferece aos processos de aplicação, o TCP não transmite bytes individuais pela Internet. Em vez disso, o TCP no host de origem armazena em buffer bytes suficientes do processo de envio para preencher um pacote de tamanho razoável e então envia esse pacote para seu par no host de destino. O TCP no host de destino então esvazia o conteúdo do pacote em um buffer de

recebimento, e o processo de recebimento lê desse buffer quando quiser. Essa situação é ilustrada na Figura 127 , que, para simplificar, mostra os dados fluindo em apenas uma direção. Lembre-se de que, em geral, uma única conexão TCP suporta fluxos de bytes fluindo em ambas as direções.

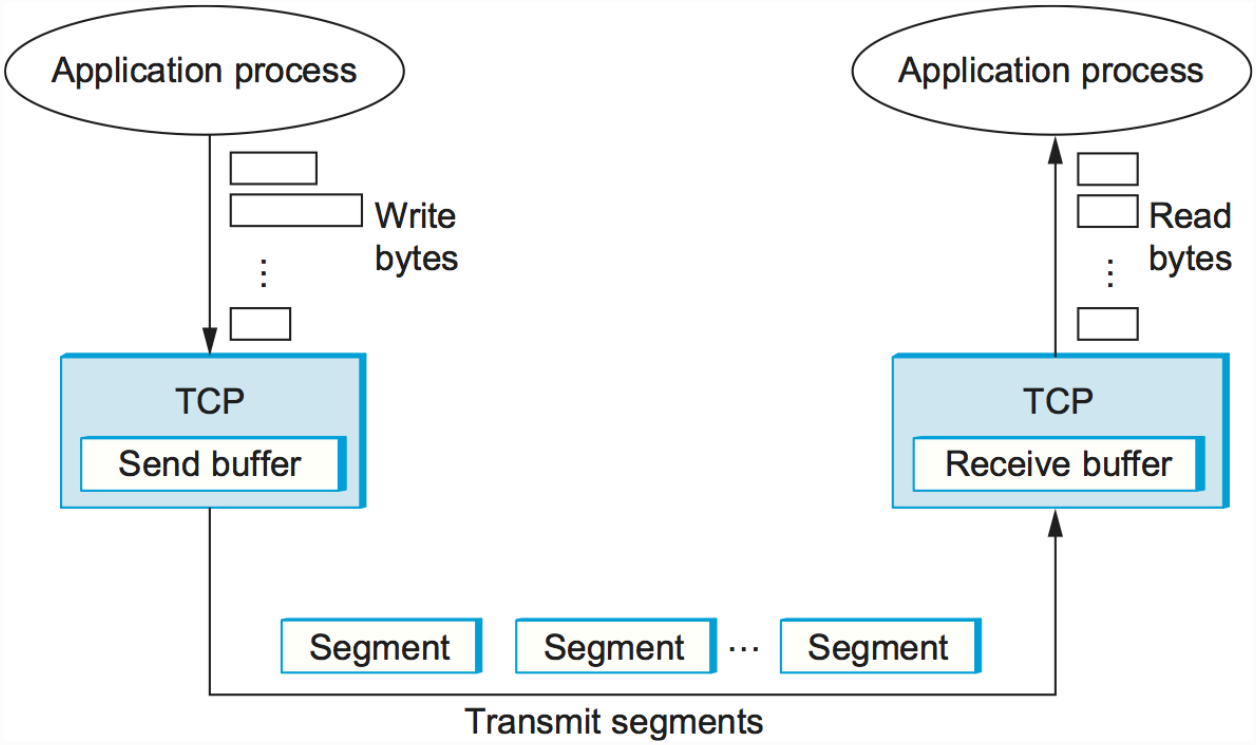

Figura 127. *Como o TCP gerencia um fluxo de bytes.*

Os pacotes trocados entre pares TCP na Figura 127 são chamados de *segmentos* , pois cada um carrega um segmento do fluxo de bytes. Cada segmento TCP contém o cabeçalho esquematicamente representado na Figura 128. A relevância da maioria desses campos ficará evidente ao longo desta seção. Por enquanto, vamos apenas apresentá-los.

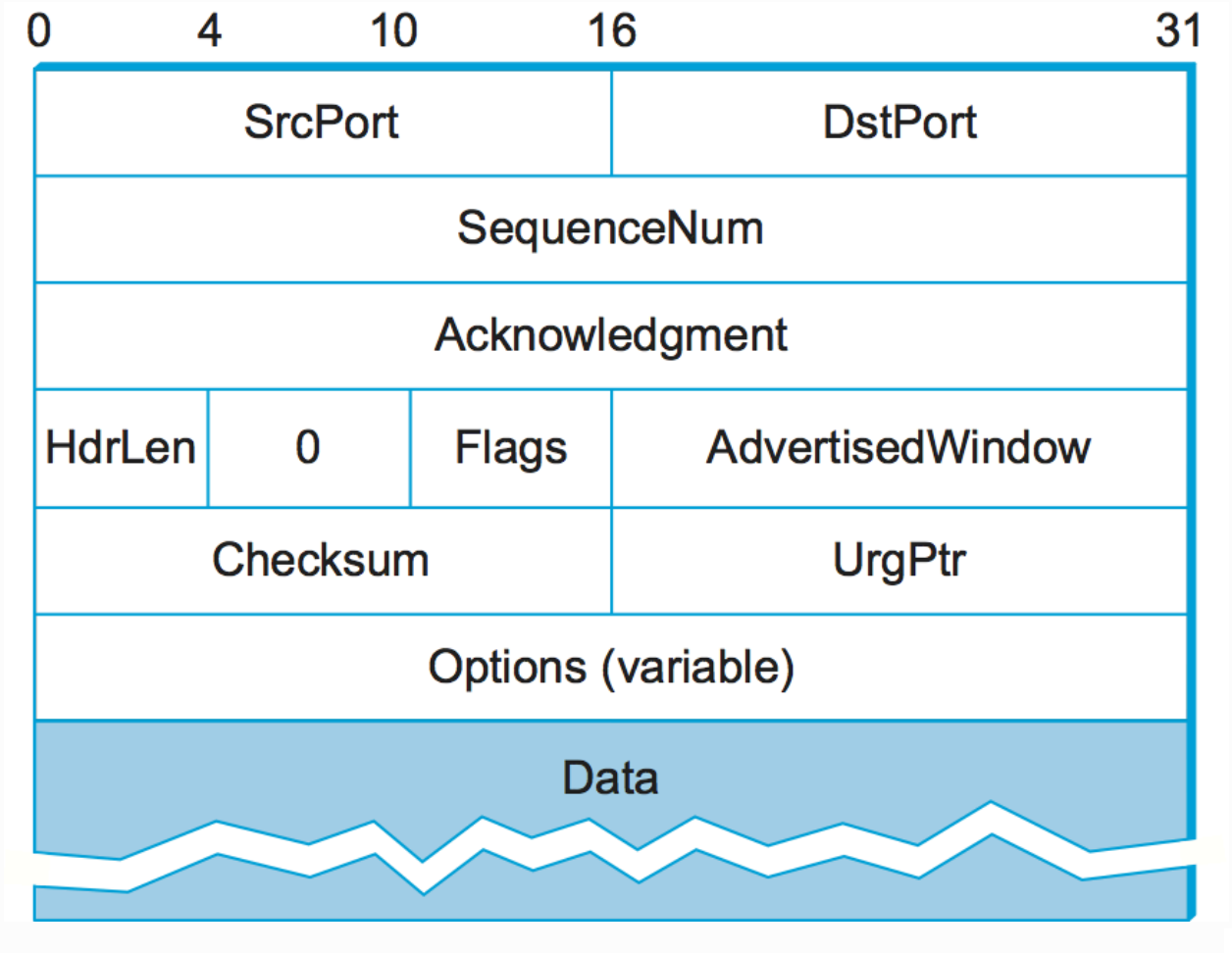

Figura 128. *Formato do cabeçalho TCP.*

Os campos `SrcPort` e `DstPort` identificam as portas de origem e destino, respectivamente, assim como no UDP. Esses dois campos, mais os endereços IP de origem e destino, combinam-se para identificar exclusivamente cada conexão TCP. Ou seja, a chave de demux do TCP é dada pela tupla quádrupla

```
(SrcPort, SrcIPAddr, DstPort, DstIPAddr)
```

Observe que, como as conexões TCP vêm e vão, é possível que uma conexão entre um determinado par de portas seja estabelecida, usada para enviar e receber dados e, posteriormente, encerrada, e posteriormente o mesmo par de portas seja envolvido em

uma segunda conexão. Às vezes, nos referimos a essa situação como duas *encarnações* diferentes da mesma conexão.

Os campos `Acknowledgement`, `SequenceNum`e `AdvertisedWindow` estão todos envolvidos no algoritmo de janela deslizante do TCP. Como o TCP é um protocolo orientado a bytes, cada byte de dados tem um número de sequência. O `SequenceNum`campo contém o número de sequência do primeiro byte de dados transportado naquele segmento, e os campos `Acknowledgement`e `AdvertisedWindow`carregam informações sobre o fluxo de dados indo na outra direção. Para simplificar nossa discussão, ignoramos o fato de que os dados podem fluir em ambas as direções e nos concentramos em dados que têm um determinado `SequenceNum`fluxo em uma direção e `Acknowledgement`valores `AdvertisedWindow`e fluindo na direção oposta, conforme ilustrado na Figura 129. O uso desses três campos é descrito mais detalhadamente posteriormente neste capítulo.

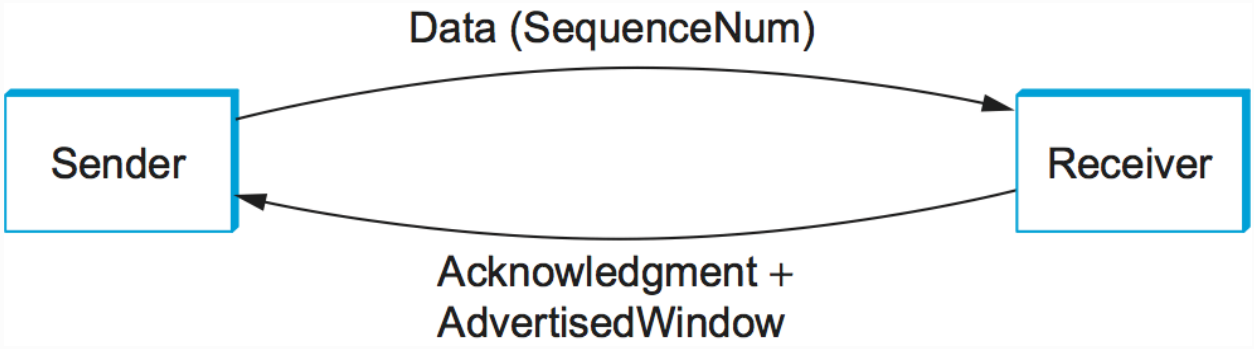

Figura 129. *Ilustração simplificada (mostrando apenas uma direção) do processo TCP, com fluxo de dados em uma direção e ACKs na outra.*

O campo de 6 bits `Flags`é usado para retransmitir informações de controle entre pares TCP. Os possíveis sinalizadores incluem `SYN`, `FIN`, `RESET`, `PUSH`, `URG`e `ACK`. Os sinalizadores `SYN`e `FIN`são usados ao estabelecer e encerrar uma conexão TCP, respectivamente. Seu uso é descrito em uma seção posterior. O `ACK`sinalizador é definido sempre que o `Acknowledgement`campo é válido, o que implica que o receptor deve prestar atenção a ele. O `URG`sinalizador significa que este segmento contém dados urgentes. Quando este sinalizador é definido, o `UrgPtr`campo indica onde os dados não urgentes contidos

neste segmento começam. Os dados urgentes estão contidos na frente do corpo do segmento, até e incluindo um valor de `UrgPtr`bytes no segmento. O `PUSH` sinalizador significa que o remetente invocou a operação push, o que indica ao lado receptor do TCP que ele deve notificar o processo receptor desse fato. Discutiremos esses dois últimos recursos mais detalhadamente em uma seção posterior. Finalmente, o `RESET`sinalizador significa que o receptor ficou confuso — por exemplo, porque recebeu um segmento que não esperava receber — e, portanto, deseja abortar a conexão.

Por fim, o `Checksum`campo é usado exatamente da mesma forma que para o UDP — ele é calculado sobre o cabeçalho TCP, os dados TCP e o pseudocabeçalho, que é composto pelos campos de endereço de origem, endereço de destino e comprimento do cabeçalho IP. A soma de verificação é necessária para TCP tanto em IPv4 quanto em IPv6. Além disso, como o cabeçalho TCP tem comprimento variável (opções podem ser anexadas após os campos obrigatórios), `HdrLen`é incluído um campo que fornece o comprimento do cabeçalho em palavras de 32 bits. Este campo também é conhecido como `Offset`campo, pois mede o deslocamento do início do pacote até o início dos dados.

5.2.3 Estabelecimento e Término de Conexão

Uma conexão TCP começa com um cliente (chamador) fazendo uma abertura ativa para um servidor (chamado). Supondo que o servidor tenha feito uma abertura passiva anteriormente, os dois lados se envolvem em uma troca de mensagens para estabelecer a conexão. (Lembre-se do Capítulo 1 que uma parte que deseja iniciar uma conexão executa uma abertura ativa, enquanto uma parte disposta a aceitar uma conexão faz uma abertura passiva. 1) Somente após o término dessa fase de estabelecimento de conexão os dois lados começam a enviar dados. Da mesma forma, assim que um participante termina de enviar dados, ele fecha uma direção da conexão, o que faz com que o TCP inicie uma rodada de mensagens de término de conexão. Observe que, enquanto a configuração da conexão é uma atividade assimétrica (um lado faz uma abertura passiva e o outro lado faz uma abertura ativa), a desconexão da

conexão é simétrica (cada lado tem que fechar a conexão independentemente). Portanto, é possível que um lado tenha feito um fechamento, o que significa que ele não pode mais enviar dados, mas para o outro lado manter a outra metade da conexão bidirecional aberta e continuar enviando dados.

1

Para ser mais preciso, o TCP permite que a configuração da conexão seja simétrica, com ambos os lados tentando abrir a conexão ao mesmo tempo, mas o caso comum é que um lado faça uma abertura ativa e o outro lado faça uma abertura passiva.

Aperto de mão triplo

O algoritmo usado pelo TCP para estabelecer e encerrar uma conexão é chamado de *handshake triplo* . Primeiro, descrevemos o algoritmo básico e, em seguida, mostramos como ele é usado pelo TCP. O handshake triplo envolve a troca de três mensagens entre o cliente e o servidor, conforme ilustrado pela linha do tempo apresentada na Figura 130 .

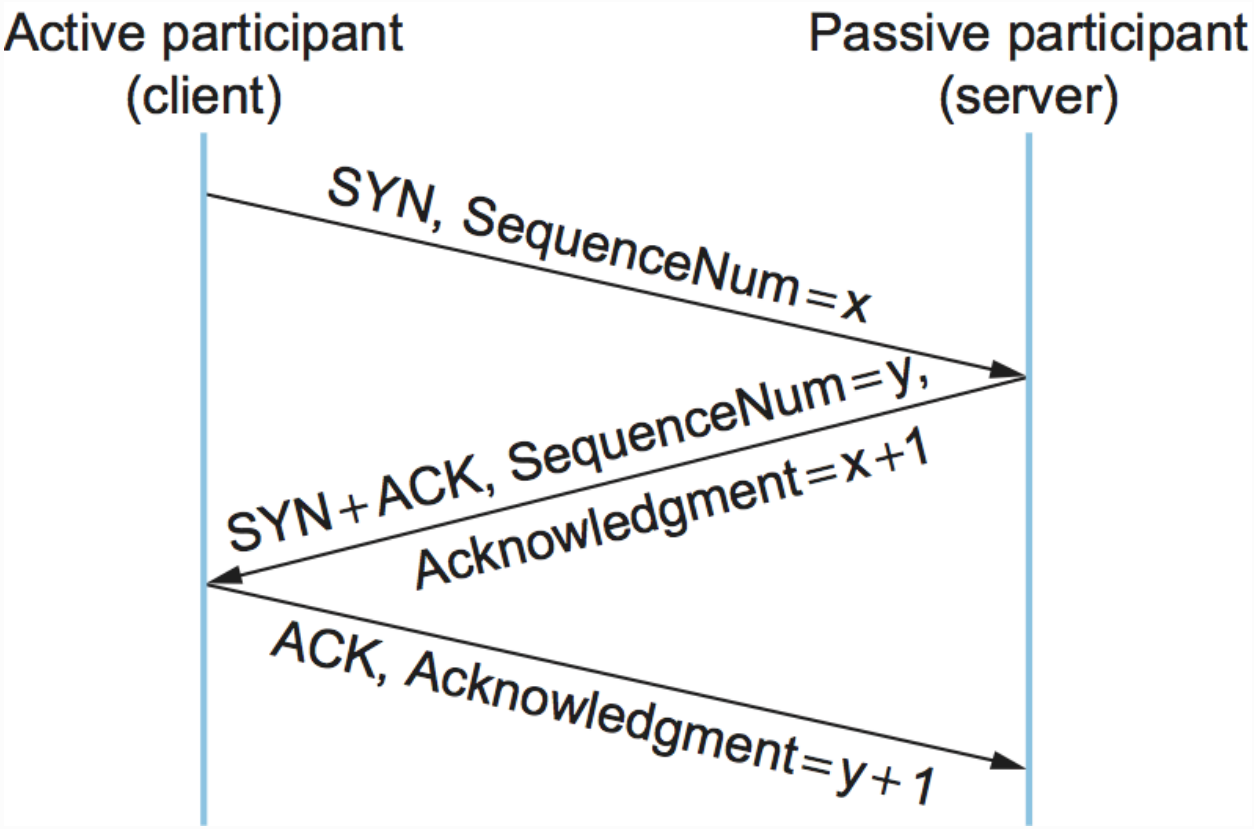

Figura 130. *Linha do tempo para o algoritmo de handshake de três vias.*

A ideia é que duas partes queiram concordar com um conjunto de parâmetros, que, no caso de abertura de uma conexão TCP, são os números de sequência iniciais que os dois lados planejam usar para seus respectivos fluxos de bytes. Em geral, os parâmetros podem ser quaisquer fatos que cada lado queira que o outro saiba. Primeiro, o cliente (o participante ativo) envia um segmento para o servidor (o participante passivo) informando o número de sequência inicial que planeja usar (`Flags`= `SYN`, `SequenceNum`= x). O servidor então responde com um único segmento que reconhece o número de sequência do cliente ( ) e declara seu próprio número de sequência inicial ( ). Ou seja, os bits e são definidos no campo desta segunda mensagem. Finalmente, o cliente responde com um terceiro segmento que reconhece o número de sequência do servidor ( ). A razão pela qual cada lado reconhece um número de sequência que é uma unidade maior do que o enviado é que o campo, na verdade, identifica o "próximo número de sequência esperado", reconhecendo

implicitamente todos os números de sequência anteriores. Embora não seja mostrado nesta linha do tempo, um cronômetro é agendado para cada um dos dois primeiros segmentos e, se a resposta esperada não for recebida, o segmento é retransmitido.`Flags = ACK, Ack = x + 1Flags = SYN, SequenceNum = ySYNACKFlagsFlags = ACK, Ack = y + 1Acknowledgement`

Você pode estar se perguntando por que o cliente e o servidor precisam trocar números de sequência iniciais entre si no momento da configuração da conexão. Seria mais simples se cada lado simplesmente começasse em algum número de sequência "bem conhecido", como 0. De fato, a especificação TCP exige que cada lado de uma conexão selecione um número de sequência inicial aleatoriamente. A razão para isso é proteger contra duas encarnações da mesma conexão que reutilizam os mesmos números de sequência muito cedo — isto é, enquanto ainda há a chance de que um segmento de uma encarnação anterior de uma conexão possa interferir em uma encarnação posterior da conexão.

Diagrama de Transição de Estado

O TCP é complexo o suficiente para que sua especificação inclua um diagrama de transição de estado. Uma cópia desse diagrama é apresentada na Figura 131. Este diagrama mostra apenas os estados envolvidos na abertura de uma conexão (tudo acima de ESTABLISHED) e no fechamento de uma conexão (tudo abaixo de ESTABLISHED). Tudo o que acontece enquanto uma conexão está aberta — ou seja, a operação do algoritmo de janela deslizante — fica oculto no estado ESTABLISHED.

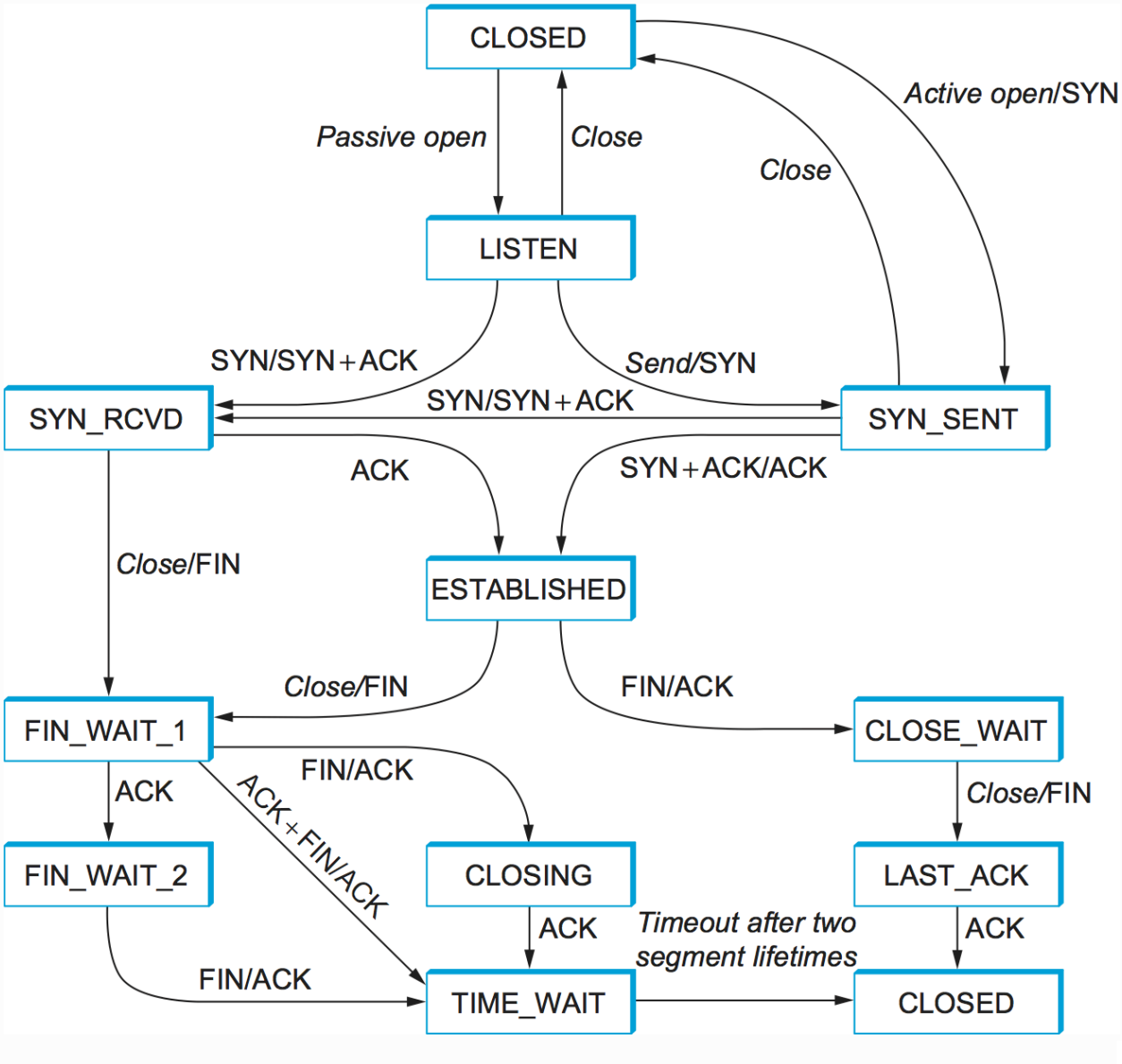

Figura 131. *Diagrama de transição de estado do TCP.*

O diagrama de transição de estado do TCP é bastante fácil de entender. Cada caixa denota um estado em que uma extremidade de uma conexão TCP pode se encontrar. Todas as conexões começam no estado FECHADO. À medida que a conexão progride, ela se move de um estado para outro de acordo com os arcos. Cada arco é rotulado com uma tag no formato *evento/ação* . Assim, se uma conexão estiver no estado LISTEN e um segmento SYN chegar (ou seja, um segmento com o `SYN`sinalizador

definido), a conexão fará uma transição para o estado SYN_RCVD e executará a ação de responder com um `ACK+SYN`segmento.

Observe que dois tipos de eventos acionam uma transição de estado: (1) um segmento chega do peer (por exemplo, o evento no arco de LISTEN para SYN_RCVD), ou (2) o processo de aplicação local invoca uma operação no TCP (por exemplo, o evento *de abertura ativa* no arco de CLOSED para SYN_SENT). Em outras palavras, o diagrama de transição de estado do TCP define efetivamente a *semântica* tanto de sua interface peer-to-peer quanto de sua interface de serviço. A *sintaxe* dessas duas interfaces é dada pelo formato do segmento (como ilustrado na Figura 128) e por alguma interface de programação de aplicação, como a API de soquete, respectivamente.

Agora, vamos rastrear as transições típicas realizadas no diagrama da Figura 131. Lembre-se de que, em cada extremidade da conexão, o TCP realiza diferentes transições de um estado para outro. Ao abrir uma conexão, o servidor primeiro invoca uma operação de abertura passiva no TCP, o que faz com que o TCP passe para o estado LISTEN. Posteriormente, o cliente realiza uma abertura ativa, o que faz com que sua extremidade da conexão envie um segmento SYN para o servidor e passe para o estado SYN_SENT. Quando o segmento SYN chega ao servidor, ele passa para o estado SYN_RCVD e responde com um segmento SYN+ACK. A chegada desse segmento faz com que o cliente passe para o estado ESTABLISHED e envie um ACK de volta para o servidor. Quando esse ACK chega, o servidor finalmente passa para o estado ESTABLISHED. Em outras palavras, acabamos de rastrear o handshake triplo.

Há três coisas a serem observadas sobre a metade do diagrama de transição de estado referente ao estabelecimento da conexão. Primeiro, se o ACK do cliente para o servidor for perdido, correspondendo à terceira etapa do handshake triplo, a conexão ainda funcionará corretamente. Isso ocorre porque o lado do cliente já está no estado ESTABELECIDO, de modo que o processo do aplicativo local pode começar a enviar dados para a outra extremidade. Cada um desses segmentos de dados terá o `ACK`sinalizador definido e o valor correto no `Acknowledgement`campo, de modo que o servidor passará para o estado ESTABELECIDO quando o primeiro segmento de

dados chegar. Este é, na verdade, um ponto importante sobre o TCP — cada segmento informa qual número de sequência o remetente espera ver em seguida, mesmo que isso repita o mesmo número de sequência contido em um ou mais segmentos anteriores.

A segunda coisa a se notar sobre o diagrama de transição de estado é que há uma transição estranha para fora do estado LISTEN sempre que o processo local invoca uma operação *de envio* no TCP. Ou seja, é possível que um participante passivo identifique ambas as extremidades da conexão (ou seja, ele próprio e o participante remoto que deseja que se conecte a ele) e, em seguida, mude de ideia sobre esperar pelo outro lado e, em vez disso, estabeleça ativamente a conexão. Até onde sabemos, esta é uma característica do TCP da qual nenhum processo de aplicação realmente se aproveita.

O último ponto a ser observado no diagrama são os arcos que não são mostrados. Especificamente, a maioria dos estados que envolvem o envio de um segmento para o outro lado também agenda um tempo limite que eventualmente faz com que o segmento seja reenviado caso a resposta esperada não ocorra. Essas retransmissões não são representadas no diagrama de transição de estado. Se, após várias tentativas, a resposta esperada não chegar, o TCP desiste e retorna ao estado FECHADO.

Voltando nossa atenção agora para o processo de encerramento de uma conexão, o importante a ter em mente é que o processo de aplicação em ambos os lados da conexão deve fechar sua metade da conexão independentemente. Se apenas um lado fechar a conexão, isso significa que ele não tem mais dados para enviar, mas ainda está disponível para receber dados do outro lado. Isso complica o diagrama de transição de estado, pois ele deve levar em conta a possibilidade de os dois lados invocarem o operador *close* ao mesmo tempo, bem como a possibilidade de que primeiro um lado invoque close e, em algum momento posterior, o outro lado invoque close. Assim, em qualquer lado, há três combinações de transições que levam uma conexão do estado ESTABLISHED (ESTABELECIDO) para o estado CLOSED (FECHADO):

- Este lado fecha primeiro: ESTABELECIDO
- →
  - FIN_WAIT_1
- →
  - FIN_WAIT_2
- →
  - TEMPO_DE_ESPERA
- →
- FECHADO.
  - O outro lado fecha primeiro: ESTABELECIDO
- →
  - FECHAR_AGUARDAR
- →
  - ÚLTIMO_ACK
- →
- FECHADO.
  - Ambos os lados fecham ao mesmo tempo: ESTABELECIDO
- →
  - FIN_WAIT_1
- →
  - ENCERRAMENTO
- →
  - TEMPO_DE_ESPERA
- →
- FECHADO.

Na verdade, existe uma quarta sequência de transições, embora rara, que leva ao estado FECHADO; ela segue o arco de FIN_WAIT_1 a TIME_WAIT. Deixamos como exercício para você descobrir qual combinação de circunstâncias leva a essa quarta possibilidade.

O principal aspecto a ser reconhecido sobre a interrupção de conexão é que uma conexão no estado TIME_WAIT não pode passar para o estado CLOSED até que tenha esperado duas vezes o tempo máximo que um datagrama IP pode viver na Internet (ou seja, 120 segundos). A razão para isso é que, embora o lado local da conexão tenha enviado um ACK em resposta ao segmento FIN do outro lado, ele não sabe que o ACK foi entregue com sucesso. Como consequência, o outro lado pode retransmitir seu segmento FIN, e este segundo segmento FIN pode ser atrasado na rede. Se a conexão pudesse passar diretamente para o estado CLOSED, outro par de processos de aplicativo poderia aparecer e abrir a mesma conexão (ou seja, usar o mesmo par de números de porta), e o segmento FIN atrasado da encarnação anterior da conexão iniciaria imediatamente o encerramento da encarnação posterior dessa conexão.

5.2.4 Janela deslizante revisitada

Agora estamos prontos para discutir a variante do TCP do algoritmo de janela deslizante, que atende a vários propósitos: (1) garante a entrega confiável de dados, (2) assegura que os dados sejam entregues em ordem e (3) impõe o controle de fluxo entre o remetente e o destinatário. O uso do algoritmo de janela deslizante pelo TCP é o mesmo que no nível de enlace no caso das duas primeiras dessas três funções. Onde o TCP difere do algoritmo de nível de enlace é que ele também incorpora a função de controle de fluxo. Em particular, em vez de ter uma janela deslizante de tamanho fixo, o destinatário *anuncia* um tamanho de janela ao remetente. Isso é feito usando o `AdvertisedWindow`campo no cabeçalho TCP. O remetente fica então limitado a ter no máximo um valor de `AdvertisedWindow` bytes de dados não confirmados em um determinado momento. O destinatário seleciona um valor adequado com `AdvertisedWindow`base na quantidade de memória alocada à conexão para fins de buffer de dados. A ideia é evitar que o remetente sobrecarregue o buffer do destinatário. Discutiremos isso com mais detalhes a seguir.

Entrega confiável e ordenada

Para ver como os lados de envio e recebimento do TCP interagem entre si para implementar uma entrega confiável e ordenada, considere a situação ilustrada na Figura 132. O TCP, no lado de envio, mantém um buffer de envio. Esse buffer é usado para armazenar dados que foram enviados, mas ainda não confirmados, bem como dados que foram gravados pelo aplicativo de envio, mas não transmitidos. No lado de recebimento, o TCP mantém um buffer de recebimento. Esse buffer armazena dados que chegam fora de ordem, bem como dados que estão na ordem correta (ou seja, não há bytes faltantes anteriormente no fluxo), mas que o processo do aplicativo ainda não teve a chance de ler.

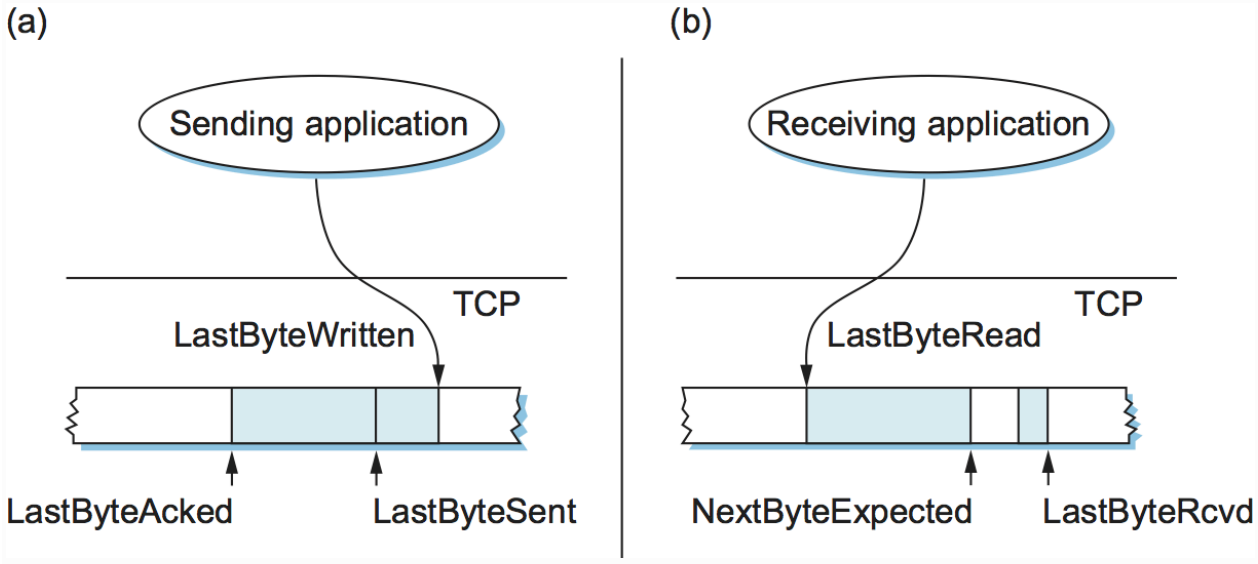

Figura 132. *Relação entre o buffer de envio TCP (a) e o buffer de recebimento (b).*

Para simplificar a discussão a seguir, ignoramos inicialmente o fato de que tanto os buffers quanto os números de sequência têm um tamanho finito e, portanto, eventualmente se repetem. Além disso, não distinguimos entre um ponteiro para um buffer onde um determinado byte de dados está armazenado e o número de sequência desse byte.

Olhando primeiro para o lado do envio, três ponteiros são mantidos no buffer de envio, cada um com um significado óbvio: `LastByteAcked`, `LastByteSent`, e `LastByteWritten`. Claramente,

```
LastByteAcked <= LastByteSent
```

já que o receptor não pode ter reconhecido um byte que ainda não foi enviado, e

```
LastByteSent <= LastByteWritten
```

já que o TCP não pode enviar um byte que o processo do aplicativo ainda não tenha escrito. Observe também que nenhum dos bytes à esquerda de `LastByteAcked`precisa ser salvo no buffer porque já foi confirmado, e nenhum dos bytes à direita de `LastByteWritten`precisa ser armazenado em buffer porque ainda não foi gerado.

Um conjunto semelhante de ponteiros (números de sequência) é mantido no lado receptor: `LastByteRead`, `NextByteExpected`, e `LastByteRcvd`. As desigualdades são um pouco menos intuitivas, no entanto, devido ao problema de entrega fora de ordem. O primeiro relacionamento

```
LastByteRead < NextByteExpected
```

é verdadeiro porque um byte não pode ser lido pelo aplicativo até que seja recebido *e* todos os bytes precedentes também tenham sido recebidos. `NextByteExpected`aponta para o byte imediatamente após o último byte que atende a este critério. Segundo,

```
NextByteExpected <= LastByteRcvd + 1
```

já que, se os dados chegaram em ordem, `NextByteExpected`aponta para o byte depois de `LastByteRcvd`, enquanto que se os dados chegaram fora de ordem, então `NextByteExpected`aponta para o início da primeira lacuna nos dados, como na Figura 132. Observe que os bytes à esquerda de `LastByteRead`não precisam ser armazenados em buffer porque já foram lidos pelo processo do aplicativo local, e os bytes à direita de `LastByteRcvd`não precisam ser armazenados em buffer porque ainda não chegaram.

Controle de fluxo

A maior parte da discussão acima é semelhante à encontrada no algoritmo de janela deslizante padrão; a única diferença real é que, desta vez, abordamos o fato de que os processos de envio e recebimento do aplicativo estão preenchendo e esvaziando seus buffers locais, respectivamente. (A discussão anterior encobriu o fato de que os dados que chegavam de um nó upstream estavam preenchendo o buffer de envio e os dados transmitidos para um nó downstream estavam esvaziando o buffer de recebimento.)

Você deve se certificar de que entendeu isso antes de prosseguir, pois agora chega o ponto em que os dois algoritmos diferem mais significativamente. A seguir, reintroduzimos o fato de que ambos os buffers têm um tamanho finito, denotado por `MaxSendBuffer`e `MaxRcvBuffer`, embora não nos preocupemos com os detalhes de como eles são implementados. Em outras palavras, estamos interessados apenas no número de bytes armazenados em buffer, não em onde esses bytes são realmente armazenados.

Lembre-se de que, em um protocolo de janela deslizante, o tamanho da janela define a quantidade de dados que pode ser enviada sem aguardar a confirmação do receptor. Assim, o receptor limita o remetente, anunciando uma janela que não é maior do que a quantidade de dados que ele pode armazenar em buffer. Observe que o TCP no lado do receptor deve manter

```
LastByteRcvd - LastByteRead <= MaxRcvBuffer
```

para evitar estourar seu buffer. Portanto, ele anuncia um tamanho de janela de

```
AdvertisedWindow = MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead)
```

que representa a quantidade de espaço livre restante em seu buffer. Conforme os dados chegam, o receptor os reconhece, desde que todos os bytes precedentes também tenham chegado. Além disso, `LastByteRcvd`move-se para a direita (é incrementado), o que significa que a janela anunciada potencialmente encolhe. Se ela encolhe ou não depende da velocidade com que o processo do aplicativo local está consumindo dados. Se o processo local estiver lendo os dados tão rápido quanto eles chegam (fazendo com que `LastByteRead`sejam incrementados na mesma taxa que `LastByteRcvd`), então a janela anunciada permanece aberta (ou seja,). Se, no entanto, o processo receptor ficar para trás, talvez porque ele execute uma operação muito cara em cada byte de dados que ele lê, então a janela anunciada fica menor com cada segmento que chega, até que eventualmente vá para 0.`AdvertisedWindow = MaxRcvBuffer`

O TCP no lado do envio deve então aderir à janela anunciada que recebe do receptor. Isso significa que, a qualquer momento, ele deve garantir que

```
LastByteSent - LastByteAcked <= AdvertisedWindow
```

Em outras palavras, o remetente calcula uma janela *efetiva* que limita a quantidade de dados que ele pode enviar:

```
EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)
```

Claramente, `EffectiveWindow`deve ser maior que 0 para que a origem possa enviar mais dados. É possível, portanto, que um segmento chegue reconhecendo x bytes, permitindo assim que o remetente incremente `LastByteAcked`em x, mas como o processo receptor não estava lendo nenhum dado, a janela anunciada agora é x bytes menor do que a anterior. Nessa situação, o remetente conseguiria liberar espaço no buffer, mas não enviaria mais dados.

Enquanto isso acontece, o lado de envio também deve garantir que o processo do aplicativo local não transborde o buffer de envio, ou seja,

```
LastByteWritten - LastByteAcked <= MaxSendBuffer
```

Se o processo de envio tentar gravar y bytes no TCP, mas

```
(LastByteWritten - LastByteAcked) + y > MaxSendBuffer
```

então o TCP bloqueia o processo de envio e não permite que ele gere mais dados.

Agora é possível entender como um processo de recebimento lento acaba interrompendo um processo de envio rápido. Primeiro, o buffer de recebimento fica cheio, o que significa que a janela anunciada diminui para 0. Uma janela anunciada de 0 significa que o lado remetente não pode transmitir nenhum dado, mesmo que os dados enviados anteriormente tenham sido confirmados com sucesso. Finalmente, não conseguir transmitir nenhum dado significa que o buffer de envio fica cheio, o que faz com que o TCP bloqueie o processo de envio. Assim que o processo de recebimento começa a ler os dados novamente, o TCP do lado do recebimento consegue abrir sua janela novamente, o que permite que o TCP do lado do envio transmita dados para fora de seu buffer. Quando esses dados são finalmente confirmados, `LastByteAcked`são

incrementados, o espaço do buffer que contém esses dados confirmados fica livre, e o processo de envio é desbloqueado e pode prosseguir.

Resta apenas um detalhe a ser resolvido: como o lado emissor sabe que a janela anunciada não é mais 0? Conforme mencionado acima, o TCP *sempre* envia um segmento em resposta a um segmento de dados recebido, e essa resposta contém os valores mais recentes para os campos `Acknowledge`e `AdvertisedWindow`, mesmo que esses valores não tenham sido alterados desde o último envio. O problema é o seguinte: depois que o lado receptor anuncia um tamanho de janela 0, o remetente não tem permissão para enviar mais dados, o que significa que não tem como descobrir que a janela anunciada não é mais 0 em algum momento no futuro. O TCP no lado receptor não envia espontaneamente segmentos que não são de dados; ele apenas os envia em resposta a um segmento de dados que chega.

O TCP lida com essa situação da seguinte maneira. Sempre que o outro lado anuncia um tamanho de janela igual a 0, o lado emissor persiste em enviar um segmento com 1 byte de dados de vez em quando. Ele sabe que esses dados provavelmente não serão aceitos, mas tenta mesmo assim, porque cada um desses segmentos de 1 byte dispara uma resposta que contém a janela anunciada atual. Eventualmente, uma dessas sondagens de 1 byte dispara uma resposta que relata uma janela anunciada diferente de zero.

Observe que essas mensagens de 1 byte são chamadas de *Sondas de Janela Zero* e, na prática, são enviadas a cada 5 a 60 segundos. Quanto ao byte de dados a ser enviado na sonda: é o próximo byte de dados reais logo fora da janela. (Precisam ser dados reais, caso sejam aceitos pelo receptor.)

Conclusão principal

Observe que o motivo pelo qual o lado emissor envia periodicamente este segmento de sondagem é que o TCP foi projetado para tornar o lado receptor o mais simples

possível — ele simplesmente responde aos segmentos do emissor e nunca inicia nenhuma atividade por conta própria. Este é um exemplo de uma regra de projeto de protocolo bem reconhecida (embora não universalmente aplicada), que, por falta de um nome melhor, chamamos de regra do *emissor inteligente/receptor burro* . Lembre-se de que vimos outro exemplo dessa regra quando discutimos o uso de NAKs no algoritmo de janela deslizante. [Próximo]

Protegendo contra o envoltório

Esta subseção e a próxima consideram o tamanho dos `SequenceNum` campos `AdvertisedWindow`e e as implicações de seus tamanhos na correção e no desempenho do TCP. `SequenceNum`O campo do TCP tem 32 bits de comprimento, e o seu `AdvertisedWindow`campo tem 16 bits, o que significa que o TCP atendeu facilmente ao requisito do algoritmo de janela deslizante de que o espaço do número de sequência seja duas vezes maior que o tamanho da janela: $2^{32} >> 2 \times 2^{16}$. No entanto, esse requisito não é o ponto interessante desses dois campos. Considere cada campo separadamente.

A relevância do espaço de números sequenciais de 32 bits é que o número sequencial usado em uma determinada conexão pode ser retomado — um byte com número sequencial S pode ser enviado de uma só vez e, posteriormente, um segundo byte com o mesmo número sequencial S pode ser enviado. Novamente, assumimos que os pacotes não podem sobreviver na Internet por mais tempo do que o MSL recomendado. Portanto, atualmente precisamos garantir que o número sequencial não seja retomado dentro de um período de 120 segundos. Se isso acontece ou não depende da velocidade com que os dados podem ser transmitidos pela Internet — ou seja, da velocidade com que o espaço de números sequenciais de 32 bits pode ser consumido. (Esta discussão pressupõe que estamos tentando consumir o espaço de números sequenciais o mais rápido possível, mas é claro que faremos isso se estivermos fazendo nosso trabalho de manter o canal cheio.) A Tabela 22 mostra quanto tempo leva para o número sequencial ser retomado em redes com várias larguras de banda.

Largura de banda	Tempo até o Wraparound
T1 (1,5 Mbps)	6,4 horas
T3 (45 Mbps)	13 minutos
Ethernet rápida (100 Mbps)	6 minutos
OC-3 (155 Mbps)	4 minutos
OC-48 (2,5 Gbps)	14 segundos
OC-192 (10 Gbps)	3 segundos
10 GigE (10 Gbps)	3 segundos

Como você pode ver, o espaço de números de sequência de 32 bits é adequado para larguras de banda modestas, mas, considerando que os links OC-192 agora são

comuns na infraestrutura da Internet e que a maioria dos servidores agora vem com interfaces Ethernet de 10 GB (ou 10 Gbps), já passamos do ponto em que 32 bits é um número pequeno demais. Felizmente, a IETF desenvolveu uma extensão para o TCP que efetivamente estende o espaço de números de sequência para proteger contra o enrolamento de números de sequência. Esta e outras extensões relacionadas serão descritas em uma seção posterior.

Mantendo o tubo cheio

A relevância do `AdvertisedWindow`campo de 16 bits é que ele deve ser grande o suficiente para permitir que o remetente mantenha o pipe cheio. Claramente, o destinatário tem a liberdade de não abrir a janela tão grande quanto o `AdvertisedWindow`campo permitir; estamos interessados na situação em que o destinatário tenha espaço de buffer suficiente para processar o máximo de dados possível `AdvertisedWindow`.

Nesse caso, não é apenas a largura de banda da rede, mas também o produto atraso x largura de banda que determina o tamanho do `AdvertisedWindow`campo — a janela precisa ser aberta o suficiente para permitir a transmissão de dados equivalentes ao produto atraso x largura de banda. Considerando um RTT de 100 ms (um número típico para uma conexão entre países nos Estados Unidos), a Tabela 23 apresenta o produto atraso x largura de banda para diversas tecnologias de rede.

Largura de banda	Produto de atraso × largura de banda
T1 (1,5 Mbps)	18 KB

T3 (45 Mbps)	549 KB
Ethernet rápida (100 Mbps)	1,2 MB
OC-3 (155 Mbps)	1,8 MB
OC-48 (2,5 Gbps)	29,6 MB
OC-192 (10 Gbps)	118,4 MB
10 GigE (10 Gbps)	118,4 MB

Como você pode ver, o campo do TCP `AdvertisedWindow`está em uma situação ainda pior do que o seu `SequenceNum`campo — ele não é grande o suficiente para lidar nem mesmo com uma conexão T3 através dos Estados Unidos continentais, já que um campo de 16 bits nos permite anunciar uma janela de apenas 64 KB. A mesma extensão TCP mencionada acima fornece um mecanismo para aumentar efetivamente o tamanho da janela anunciada.

5.2.5 Acionamento da Transmissão

Em seguida, consideramos uma questão surpreendentemente sutil: como o TCP decide transmitir um segmento. Conforme descrito anteriormente, o TCP suporta uma abstração de fluxo de bytes; ou seja, os programas de aplicação gravam bytes no fluxo, e cabe ao TCP decidir se possui bytes suficientes para enviar um segmento. Quais fatores regem essa decisão?

Se ignorarmos a possibilidade de controle de fluxo — ou seja, assumirmos que a janela está totalmente aberta, como seria o caso quando uma conexão é iniciada — o TCP possui três mecanismos para disparar a transmissão de um segmento. Primeiro, o TCP mantém uma variável, normalmente chamada de *tamanho máximo do segmento* (`MSS`), e envia um segmento assim que coleta `MSS`bytes do processo de envio. `MSS`geralmente é definido como o tamanho do maior segmento que o TCP pode enviar sem causar a fragmentação do IP local. Ou seja, `MSS`é definido como a unidade máxima de transmissão (MTU) da rede diretamente conectada, menos o tamanho dos cabeçalhos TCP e IP. O segundo fator que dispara o TCP para transmitir um segmento é que o processo de envio tenha solicitado explicitamente que ele o fizesse. Especificamente, o TCP suporta uma operação *push* , e o processo de envio invoca essa operação para efetivamente esvaziar o buffer de bytes não enviados. O gatilho final para a transmissão de um segmento é o disparo de um temporizador; o segmento resultante contém tantos bytes quantos estão atualmente armazenados em buffer para transmissão. No entanto, como veremos em breve, esse "temporizador" não é exatamente o que você espera.

Síndrome da Janela Tola

É claro que não podemos simplesmente ignorar o controle de fluxo, que desempenha um papel óbvio na limitação do remetente. Se o remetente tiver `MSS`bytes de dados para enviar e a janela estiver aberta pelo menos esse tempo, o remetente transmite um segmento completo. Suponha, no entanto, que o remetente esteja acumulando bytes para enviar, mas a janela esteja fechada no momento. Agora, suponha que um ACK chegue e efetivamente abra a janela o suficiente para que o remetente transmita, digamos, `MSS/2`bytes. O remetente deve transmitir um segmento pela metade ou

esperar que a janela abra completamente `MSS`? A especificação original não mencionava esse ponto, e as primeiras implementações do TCP decidiram prosseguir e transmitir um segmento pela metade. Afinal, não há como dizer quanto tempo levará até que a janela se abra novamente.

Acontece que a estratégia de aproveitar agressivamente qualquer janela disponível leva a uma situação agora conhecida como *síndrome da janela boba* . A Figura 133 ajuda a visualizar o que acontece. Se você pensar em um fluxo TCP como uma esteira rolante com contêineres "cheios" (segmentos de dados) indo em uma direção e contêineres vazios (ACKs) indo na direção reversa, então `MSS`segmentos de tamanho correspondem a contêineres grandes e segmentos de 1 byte correspondem a contêineres muito pequenos. Enquanto o remetente estiver enviando `MSS`segmentos de tamanho e o destinatário fizer ACKs de pelo menos um `MSS`dado por vez, tudo estará bem (Figura 133(a)). Mas, e se o destinatário tiver que reduzir a janela, de modo que em algum momento o remetente não possa enviar um pacote cheio `MSS`de dados? Se o remetente preencher agressivamente um contêiner menor que o `MSS`vazio assim que ele chegar, o destinatário fará o ACK desse número menor de bytes e, portanto, o contêiner pequeno introduzido no sistema permanecerá no sistema indefinidamente. Ou seja, ele é imediatamente preenchido e esvaziado em cada extremidade e nunca se aglutina com contêineres adjacentes para criar contêineres maiores, como na Figura 133(b) . Esse cenário foi descoberto quando as primeiras implementações do TCP se viram regularmente preenchendo a rede com pequenos segmentos.

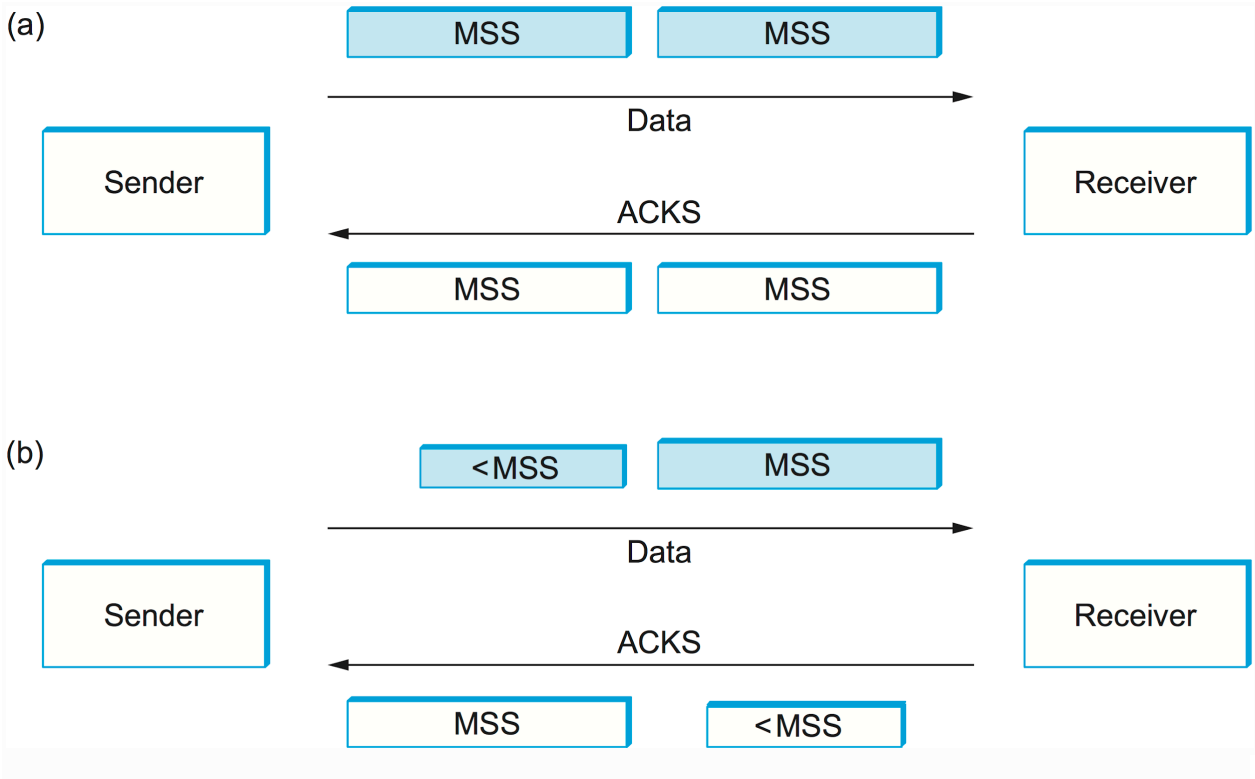

Figura 133. *Síndrome da janela boba. (a) Enquanto o remetente envia segmentos do tamanho de um MSS e o receptor confirma um MSS por vez, o sistema funciona perfeitamente. (b) Assim que o remetente envia menos de um MSS, ou o receptor confirma menos de um MSS, um pequeno "contêiner" entra no sistema e continua a circular.*

Observe que a síndrome da janela boba só é um problema quando o remetente transmite um pequeno segmento ou o destinatário abre a janela um pouco. Se nenhuma dessas situações ocorrer, o pequeno contêiner nunca será introduzido no fluxo. Não é possível proibir o envio de pequenos segmentos; por exemplo, a aplicação pode executar um *push* após enviar um único byte. É possível, no entanto, impedir que o destinatário introduza um pequeno contêiner (ou seja, uma pequena janela aberta). A regra é que, após anunciar uma janela zero, o destinatário deve aguardar um espaço igual a an `MSS`antes de anunciar uma janela aberta.

Como não podemos eliminar a possibilidade de um pequeno contêiner ser introduzido no fluxo, também precisamos de mecanismos para uni-los. O receptor pode fazer isso atrasando ACKs — enviando um ACK combinado em vez de vários menores —, mas

esta é apenas uma solução parcial, pois o receptor não tem como saber por quanto tempo é seguro atrasar, esperando a chegada de outro segmento ou a leitura de mais dados pelo aplicativo (abrindo assim a janela). A solução final cabe ao remetente, o que nos traz de volta à nossa questão original: quando o remetente TCP decide transmitir um segmento?

Algoritmo de Nagle

Voltando ao remetente TCP, se houver dados para enviar, mas a janela estiver aberta há menos de `MSS`, talvez seja necessário aguardar algum tempo antes de enviar os dados disponíveis. Mas a questão é: quanto tempo? Se esperarmos demais, prejudicamos aplicativos interativos como o Telnet. Se não esperarmos o suficiente, corremos o risco de enviar um monte de pacotes minúsculos e cair na síndrome da janela boba. A solução é introduzir um temporizador e transmitir quando o temporizador expirar.

Embora pudéssemos usar um temporizador baseado em relógio — por exemplo, um que dispara a cada 100 ms — Nagle introduziu uma solução elegante *de autocronometragem* . A ideia é que, enquanto o TCP tiver dados em trânsito, o remetente eventualmente receberá um ACK. Esse ACK pode ser tratado como um disparo de temporizador, acionando a transmissão de mais dados. O algoritmo de Nagle fornece uma regra simples e unificada para decidir quando transmitir:

```
When the application produces data to send

    if both the available data and the window >= MSS

        send a full segment

    else

        if there is unACKed data in flight

            buffer the new data until an ACK arrives
```

```
        else

            send all the new data now
```

Em outras palavras, é sempre aceitável enviar um segmento completo se a janela permitir. Também é aceitável enviar imediatamente uma pequena quantidade de dados se não houver segmentos em trânsito, mas se houver algo em andamento, o remetente deve aguardar um ACK antes de transmitir o próximo segmento. Assim, um aplicativo interativo como o Telnet, que grava continuamente um byte por vez, enviará dados à taxa de um segmento por RTT. Alguns segmentos conterão um único byte, enquanto outros conterão tantos bytes quantos o usuário conseguir digitar em um tempo de ida e volta. Como alguns aplicativos não podem se dar ao luxo de tal atraso para cada gravação em uma conexão TCP, a interface do soquete permite que o aplicativo desative o algoritmo de Nagle configurando a `TCP_NODELAY`opção. Definir essa opção significa que os dados são transmitidos o mais rápido possível.

5.2.6 Retransmissão Adaptativa

Como o TCP garante a entrega confiável de dados, ele retransmite cada segmento caso um ACK não seja recebido em um determinado período. O TCP define esse tempo limite como uma função do RTT esperado entre as duas extremidades da conexão. Infelizmente, dada a gama de RTTs possíveis entre qualquer par de hosts na Internet, bem como a variação do RTT entre os mesmos dois hosts ao longo do tempo, escolher um valor de tempo limite apropriado não é tão fácil. Para resolver esse problema, o TCP utiliza um mecanismo de retransmissão adaptativa. A seguir, descreveremos esse mecanismo e como ele evoluiu ao longo do tempo, à medida que a comunidade da Internet adquiriu mais experiência no uso do TCP.

Algoritmo Original

Começamos com um algoritmo simples para calcular um valor de tempo limite entre um par de hosts. Este é o algoritmo originalmente descrito na especificação TCP — e a descrição a seguir o apresenta nesses termos —, mas pode ser usado por qualquer protocolo ponta a ponta.

A ideia é manter uma média contínua do RTT e, em seguida, calcular o tempo limite como uma função desse RTT. Especificamente, sempre que o TCP envia um segmento de dados, ele registra o tempo. Quando um ACK para esse segmento chega, o TCP lê o tempo novamente e, em seguida, considera a diferença entre esses dois tempos como `SampleRTT`. O TCP então calcula `EstimatedRTT`como uma média ponderada entre a estimativa anterior e esta nova amostra. Ou seja,

```
EstimatedRTT = alpha x EstimatedRTT + (1 - alpha) x SampleRTT
```

O parâmetro `alpha`é selecionado para *suavizar* o tempo limite `EstimatedRTT`. Um valor pequeno `alpha`rastreia mudanças no RTT, mas talvez seja muito influenciado por flutuações temporárias. Por outro lado, um valor grande `alpha`é mais estável, mas talvez não seja rápido o suficiente para se adaptar a mudanças reais. A especificação original do TCP recomendava uma configuração `alpha`entre 0,8 e 0,9. O TCP então usa `EstimatedRTT`para calcular o tempo limite de forma bastante conservadora:

```
TimeOut = 2 x EstimatedRTT
```

Algoritmo de Karn/Partridge

Após vários anos de uso na Internet, uma falha bastante óbvia foi descoberta neste algoritmo simples. O problema era que um ACK não reconhece realmente uma transmissão; na verdade, ele reconhece o recebimento de dados. Em outras palavras, sempre que um segmento é retransmitido e um ACK chega ao remetente, é impossível

determinar se esse ACK deve ser associado à primeira ou à segunda transmissão do segmento para fins de medição do RTT da amostra. É necessário saber a qual transmissão associá-lo para calcular um `SampleRTT`. Conforme ilustrado na Figura 134 , se você assumir que o ACK é para a transmissão original, mas na verdade era para a segunda, então o `SampleRTT`é muito grande (a); se você assumir que o ACK é para a segunda transmissão, mas na verdade era para a primeira, então o `SampleRTT`é muito pequeno (b).

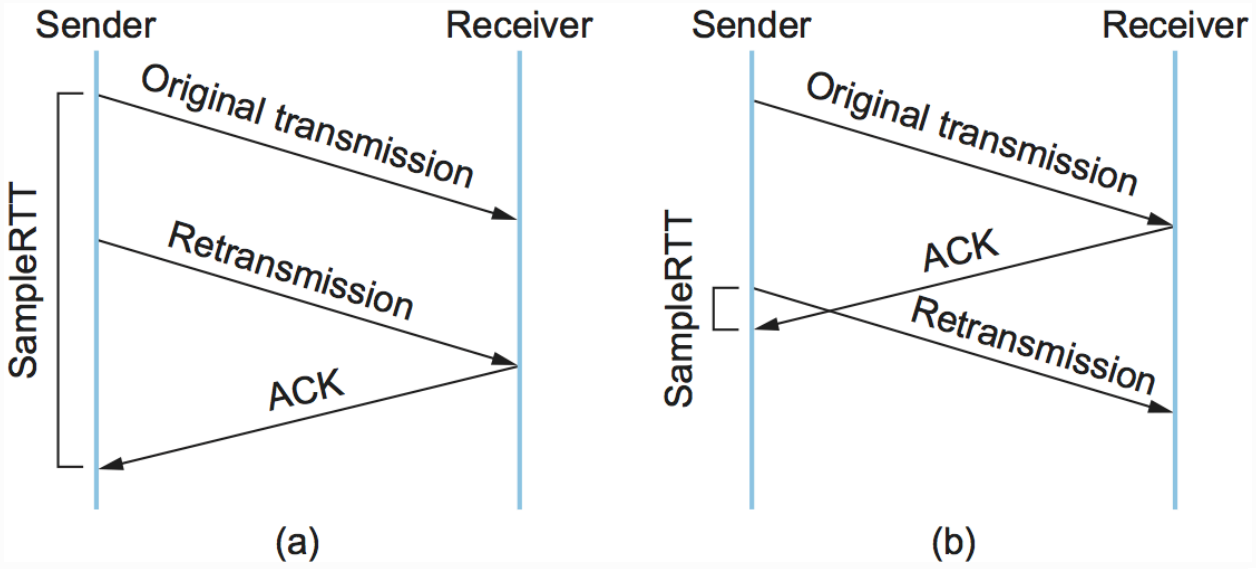

Figura 134. *Associando o ACK com (a) transmissão original versus (b) retransmissão.*

A solução, proposta em 1987, é surpreendentemente simples. Sempre que o TCP retransmite um segmento, ele para de coletar amostras do RTT; ele mede apenas `SampleRTT`segmentos que foram enviados apenas uma vez. Essa solução é conhecida como algoritmo de Karn/Partridge, em homenagem aos seus inventores. A correção proposta também inclui uma segunda pequena alteração no mecanismo de timeout do TCP. Cada vez que o TCP retransmite, ele define o próximo timeout como o dobro do último timeout, em vez de baseá-lo no último `EstimatedRTT`. Ou seja, Karn e Partridge propuseram que o TCP usasse o backoff exponencial, semelhante ao que a Ethernet faz. A motivação para usar o backoff exponencial é simples: o congestionamento é a causa mais provável da perda de segmentos, o que significa que a fonte TCP não deve

reagir de forma muito agressiva a um timeout. De fato, quanto mais vezes a conexão expira, mais cautelosa a fonte deve se tornar. Veremos essa ideia novamente, incorporada em um mecanismo muito mais sofisticado, no próximo capítulo.

Algoritmo de Jacobson/Karels

O algoritmo Karn/Partridge foi introduzido em uma época em que a internet sofria com altos níveis de congestionamento de rede. Sua abordagem foi projetada para corrigir algumas das causas desse congestionamento, mas, embora representasse uma melhoria, o congestionamento não foi eliminado. No ano seguinte (1988), dois outros pesquisadores — Jacobson e Karels — propuseram uma mudança mais drástica no TCP para combater o congestionamento. A maior parte dessa mudança proposta é descrita no próximo capítulo. Aqui, focamos no aspecto dessa proposta relacionado à decisão de quando expirar o tempo limite e retransmitir um segmento.

A propósito, deve ficar claro como o mecanismo de timeout se relaciona com o congestionamento — se o timeout for atingido muito cedo, você poderá retransmitir um segmento desnecessariamente, o que só aumenta a carga na rede. A outra razão para a necessidade de um valor de timeout preciso é que um timeout é considerado como implicando congestionamento, o que aciona um mecanismo de controle de congestionamento. Por fim, observe que não há nada no cálculo de timeout de Jacobson/Karel que seja específico do TCP. Ele poderia ser usado por qualquer protocolo ponta a ponta.

O principal problema com o cálculo original é que ele não leva em consideração a variância dos RTTs da amostra. Intuitivamente, se a variação entre as amostras for pequena, então a estimativa `EstimatedRTT`pode ser mais confiável e não há razão para multiplicar essa estimativa por 2 para calcular o tempo limite. Por outro lado, uma grande variância nas amostras sugere que o valor do tempo limite não deve ser muito fortemente acoplado ao `EstimatedRTT`.

Na nova abordagem, o remetente mede uma nova `SampleRTT`amostra, como antes. Em seguida, ele a incorpora ao cálculo de tempo limite da seguinte forma:

```
Difference = SampleRTT - EstimatedRTT

EstimatedRTT = EstimatedRTT + ( delta x Difference)

Deviation = Deviation + delta (|Difference| - Deviation)
```

onde `delta`está entre 0 e 1. Ou seja, calculamos tanto o RTT médio quanto a variação dessa média.

O TCP então calcula o valor do tempo limite como uma função de `EstimatedRTT`e `Deviation`da seguinte forma:

```
TimeOut = mu x EstimatedRTT + phi x Deviation
```

onde, com base na experiência, `mu`normalmente é definido como 1 e `phi`é definido como 4. Assim, quando a variância é pequena, `TimeOut`é próximo de `EstimatedRTT`; uma grande variância faz com que o `Deviation`termo domine o cálculo.

Implementação

Há dois itens a serem observados sobre a implementação de timeouts no TCP. O primeiro é que é possível implementar o cálculo para `EstimatedRTT`e `Deviation`sem usar aritmética de ponto flutuante. Em vez disso, todo o cálculo é escalado por 2^n, com delta selecionado para ser $1/2^n$. Isso nos permite fazer aritmética de inteiros, implementando multiplicação e divisão usando deslocamentos, obtendo assim maior desempenho. O cálculo resultante é dado pelo seguinte fragmento de código, onde n = 3 (ou seja,). Observe que e são armazenados em suas formas escalonadas, enquanto

o valor de no início do código e de no final são valores reais, não escalonados. Se você achar o código difícil de seguir, você pode querer tentar inserir alguns números reais nele e verificar se ele dá os mesmos resultados que as equações acima.`delta = 1/8EstimatedRTTDeviationSampleRTTTimeOut`

```
{

    SampleRTT -= (EstimatedRTT >> 3);

    EstimatedRTT += SampleRTT;

    if (SampleRTT < 0)

        SampleRTT = -SampleRTT;

    SampleRTT -= (Deviation >> 3);

    Deviation += SampleRTT;

    TimeOut = (EstimatedRTT >> 3) + (Deviation >> 1);

}
```

O segundo ponto a ser observado é que o algoritmo de Jacobson/Karels é tão bom quanto o relógio usado para ler a hora atual. Em implementações típicas do Unix na época, a granularidade do relógio era de até 500 ms, o que é significativamente maior do que o RTT médio entre países, algo entre 100 e 200 ms. Para piorar a situação, a implementação do TCP no Unix apenas verificava se um timeout deveria ocorrer toda vez que esse relógio de 500 ms disparasse e coletava uma amostra do tempo de ida e volta apenas uma vez por RTT. A combinação desses dois fatores poderia significar que um timeout ocorreria 1 segundo após a transmissão do segmento. Mais uma vez, as extensões do TCP incluem um mecanismo que torna esse cálculo do RTT um pouco mais preciso.

Todos os algoritmos de retransmissão que discutimos são baseados em tempos limite de confirmação, que indicam que um segmento provavelmente foi perdido. Observe que um tempo limite, no entanto, não informa ao remetente se quaisquer segmentos enviados após o segmento perdido foram recebidos com sucesso. Isso ocorre porque as confirmações TCP são cumulativas; elas identificam apenas o último segmento recebido sem nenhuma lacuna anterior. A recepção de segmentos que ocorrem após uma lacuna torna-se mais frequente à medida que redes mais rápidas levam a janelas maiores. Se os ACKs também informassem ao remetente quais segmentos subsequentes, se houver, foram recebidos, o remetente poderia ser mais inteligente sobre quais segmentos retransmite, tirar conclusões mais precisas sobre o estado de congestionamento e fazer estimativas mais precisas de RTT. Uma extensão TCP que suporta isso é descrita em uma seção posterior.

Conclusão principal

Há outro ponto a ser destacado sobre o cálculo de timeouts. É um assunto surpreendentemente complexo, tanto que existe uma RFC inteira dedicada ao tópico: a RFC 6298. A conclusão é que, às vezes, especificar completamente um protocolo envolve tantas minúcias que a linha entre especificação e implementação se torna tênue. Isso já aconteceu mais de uma vez com o TCP, levando alguns a argumentarem que "a implementação **é** a especificação". Mas isso não é necessariamente algo ruim, desde que a implementação de referência esteja disponível como software de código aberto. De forma mais geral, a indústria está vendo o software de código aberto ganhar importância à medida que os padrões abertos perdem importância. [Próximo]

5.2.7 Limites de registro

Como o TCP é um protocolo de fluxo de bytes, o número de bytes gravados pelo remetente não é necessariamente igual ao número de bytes lidos pelo destinatário. Por exemplo, o aplicativo pode gravar 8 bytes, depois 2 bytes e, em seguida, 20 bytes em uma conexão TCP, enquanto no lado receptor o aplicativo lê 5 bytes por vez dentro de

um loop que itera 6 vezes. O TCP não insere limites de registro entre o 8º e o 9º bytes, nem entre o 10º e o 11º bytes. Isso contrasta com um protocolo orientado a mensagens, como o UDP, no qual a mensagem enviada tem exatamente o mesmo comprimento da mensagem recebida.

Embora o TCP seja um protocolo de fluxo de bytes, ele possui dois recursos diferentes que podem ser usados pelo remetente para inserir limites de registro nesse fluxo de bytes, informando assim ao destinatário como dividir o fluxo de bytes em registros. (A capacidade de marcar limites de registro é útil, por exemplo, em muitas aplicações de banco de dados.) Ambos os recursos foram originalmente incluídos no TCP por razões completamente diferentes; eles só passaram a ser usados para esse propósito com o tempo.

O primeiro mecanismo é o recurso de dados urgentes, conforme implementado pelo `URG`sinalizador e pelo `UrgPtr`campo no cabeçalho TCP. Originalmente, o mecanismo de dados urgentes foi projetado para permitir que o aplicativo emissor enviasse dados *fora de banda* para seu par. Por "fora de banda" entendemos dados separados do fluxo normal de dados (por exemplo, um comando para interromper uma operação já em andamento). Esses dados fora de banda eram identificados no segmento usando o `UrgPtr`campo e deveriam ser entregues ao processo receptor assim que chegassem, mesmo que isso significasse entregá-los antes dos dados com um número de sequência anterior. Com o tempo, no entanto, esse recurso não foi mais utilizado, então, em vez de significar dados "urgentes", passou a ser usado para significar dados "especiais", como um marcador de registro. Esse uso se desenvolveu porque, assim como na operação push, o TCP do lado receptor deve informar ao aplicativo que dados urgentes chegaram. Ou seja, os dados urgentes em si não são importantes. O importante é que o processo emissor possa efetivamente enviar um sinal ao receptor.

O segundo mecanismo para inserir marcadores de fim de registro em um byte é a operação *push* . Originalmente, esse mecanismo foi projetado para permitir que o processo de envio informe ao TCP que ele deve enviar (liberar) quaisquer bytes coletados para seu par. A operação *push* pode ser usada para implementar limites de

registro, pois a especificação determina que o TCP deve enviar quaisquer dados que tenha armazenado em buffer na origem quando o aplicativo solicitar push e, opcionalmente, o TCP no destino notifica o aplicativo sempre que um segmento de entrada tiver o sinalizador PUSH definido. Se o lado receptor suportar essa opção (a interface do soquete não), a operação push pode ser usada para dividir o fluxo TCP em registros.

É claro que o programa aplicativo é sempre livre para inserir limites de registro sem qualquer assistência do TCP. Por exemplo, ele pode enviar um campo que indica o comprimento de um registro a seguir ou pode inserir seus próprios marcadores de limite de registro no fluxo de dados.

5.2.8 Extensões TCP

Mencionamos em quatro pontos diferentes nesta seção que agora existem extensões para o TCP que ajudam a mitigar alguns problemas que o TCP enfrentou à medida que a rede subjacente se tornou mais rápida. Essas extensões são projetadas para ter o menor impacto possível no TCP. Em particular, elas são realizadas como opções que podem ser adicionadas ao cabeçalho TCP. (Já abordamos esse ponto anteriormente, mas a razão pela qual o cabeçalho TCP tem um `HdrLen`campo é que o cabeçalho pode ter comprimento variável; a parte variável do cabeçalho TCP contém as opções que foram adicionadas.) A importância de adicionar essas extensões como opções, em vez de alterar o núcleo do cabeçalho TCP, é que os hosts ainda podem se comunicar usando TCP mesmo que não implementem as opções. Os hosts que implementam as extensões opcionais, no entanto, podem tirar proveito delas. Os dois lados concordam que usarão as opções durante a fase de estabelecimento da conexão TCP.

A primeira extensão ajuda a aprimorar o mecanismo de tempo limite do TCP. Em vez de medir o RTT usando um evento de granularidade grossa, o TCP pode ler o relógio real do sistema quando estiver prestes a enviar um segmento e inserir esse horário — pense nele como um *carimbo de data/hora* de 32 bits — no cabeçalho do segmento. O receptor então ecoa esse carimbo de data/hora de volta para o remetente em sua

confirmação, e o remetente subtrai esse carimbo de data/hora do horário atual para medir o RTT. Em essência, a opção de carimbo de data/hora fornece um local conveniente para o TCP armazenar o registro de quando um segmento foi transmitido; ela armazena o horário no próprio segmento. Observe que os pontos finais da conexão não precisam de relógios sincronizados, pois o carimbo de data/hora é gravado e lido na mesma extremidade da conexão.

A segunda extensão aborda o problema do campo de 32 bits do TCP `SequenceNum`se enrolar muito cedo em uma rede de alta velocidade. Em vez de definir um novo campo de número de sequência de 64 bits, o TCP usa o carimbo de tempo de 32 bits que acabamos de descrever para estender efetivamente o espaço do número de sequência. Em outras palavras, o TCP decide se aceita ou rejeita um segmento com base em um identificador de 64 bits que possui o `SequenceNum`campo nos 32 bits de ordem inferior e o carimbo de tempo nos 32 bits de ordem superior. Como o carimbo de tempo é sempre crescente, ele serve para distinguir entre duas encarnações diferentes do mesmo número de sequência. Observe que o carimbo de tempo está sendo usado nesta configuração apenas para proteger contra o enrolamento; ele não é tratado como parte do número de sequência para fins de ordenação ou reconhecimento de dados.

A terceira extensão permite que o TCP anuncie uma janela maior, permitindo assim que ele preencha canais maiores com atraso × largura de banda, possibilitados por redes de alta velocidade. Essa extensão envolve uma opção que define um *fator de escala* para a janela anunciada. Ou seja, em vez de interpretar o número que aparece no `AdvertisedWindow` campo como uma indicação de quantos bytes o remetente pode ter sem confirmação, essa opção permite que os dois lados do TCP concordem que o `AdvertisedWindow`campo conta pedaços maiores (por exemplo, quantas unidades de dados de 16 bytes o remetente pode ter sem confirmação). Em outras palavras, a opção de escala de janela especifica quantos bits cada lado deve deslocar o `AdvertisedWindow`campo para a esquerda antes de usar seu conteúdo para calcular uma janela efetiva.

A quarta extensão permite que o TCP complemente sua confirmação cumulativa com confirmações seletivas de quaisquer segmentos adicionais que tenham sido recebidos, mas não sejam contíguos a todos os segmentos recebidos anteriormente. Esta é a opção *de confirmação seletiva* , ou *SACK* . Quando a opção SACK é usada, o receptor continua a confirmar os segmentos normalmente — o significado do `Acknowledge`campo não muda —, mas também usa campos opcionais no cabeçalho para confirmar quaisquer blocos adicionais de dados recebidos. Isso permite que o remetente retransmita apenas os segmentos que estão faltando, de acordo com a confirmação seletiva.

Sem o SACK, existem apenas duas estratégias razoáveis para um remetente. A estratégia pessimista responde a um timeout retransmitindo não apenas o segmento que expirou, mas quaisquer segmentos transmitidos posteriormente. Na prática, a estratégia pessimista assume o pior: que todos esses segmentos foram perdidos. A desvantagem da estratégia pessimista é que ela pode retransmitir desnecessariamente segmentos que foram recebidos com sucesso na primeira vez. A outra estratégia é a estratégia otimista, que responde a um timeout retransmitindo apenas o segmento que expirou. Na prática, a abordagem otimista assume o cenário mais otimista: que apenas um segmento foi perdido. A desvantagem da estratégia otimista é que ela é muito lenta, desnecessariamente, quando uma série de segmentos consecutivos é perdida, como pode acontecer em caso de congestionamento. Ela é lenta porque a perda de cada segmento não é descoberta até que o remetente receba um ACK para a retransmissão do segmento anterior. Portanto, ela consome um RTT por segmento até retransmitir todos os segmentos da série perdida. Com a opção SACK, uma estratégia melhor está disponível para o remetente: retransmitir apenas os segmentos que preenchem as lacunas entre os segmentos que foram reconhecidos seletivamente.

A propósito, essas extensões não são a história completa. Veremos mais algumas extensões no próximo capítulo, quando analisarmos como o TCP lida com o congestionamento. A Autoridade para Atribuição de Números da Internet (IANA) monitora todas as opções definidas para o TCP (e para muitos outros protocolos da

internet). Consulte as referências no final do capítulo para obter um link para o registro de números de protocolo da IANA.

5.2.9 Desempenho

Lembre-se de que o Capítulo 1 apresentou as duas métricas quantitativas pelas quais o desempenho da rede é avaliado: latência e taxa de transferência. Como mencionado naquela discussão, essas métricas são influenciadas não apenas pelo hardware subjacente (por exemplo, atraso de propagação e largura de banda do link), mas também por sobrecargas de software. Agora que temos um gráfico completo de protocolos baseados em software disponível, que inclui protocolos de transporte alternativos, podemos discutir como medir seu desempenho de forma significativa. A importância dessas medições é que elas representam o desempenho observado por programas aplicativos.

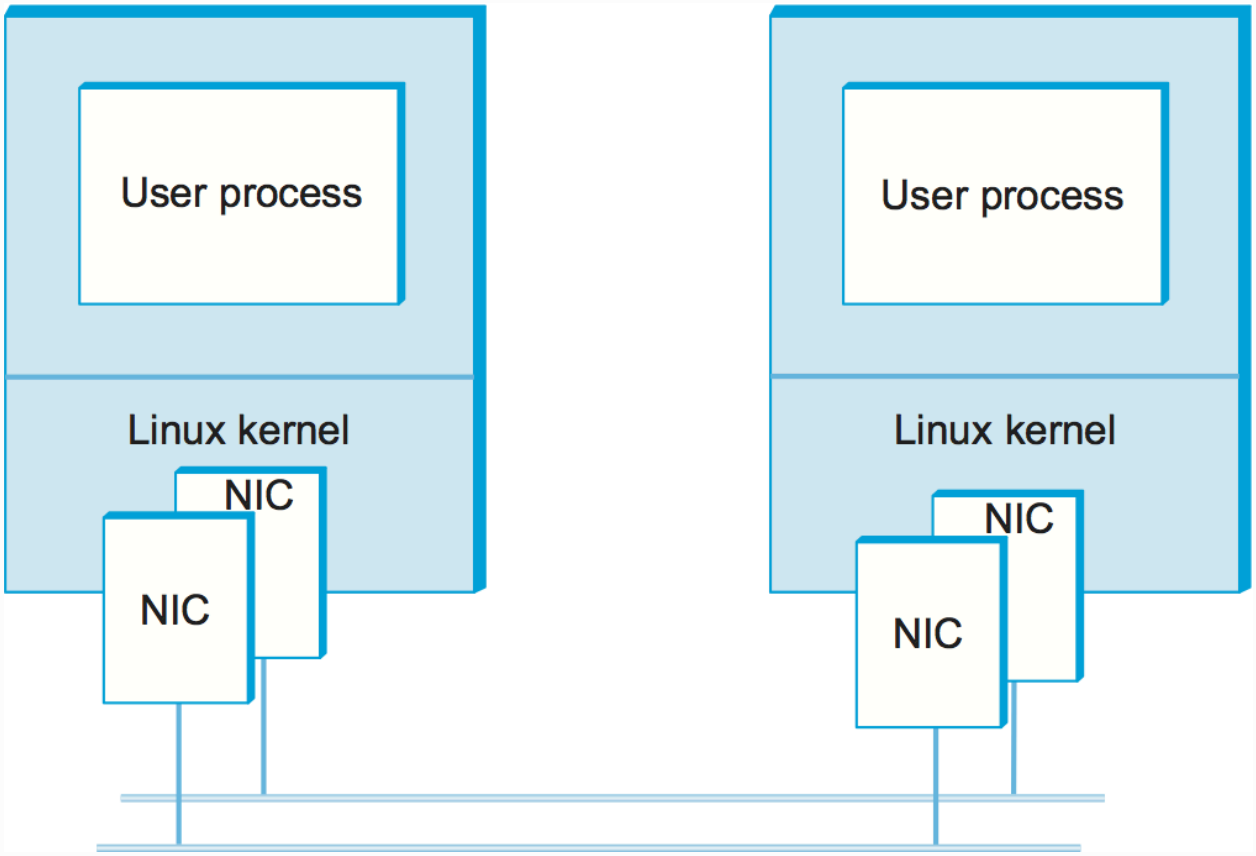

Figura 135. *Sistema medido: Duas estações de trabalho Linux e um par de links Ethernet Gbps.*

Começamos, como qualquer relatório de resultados experimentais deve, descrevendo nosso método experimental. Isso inclui o aparato utilizado nos experimentos; neste caso, cada estação de trabalho possui um par de processadores Xeon de 2,4 GHz com CPU dupla e Linux. Para permitir velocidades acima de 1 Gbps, um par de adaptadores Ethernet (denominados NIC, para placa de interface de rede) é usado em cada máquina. A Ethernet abrange uma única sala de máquinas, portanto, a propagação não é um problema, tornando-se uma medida da sobrecarga do processador/software. Um programa de teste em execução na interface do soquete simplesmente tenta transferir dados o mais rápido possível de uma máquina para a outra. A Figura 135 ilustra a configuração.

Você pode notar que esta configuração experimental não é especialmente inovadora em termos de hardware ou velocidade de link. O objetivo desta seção não é mostrar a velocidade de execução de um protocolo específico, mas ilustrar a metodologia geral para medir e relatar o desempenho do protocolo.

O teste de throughput é realizado para uma variedade de tamanhos de mensagens usando uma ferramenta de benchmarking padrão chamada TTCP. Os resultados do teste de throughput são apresentados na Figura 136. O principal ponto a ser observado neste gráfico é que o throughput melhora à medida que as mensagens aumentam de tamanho. Isso faz sentido — cada mensagem envolve uma certa quantidade de overhead, portanto, uma mensagem maior significa que esse overhead é amortizado em mais bytes. A curva de throughput se achata acima de 1 KB, ponto em que o overhead por mensagem se torna insignificante quando comparado ao grande número de bytes que a pilha de protocolos precisa processar.

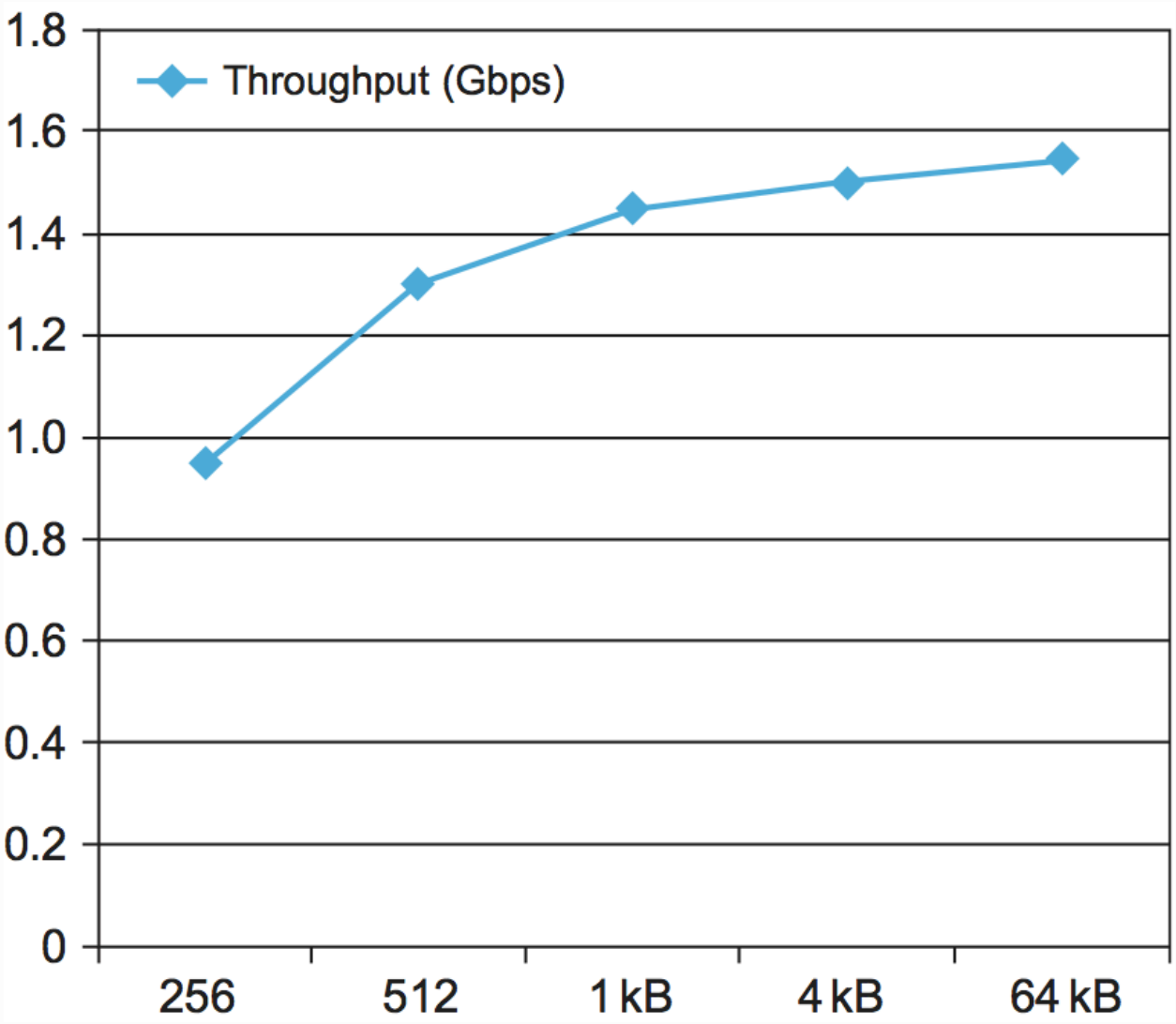

Figura 136. *Taxa de transferência medida usando TCP, para vários tamanhos de mensagem.*

Vale ressaltar que a taxa de transferência máxima é inferior a 2 Gbps, a velocidade de link disponível nesta configuração. Testes e análises adicionais dos resultados seriam necessários para descobrir onde está o gargalo (ou se há mais de um). Por exemplo, analisar a carga da CPU pode indicar se a CPU é o gargalo ou se a largura de banda da memória, o desempenho do adaptador ou algum outro problema são os culpados.

Observamos também que a rede neste teste é basicamente "perfeita". Ela praticamente não apresenta atrasos ou perdas, portanto, os únicos fatores que afetam o

desempenho são a implementação do TCP e o hardware e software da estação de trabalho. Em contraste, na maioria das vezes lidamos com redes que estão longe de ser perfeitas, notadamente nossos enlaces de última milha com largura de banda limitada e enlaces sem fio propensos a perdas. Antes de podermos avaliar completamente como esses enlaces afetam o desempenho do TCP, precisamos entender como o TCP lida com *o congestionamento* , que é o tópico do próximo capítulo.

Em vários momentos da história das redes, a velocidade cada vez maior dos links de rede ameaçou ultrapassar o que poderia ser entregue às aplicações. Por exemplo, um grande esforço de pesquisa foi iniciado nos Estados Unidos em 1989 para construir "redes gigabit", cujo objetivo não era apenas construir links e switches que pudessem operar a 1 Gbps ou mais, mas também fornecer essa taxa de transferência para um único processo de aplicação. Havia alguns problemas reais (por exemplo, adaptadores de rede, arquiteturas de estações de trabalho e sistemas operacionais tiveram que ser projetados com a taxa de transferência da rede para o aplicativo em mente) e também alguns problemas percebidos que acabaram não sendo tão sérios. No topo da lista desses problemas estava a preocupação de que os protocolos de transporte existentes, em particular o TCP, pudessem não estar à altura do desafio da operação em gigabit.

Como se constata, o TCP tem se saído bem ao acompanhar as crescentes demandas de redes e aplicações de alta velocidade. Um dos fatores mais importantes foi a introdução do escalonamento de janelas para lidar com produtos de largura de banda-atraso maiores. No entanto, muitas vezes há uma grande diferença entre o desempenho teórico do TCP e o que é alcançado na prática. Problemas relativamente simples, como copiar os dados mais vezes do que o necessário à medida que passam do adaptador de rede para a aplicação, podem reduzir o desempenho, assim como a insuficiência de memória buffer quando o produto largura de banda-atraso é grande. E a dinâmica do TCP é complexa o suficiente (como ficará ainda mais evidente no próximo capítulo) para que interações sutis entre o comportamento da rede, o

comportamento da aplicação e o próprio protocolo TCP possam alterar drasticamente o desempenho.

Para os nossos propósitos, vale a pena notar que o TCP continua a apresentar um desempenho muito bom à medida que a velocidade da rede aumenta e, quando se depara com algum limite (normalmente relacionado a congestionamento, aumento da largura de banda e atraso, ou ambos), os pesquisadores correm para encontrar soluções. Vimos algumas delas neste capítulo e veremos mais no próximo.

5.2.10 Opções alternativas de projeto (SCTP, QUIC)

Embora o TCP tenha se mostrado um protocolo robusto que atende às necessidades de uma ampla gama de aplicações, o espaço de projeto para protocolos de transporte é bastante amplo. O TCP não é, de forma alguma, o único ponto válido nesse espaço. Concluímos nossa discussão sobre o TCP considerando opções alternativas de projeto. Embora ofereçamos uma explicação para o motivo pelo qual os projetistas do TCP fizeram as escolhas que fizeram, observamos que existem outros protocolos que fizeram outras escolhas, e mais protocolos semelhantes poderão surgir no futuro.

Em primeiro lugar, sugerimos, desde o primeiro capítulo deste livro, que existem pelo menos duas classes interessantes de protocolos de transporte: protocolos orientados a fluxo, como o TCP, e protocolos de solicitação/resposta, como o RPC. Em outras palavras, dividimos implicitamente o espaço de projeto pela metade e colocamos o TCP diretamente na metade do mundo orientada a fluxo. Poderíamos ainda dividir os protocolos orientados a fluxo em dois grupos — confiáveis e não confiáveis — com o primeiro contendo o TCP e o último sendo mais adequado para aplicações de vídeo interativas que preferem perder um quadro a incorrer no atraso associado a uma retransmissão.

Este exercício de construção de uma taxonomia de protocolos de transporte é interessante e poderia ser continuado com mais e mais detalhes, mas o mundo não é

tão preto no branco quanto gostaríamos. Considere a adequação do TCP como um protocolo de transporte para aplicações de solicitação/resposta, por exemplo. O TCP é um protocolo full-duplex, portanto, seria fácil abrir uma conexão TCP entre o cliente e o servidor, enviar a mensagem de solicitação em uma direção e enviar a mensagem de resposta na outra. Há duas complicações, no entanto. A primeira é que o TCP é um protocolo orientado *a bytes* , em vez de um protocolo orientado *a mensagens* , e as aplicações de solicitação/resposta sempre lidam com mensagens. (Exploraremos a questão de bytes versus mensagens com mais detalhes em breve.) A segunda complicação é que, nas situações em que tanto a mensagem de solicitação quanto a mensagem de resposta cabem em um único pacote de rede, um protocolo de solicitação/resposta bem projetado precisa de apenas dois pacotes para implementar a troca, enquanto o TCP precisaria de pelo menos nove: três para estabelecer a conexão, dois para a troca de mensagens e quatro para interromper a conexão. É claro que, se as mensagens de solicitação ou resposta forem grandes o suficiente para exigir vários pacotes de rede (por exemplo, podem ser necessários 100 pacotes para enviar uma mensagem de resposta de 100.000 bytes), a sobrecarga de configurar e desconectar a conexão é irrelevante. Em outras palavras, nem sempre um protocolo específico não pode suportar uma determinada funcionalidade; às vezes, um projeto é mais eficiente do que outro em circunstâncias específicas.

Em segundo lugar, como sugerido, você pode questionar por que o TCP optou por fornecer um serviço confiável de fluxo *de bytes* em vez de um serviço confiável de fluxo *de mensagens* ; mensagens seriam a escolha natural para um aplicativo de banco de dados que deseja trocar registros. Há duas respostas para essa pergunta. A primeira é que um protocolo orientado a mensagens deve, por definição, estabelecer um limite superior para o tamanho das mensagens. Afinal, uma mensagem infinitamente longa é um fluxo de bytes. Para qualquer tamanho de mensagem selecionado por um protocolo, haverá aplicativos que desejarão enviar mensagens maiores, tornando o protocolo de transporte inútil e forçando o aplicativo a implementar seus próprios serviços semelhantes aos de transporte. A segunda razão é que, embora os protocolos orientados a mensagens sejam definitivamente mais apropriados para aplicativos que

desejam enviar registros entre si, você pode facilmente inserir limites de registro em um fluxo de bytes para implementar essa funcionalidade.

Uma terceira decisão tomada no projeto do TCP é que ele entrega bytes *em ordem* para a aplicação. Isso significa que ele pode reter bytes que foram recebidos fora de ordem da rede, aguardando que alguns bytes faltantes preencham uma lacuna. Isso é extremamente útil para muitas aplicações, mas acaba sendo bastante inútil se a aplicação for capaz de processar dados fora de ordem. Como um exemplo simples, uma página da Web contendo vários objetos incorporados não precisa que todos os objetos sejam entregues em ordem antes de começar a exibir a página. De fato, há uma classe de aplicações que prefere lidar com dados fora de ordem na camada de aplicação, em troca de obter dados mais cedo quando os pacotes são descartados ou mal ordenados dentro da rede. O desejo de suportar tais aplicações levou à criação não de um, mas de dois protocolos de transporte padrão IETF. O primeiro deles foi o SCTP, o *Protocolo de Transmissão de Controle de Fluxo* . O SCTP fornece um serviço de entrega parcialmente ordenado, em vez do serviço estritamente ordenado do TCP. (O SCTP também toma algumas outras decisões de design que diferem do TCP, incluindo a orientação das mensagens e o suporte a múltiplos endereços IP para uma única sessão.) Mais recentemente, a IETF vem padronizando um protocolo otimizado para tráfego web, conhecido como QUIC. Falaremos mais sobre QUIC em breve.

Em quarto lugar, o TCP optou por implementar fases explícitas de configuração/desconexão, mas isso não é necessário. No caso de configuração de conexão, seria possível enviar todos os parâmetros de conexão necessários junto com a primeira mensagem de dados. O TCP optou por uma abordagem mais conservadora, que dá ao receptor a oportunidade de rejeitar a conexão antes que qualquer dado chegue. No caso de desconexão, poderíamos encerrar silenciosamente uma conexão que esteve inativa por um longo período, mas isso complicaria aplicações como login remoto, que desejam manter uma conexão ativa por semanas seguidas; tais aplicações seriam forçadas a enviar mensagens "keep alive" fora de banda para evitar que o estado da conexão na outra extremidade desapareça.

Por fim, o TCP é um protocolo baseado em janelas, mas esta não é a única possibilidade. A alternativa é um projeto *baseado em taxa* , no qual o receptor informa ao remetente a taxa — expressa em bytes ou pacotes por segundo — na qual está disposto a aceitar os dados recebidos. Por exemplo, o receptor pode informar ao remetente que pode acomodar 100 pacotes por segundo. Há uma dualidade interessante entre janelas e taxa, uma vez que o número de pacotes (bytes) na janela, dividido pelo RTT, é exatamente a taxa. Por exemplo, um tamanho de janela de 10 pacotes e um RTT de 100 ms implica que o remetente tem permissão para transmitir a uma taxa de 100 pacotes por segundo. É aumentando ou diminuindo o tamanho da janela anunciado que o receptor está efetivamente aumentando ou diminuindo a taxa na qual o remetente pode transmitir. No TCP, essa informação é realimentada ao remetente no `AdvertisedWindow`campo do ACK para cada segmento. Uma das principais questões em um protocolo baseado em taxa é a frequência com que a taxa desejada — que pode mudar ao longo do tempo — é retransmitida de volta à fonte: é para cada pacote, uma vez por RTT ou apenas quando a taxa muda? Embora tenhamos acabado de considerar a relação entre janela e taxa no contexto do controle de fluxo, trata-se de uma questão ainda mais controversa no contexto do controle de congestionamento, que discutiremos no próximo capítulo.

RÁPIDO

O QUIC teve origem no Google em 2012 e foi posteriormente desenvolvido como um padrão proposto no IETF. Ao contrário de muitos outros esforços para adicionar ao conjunto de protocolos de transporte na Internet, o QUIC alcançou ampla implantação. Conforme discutido mais adiante no Capítulo 9 , o QUIC foi motivado em grande parte pelos desafios de combinar a semântica de solicitação/resposta do HTTP com a natureza orientada a fluxo do TCP. Essas questões se tornaram mais perceptíveis ao longo do tempo, devido a fatores como o aumento de redes sem fio de alta latência, a disponibilidade de múltiplas redes para um único dispositivo (por exemplo, Wi-Fi e celular) e o uso crescente de conexões criptografadas e autenticadas na Web (conforme discutido no Capítulo 8). Embora uma descrição completa do QUIC esteja

além do nosso escopo, algumas das principais decisões de projeto valem a pena discutir.

TCP multicaminho

> Nem sempre é necessário definir um novo protocolo se você perceber que um protocolo existente não atende adequadamente a um caso de uso específico. Às vezes, é possível fazer mudanças substanciais na forma como um protocolo existente é implementado, mantendo-se fiel à especificação original. O TCP multicaminho é um exemplo dessa situação.
>
> A ideia do TCP Multicaminho é direcionar pacotes por vários caminhos pela internet, por exemplo, usando dois endereços IP diferentes para um dos pontos finais. Isso pode ser especialmente útil ao entregar dados a um dispositivo móvel conectado tanto à rede Wi-Fi quanto à rede celular (e, portanto, com dois endereços IP exclusivos). Por serem sem fio, ambas as redes podem sofrer perdas significativas de pacotes, portanto, poder usar ambas para transportar pacotes pode melhorar drasticamente a experiência do usuário. A chave é que o lado receptor do TCP reconstrua o fluxo de bytes original e ordenado antes de passar os dados para o aplicativo, que permanece sem saber que está sobre o TCP Multicaminho. (Isso contrasta com aplicativos que abrem propositalmente duas ou mais conexões TCP para obter melhor desempenho.)
>
> Por mais simples que o TCP Multipath pareça, é incrivelmente difícil de acertar, pois quebra muitas premissas sobre como o controle de fluxo TCP, a remontagem de segmentos em ordem e o controle de congestionamento são implementados. Deixamos como exercício para o leitor explorar essas sutilezas. Fazer isso é uma ótima maneira de garantir que seu conhecimento básico do TCP seja sólido.

Se a latência da rede for alta — digamos, 100 milissegundos ou mais —, alguns RTTs podem rapidamente se transformar em um incômodo visível para o usuário final.

Estabelecer uma sessão HTTP sobre TCP com Segurança da Camada de Transporte (Seção 8.5) normalmente levaria pelo menos três viagens de ida e volta (uma para estabelecer a sessão TCP e duas para configurar os parâmetros de criptografia) antes que a primeira mensagem HTTP pudesse ser enviada. Os projetistas do QUIC reconheceram que esse atraso — resultado direto de uma abordagem em camadas para o projeto do protocolo — poderia ser drasticamente reduzido se a configuração da conexão e os handshakes de segurança necessários fossem combinados e otimizados para viagens de ida e volta mínimas.

Observe também como a presença de múltiplas interfaces de rede pode afetar o design. Se o seu celular perder a conexão Wi-Fi e precisar alternar para uma conexão celular, isso normalmente exigiria um tempo limite de TCP em uma conexão e uma nova série de handshakes na outra. Tornar a conexão algo que possa persistir em diferentes conexões da camada de rede foi outro objetivo de design do QUIC.

Finalmente, como observado acima, o modelo de fluxo de bytes confiável para TCP é uma combinação ruim para uma solicitação de página da Web, quando muitos objetos precisam ser buscados e a renderização da página pode começar antes que todos eles cheguem. Embora uma solução alternativa para isso seja abrir várias conexões TCP em paralelo, essa abordagem (que foi usada nos primeiros dias da Web) tem seu próprio conjunto de desvantagens, principalmente no controle de congestionamento (consulte o Capítulo 6). Especificamente, cada conexão executa seu próprio loop de controle de congestionamento, de modo que a experiência de congestionamento em uma conexão não é aparente para as outras conexões, e cada conexão tenta descobrir a quantidade apropriada de largura de banda para consumir por conta própria. O QUIC introduziu a ideia de fluxos dentro de uma conexão, para que os objetos pudessem ser entregues fora de ordem, mantendo uma visão holística do congestionamento em todos os fluxos.

Curiosamente, na época em que o QUIC surgiu, muitas decisões de projeto já haviam sido tomadas, apresentando desafios para a implantação de um novo protocolo de transporte. Notavelmente, muitos "middleboxes", como NATs e firewalls (consulte a

Seção 8.5), têm conhecimento suficiente dos protocolos de transporte amplamente difundidos existentes (TCP e UDP) para que não se possa confiar neles para passar um novo protocolo de transporte. Como resultado, o QUIC, na verdade, roda sobre o UDP. Em outras palavras, é um protocolo de transporte rodando sobre um protocolo de transporte. Isso não é tão incomum quanto nosso foco em camadas pode sugerir, como as próximas duas subseções também ilustram. Essa escolha foi feita com o objetivo de tornar o QUIC implantável na Internet como ela existe, e tem sido bastante bem-sucedida.

O QUIC implementa o estabelecimento rápido de conexões com criptografia e autenticação no primeiro RTT. Ele fornece um identificador de conexão que persiste mesmo após mudanças na rede subjacente. Ele suporta a multiplexação de vários fluxos em uma única conexão de transporte, para evitar o bloqueio de linha que pode ocorrer quando um único pacote é descartado enquanto outros dados úteis continuam a chegar. E preserva (e, de certa forma, aprimora) as propriedades de prevenção de congestionamento do TCP, um aspecto importante dos protocolos de transporte ao qual retornaremos no Capítulo 6 .

O HTTP passou por diversas versões (1.0, 1.1, 2.0, discutidas na Seção 9.1) em um esforço para mapear seus requisitos de forma mais clara às capacidades do TCP. Com a chegada do QUIC, o HTTP/3 agora pode aproveitar uma camada de transporte que foi explicitamente projetada para atender aos requisitos de aplicação da Web.

O QUIC é um desenvolvimento muito interessante no mundo dos protocolos de transporte. Muitas das limitações do TCP são conhecidas há décadas, mas o QUIC representa um dos esforços mais bem-sucedidos até o momento para estabelecer um ponto diferente no espaço de design. Como o QUIC foi inspirado pela experiência com HTTP e a Web — que surgiu muito depois do TCP estar consolidado na internet —, ele apresenta um estudo de caso fascinante sobre as consequências imprevistas dos designs em camadas e a evolução da internet.

5.3 Chamada de Procedimento Remoto

Um padrão comum de comunicação usado por programas de aplicação estruturados como um par *cliente/servidor* é a transação de mensagem de solicitação/resposta: um cliente envia uma mensagem de solicitação a um servidor, e o servidor responde com uma mensagem de resposta, com o cliente bloqueando (suspendendo a execução) para aguardar a resposta. A Figura 137 ilustra a interação básica entre o cliente e o servidor em tal troca.

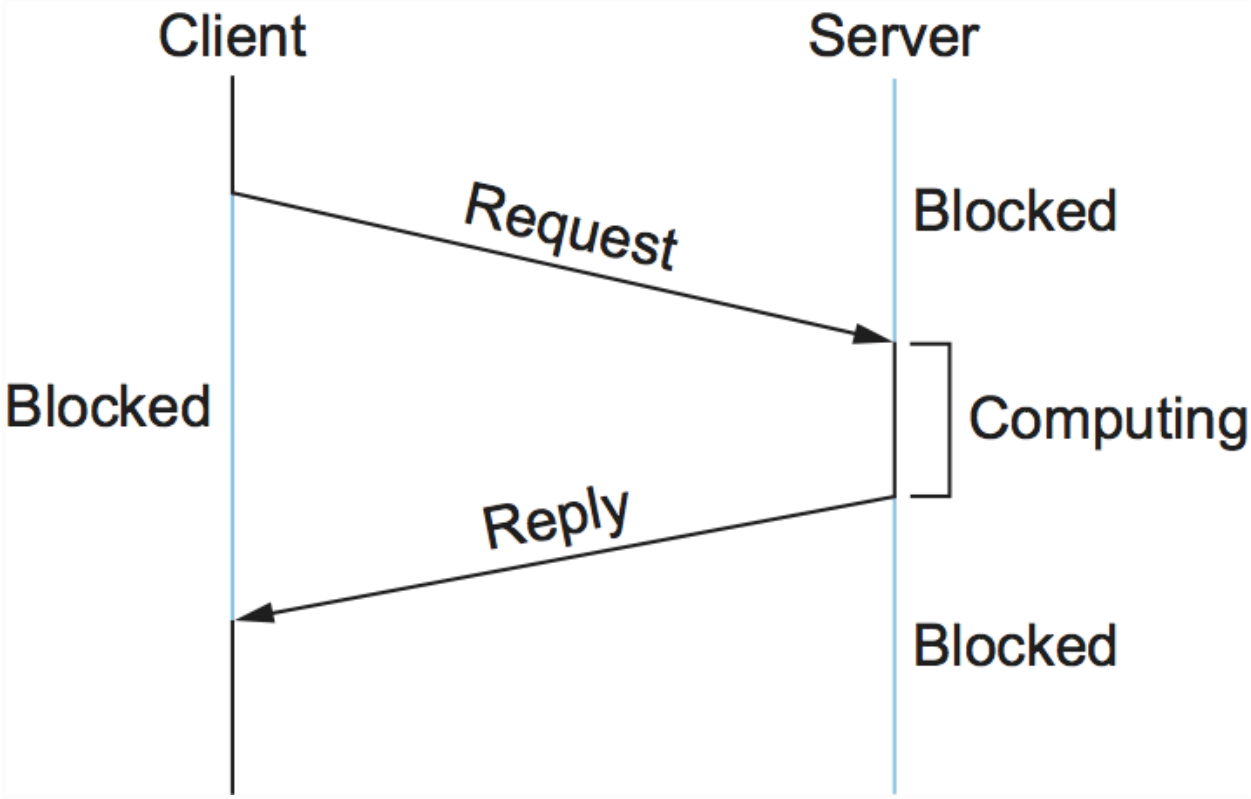

Figura 137. *Linha do tempo para RPC.*

Um protocolo de transporte que suporta o paradigma de solicitação/resposta é muito mais do que uma mensagem UDP em uma direção seguida por uma mensagem UDP na outra. Ele precisa lidar com a identificação correta de processos em hosts remotos e correlacionar solicitações com respostas. Também pode precisar superar algumas ou

todas as limitações da rede subjacente descritas na descrição do problema no início deste capítulo. Embora o TCP supere essas limitações fornecendo um serviço confiável de fluxo de bytes, ele também não se adapta perfeitamente ao paradigma de solicitação/resposta. Esta seção descreve uma terceira categoria de protocolo de transporte, chamada *Chamada de Procedimento Remoto* (RPC), que atende melhor às necessidades de uma aplicação envolvida em uma troca de mensagens de solicitação/resposta.

5.3.1 Fundamentos do RPC

O RPC não é tecnicamente um protocolo — é melhor entendido como um mecanismo geral para estruturar sistemas distribuídos. O RPC é popular porque se baseia na semântica de uma chamada de procedimento local — o programa aplicativo faz uma chamada para um procedimento sem considerar se ele é local ou remoto e bloqueia até que a chamada retorne. Um desenvolvedor de aplicativos pode não ter muita noção se o procedimento é local ou remoto, simplificando consideravelmente sua tarefa. Quando os procedimentos chamados são, na verdade, métodos de objetos remotos em uma linguagem orientada a objetos, o RPC é conhecido como *invocação de método remoto* (RMI). Embora o conceito de RPC seja simples, existem dois problemas principais que o tornam mais complexo do que as chamadas de procedimento locais:

- A rede entre o processo de chamada e o processo chamado possui propriedades muito mais complexas do que o backplane de um computador. Por exemplo, é provável que limite o tamanho das mensagens e tenha tendência a perdê-las e reordená-las.
- Os computadores nos quais os processos de chamada e de chamada são executados podem ter arquiteturas e formatos de representação de dados significativamente diferentes.

Portanto, um mecanismo RPC completo envolve na verdade dois componentes principais:

1. Um protocolo que gerencia as mensagens enviadas entre os processos do cliente e do servidor e que lida com as propriedades potencialmente indesejáveis da rede subjacente.
2. A linguagem de programação e o compilador oferecem suporte para empacotar os argumentos em uma mensagem de solicitação na máquina cliente e, então, traduzir essa mensagem de volta para os argumentos na máquina servidora, e da mesma forma com o valor de retorno (essa parte do mecanismo RPC é geralmente chamada de *compilador stub*).

A Figura 138 descreve esquematicamente o que acontece quando um cliente invoca um procedimento remoto. Primeiro, o cliente chama um stub local para o procedimento, passando a ele os argumentos requeridos pelo procedimento. Esse stub oculta o fato de o procedimento ser remoto, traduzindo os argumentos em uma mensagem de solicitação e, em seguida, invocando um protocolo RPC para enviar a mensagem de solicitação à máquina servidora. No servidor, o protocolo RPC entrega a mensagem de solicitação ao stub do servidor, que a traduz nos argumentos para o procedimento e, em seguida, chama o procedimento local. Após a conclusão do procedimento do servidor, ele retorna uma mensagem de resposta que entrega ao protocolo RPC para transmissão de volta ao cliente. O protocolo RPC no cliente passa essa mensagem para o stub do cliente, que a traduz em um valor de retorno que retorna ao programa cliente.

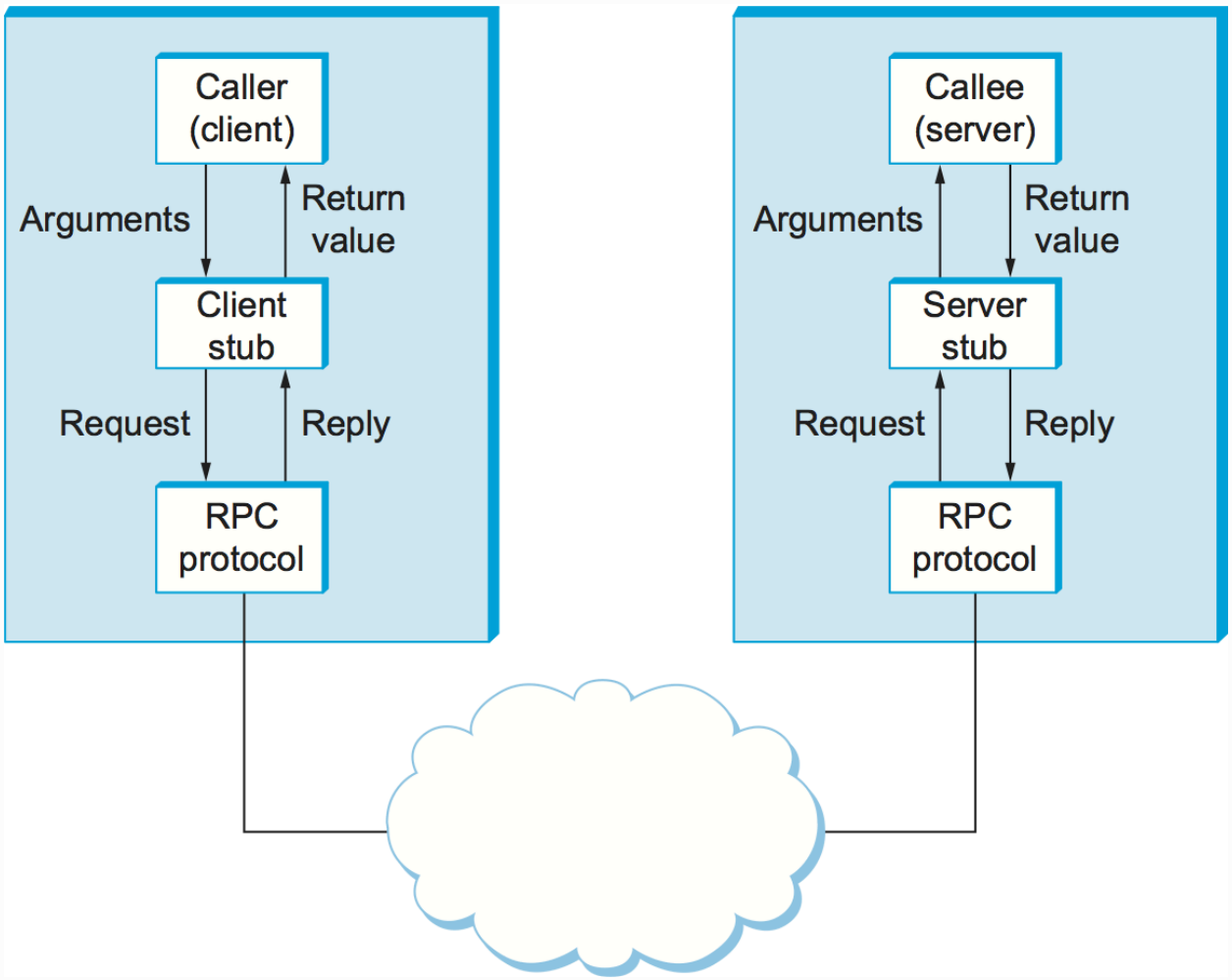

Figura 138. *Mecanismo RPC completo.*

Esta seção considera apenas os aspectos relacionados ao protocolo de um mecanismo RPC. Ou seja, ignora os stubs e se concentra no protocolo RPC, às vezes chamado de protocolo de solicitação/resposta, que transmite mensagens entre cliente e servidor. A transformação de argumentos em mensagens e *vice-versa* é abordada em outro lugar. Também é importante ter em mente que os programas cliente e servidor são escritos em alguma linguagem de programação, o que significa que um determinado mecanismo RPC pode suportar stubs Python, stubs Java, stubs GoLang e assim por diante, cada um dos quais inclui expressões idiomáticas específicas da linguagem para como os procedimentos são invocados.

O termo *RPC* refere-se a um tipo de protocolo e não a um padrão específico como o TCP, portanto, protocolos RPC específicos variam nas funções que desempenham. E, diferentemente do TCP, que é o protocolo de fluxo de bytes confiável dominante, não existe um protocolo RPC dominante. Portanto, nesta seção, falaremos mais sobre opções alternativas de design do que antes.

Identificadores em RPC

Duas funções que devem ser executadas por qualquer protocolo RPC são:

- Forneça um espaço de nome para identificar exclusivamente o procedimento a ser chamado.
- Associe cada mensagem de resposta à mensagem de solicitação correspondente.

O primeiro problema tem algumas semelhanças com o problema de identificação de nós em uma rede (algo que endereços IP fazem, por exemplo). Uma das escolhas de projeto ao identificar coisas é se esse espaço de nomes deve ser plano ou hierárquico. Um espaço de nomes plano simplesmente atribuiria um identificador único e não estruturado (por exemplo, um inteiro) a cada procedimento, e esse número seria carregado em um único campo em uma mensagem de solicitação RPC. Isso exigiria algum tipo de coordenação central para evitar a atribuição do mesmo número de procedimento a dois procedimentos diferentes. Alternativamente, o protocolo poderia implementar um espaço de nomes hierárquico, análogo ao usado para nomes de caminho de arquivo, que requer apenas que o "nome base" de um arquivo seja único dentro de seu diretório. Essa abordagem potencialmente simplifica a tarefa de garantir a exclusividade dos nomes de procedimentos. Um espaço de nomes hierárquico para RPC poderia ser implementado definindo um conjunto de campos no formato da mensagem de solicitação, um para cada nível de nomenclatura em, digamos, um espaço de nomes hierárquico de dois ou três níveis.

A chave para associar uma mensagem de resposta à solicitação correspondente é identificar exclusivamente os pares solicitação-resposta usando um campo de ID de mensagem. Uma mensagem de resposta tinha seu campo de ID de mensagem definido com o mesmo valor da mensagem de solicitação. Quando o módulo RPC cliente recebe a resposta, ele usa o ID da mensagem para procurar a solicitação pendente correspondente. Para que a transação RPC pareça uma chamada de procedimento local para o chamador, este é bloqueado até que a mensagem de resposta seja recebida. Quando a resposta é recebida, o chamador bloqueado é identificado com base no número da solicitação na resposta, o valor de retorno do procedimento remoto é obtido da resposta e o chamador é desbloqueado para que possa retornar com esse valor de retorno.

Um dos desafios recorrentes no RPC é lidar com respostas inesperadas, e vemos isso com IDs de mensagens. Por exemplo, considere a seguinte situação patológica (mas realista). Uma máquina cliente envia uma mensagem de solicitação com ID de mensagem 0, trava e reinicia, e então envia uma mensagem de solicitação não relacionada, também com ID de mensagem 0. O servidor pode não ter percebido que o cliente travou e reiniciou e, ao ver uma mensagem de solicitação com ID de mensagem 0, a reconhece e a descarta como duplicata. O cliente nunca recebe uma resposta à solicitação.

Uma maneira de eliminar esse problema é usar um *ID de inicialização* . O ID de inicialização de uma máquina é um número que é incrementado cada vez que a máquina é reinicializada; esse número é lido de um armazenamento não volátil (por exemplo, um disco ou pen drive), incrementado e gravado de volta no dispositivo de armazenamento durante o procedimento de inicialização da máquina. Esse número é então inserido em todas as mensagens enviadas por esse host. Se uma mensagem for recebida com um ID de mensagem antigo, mas com um novo ID de inicialização, ela será reconhecida como uma nova mensagem. Na prática, o ID da mensagem e o ID de inicialização se combinam para formar um ID exclusivo para cada transação.

Superando as limitações da rede

Os protocolos RPC frequentemente desempenham funções adicionais para lidar com o fato de que as redes não são canais perfeitos. Duas dessas funções são:

- Fornecer entrega de mensagens confiável
- Suporte a tamanhos grandes de mensagens por meio de fragmentação e remontagem

Um protocolo RPC poderia "resolver esse problema" optando por ser executado sobre um protocolo confiável como o TCP, mas, em muitos casos, o protocolo RPC implementa sua própria camada confiável de entrega de mensagens sobre um substrato não confiável (por exemplo, UDP/IP). Tal protocolo RPC provavelmente implementaria a confiabilidade usando confirmações e timeouts, de forma semelhante ao TCP.

O algoritmo básico é simples, como ilustrado pela linha do tempo apresentada na Figura 139. O cliente envia uma mensagem de solicitação e o servidor a confirma. Em seguida, após a execução do procedimento, o servidor envia uma mensagem de resposta e o cliente a confirma.

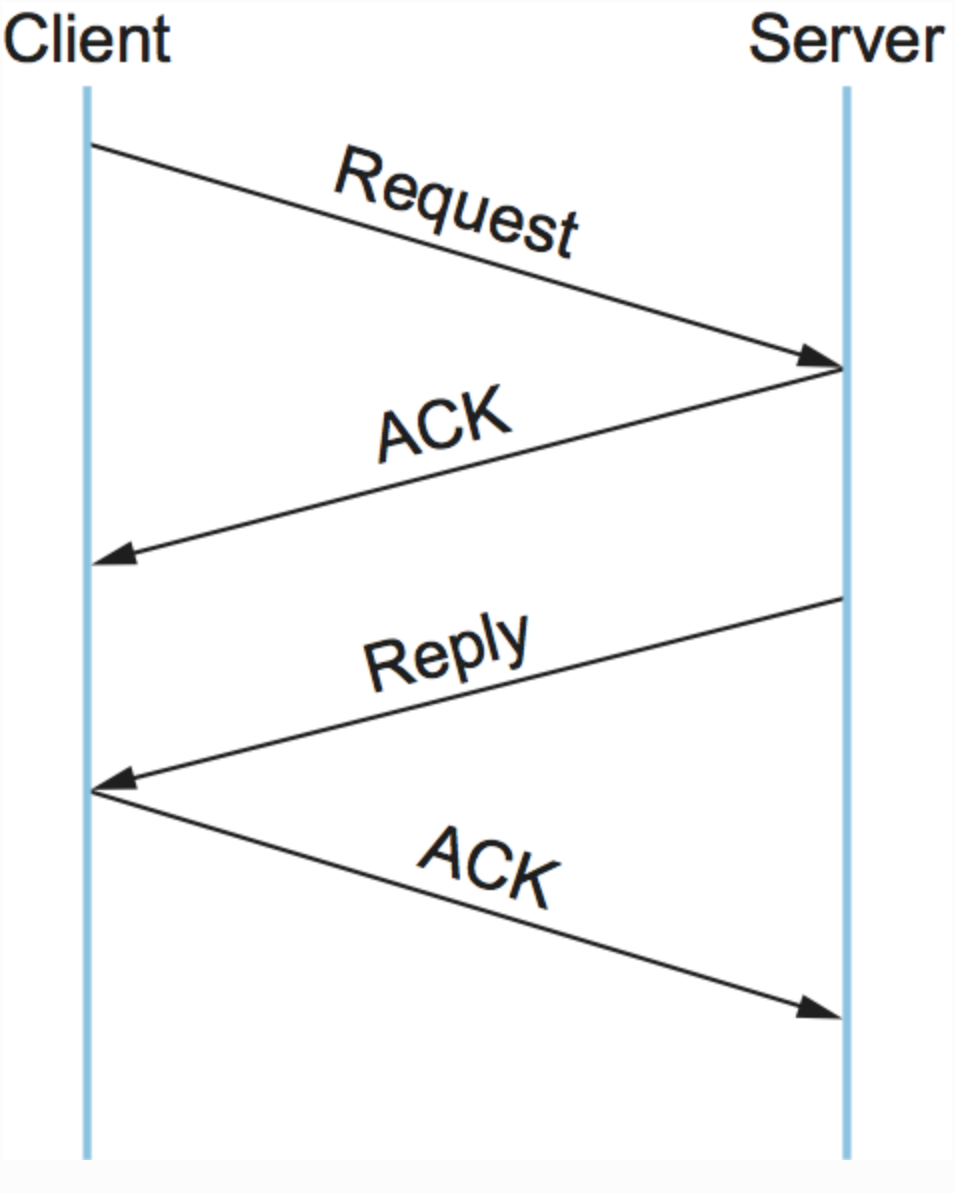

Figura 139. *Cronograma simples para um protocolo RPC confiável.*

Tanto uma mensagem contendo dados (uma mensagem de solicitação ou uma mensagem de resposta) quanto o ACK enviado para confirmar a mensagem podem ser perdidos na rede. Para lidar com essa possibilidade, tanto o cliente quanto o servidor salvam uma cópia de cada mensagem enviada até que um ACK para ela chegue. Cada lado também define um temporizador de RETRANSMIT e reenvia a mensagem caso esse temporizador expire. Ambos os lados zeram esse temporizador e tentam novamente um número combinado de vezes antes de desistir e liberar a mensagem.

Se um cliente RPC recebe uma mensagem de resposta, é evidente que a mensagem de solicitação correspondente deve ter sido recebida pelo servidor. Portanto, a própria mensagem de resposta é uma *confirmação implícita* , e qualquer confirmação adicional do servidor não é logicamente necessária. Da mesma forma, uma mensagem de solicitação poderia confirmar implicitamente a mensagem de resposta anterior — supondo que o protocolo torne as transações de solicitação-resposta sequenciais, de modo que uma transação deva ser concluída antes do início da próxima. Infelizmente, essa sequencialidade limitaria severamente o desempenho do RPC.

Uma saída para essa situação é o protocolo RPC implementar uma abstração *de canal* . Dentro de um determinado canal, as transações de solicitação/resposta são sequenciais — só pode haver uma transação ativa em um determinado canal a qualquer momento — mas pode haver múltiplos canais. Ou, dito de outra forma, a abstração de canal possibilita a *multiplexação* de múltiplas transações de solicitação/resposta RPC entre um par cliente/servidor.

Cada mensagem inclui um campo de ID de canal para indicar a qual canal a mensagem pertence. Uma mensagem de solicitação em um determinado canal reconheceria implicitamente a resposta anterior naquele canal, caso ela ainda não tivesse sido reconhecida. Um programa aplicativo pode abrir vários canais para um servidor se desejar ter mais de uma transação de solicitação/resposta entre eles ao mesmo tempo (o aplicativo precisaria de várias threads). Conforme ilustrado na Figura 140 , a mensagem de resposta serve para reconhecer a mensagem de solicitação, e uma solicitação subsequente reconhece a resposta anterior. Observe que vimos uma abordagem muito semelhante — chamada de *canais lógicos concorrentes* — em uma seção anterior como uma forma de melhorar o desempenho de um mecanismo de confiabilidade do tipo "parar e esperar".

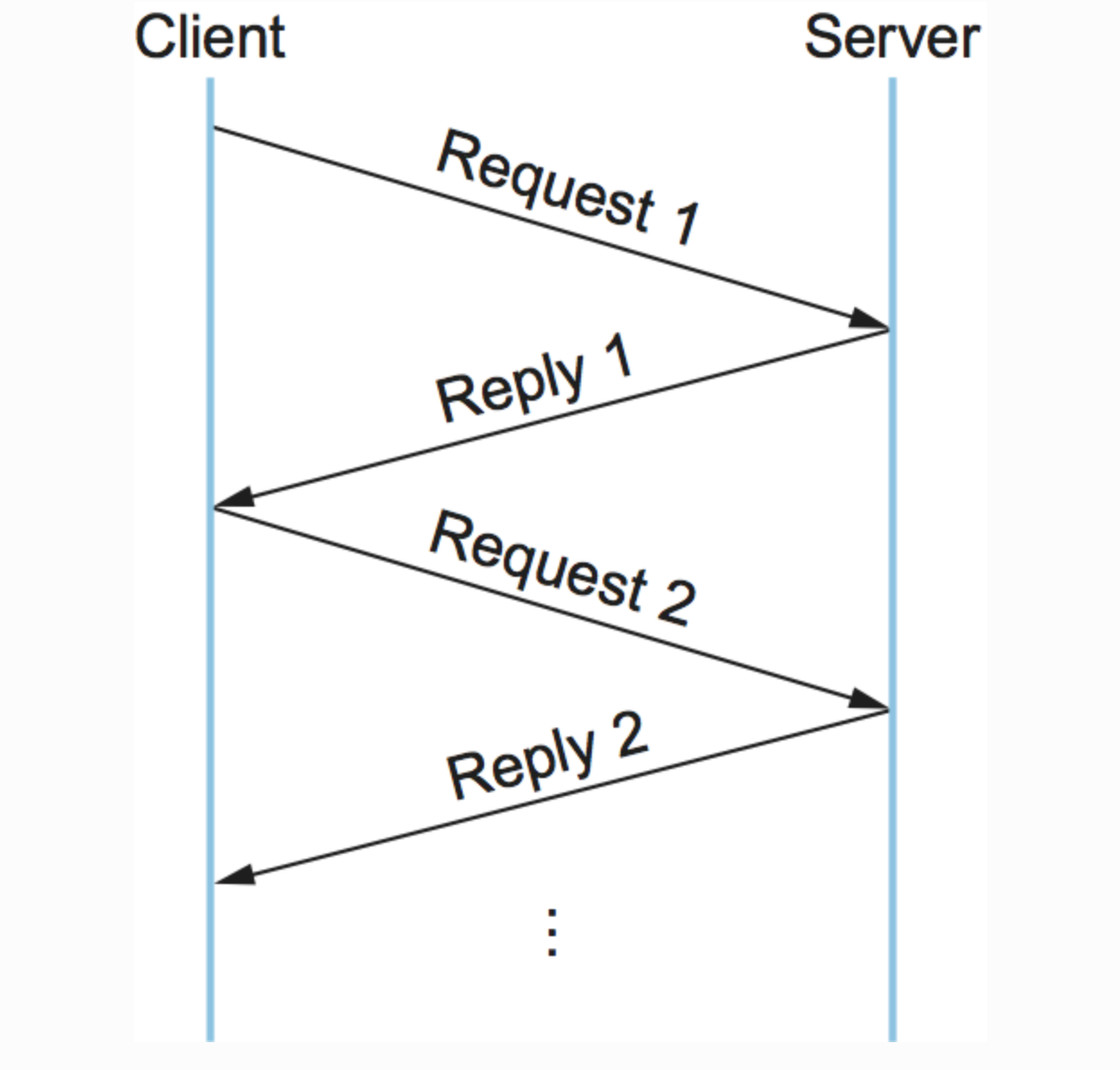

Figura 140. *Cronograma para um protocolo RPC confiável usando reconhecimento implícito.*

Outra complicação que o RPC deve abordar é que o servidor pode levar um tempo arbitrariamente longo para produzir o resultado e, pior ainda, pode travar antes de gerar a resposta. Lembre-se de que estamos falando do período de tempo após o servidor ter reconhecido a solicitação, mas antes de ter enviado a resposta. Para ajudar o cliente a distinguir entre um servidor lento e um servidor inativo, o lado cliente do RPC pode enviar periodicamente uma mensagem "Você está vivo?" para o servidor, e o lado

servidor responde com um ACK. Alternativamente, o servidor pode enviar mensagens "Ainda estou vivo" para o cliente sem que este as tenha solicitado primeiro. A abordagem é mais escalável porque coloca uma parte maior da carga por cliente (gerenciando o temporizador de tempo limite) sobre os clientes.

A confiabilidade do RPC pode incluir a propriedade conhecida como *semântica "no máximo uma vez"* . Isso significa que, para cada mensagem de solicitação enviada pelo cliente, no máximo uma cópia dessa mensagem é entregue ao servidor. Cada vez que o cliente chama um procedimento remoto, esse procedimento é invocado no máximo uma vez na máquina servidora. Dizemos "no máximo uma vez" em vez de "exatamente uma vez" porque é sempre possível que a rede ou a máquina servidora tenham falhado, impossibilitando a entrega de até mesmo uma cópia da mensagem de solicitação.

Para implementar a semântica "no máximo uma vez", o RPC no lado do servidor deve reconhecer solicitações duplicadas (e ignorá-las), mesmo que já tenha respondido com sucesso à solicitação original. Portanto, ele deve manter algumas informações de estado que identifiquem solicitações anteriores. Uma abordagem é identificar solicitações usando números de sequência, de modo que um servidor precise se lembrar apenas do número de sequência mais recente. Infelizmente, isso limitaria um RPC a uma solicitação pendente (para um determinado servidor) por vez, já que uma solicitação deve ser concluída antes que a solicitação com o próximo número de sequência possa ser transmitida. Mais uma vez, os canais fornecem uma solução. O servidor poderia reconhecer solicitações duplicadas lembrando-se do número de sequência atual para cada canal, sem limitar o cliente a uma solicitação por vez.

Por mais óbvio que pareça "no máximo uma vez", nem todos os protocolos RPC suportam esse comportamento. Alguns suportam uma semântica que é jocosamente chamada de semântica *zero-ou-mais* ; ou seja, cada invocação em um cliente resulta na invocação zero ou mais vezes do procedimento remoto. Não é difícil entender como isso causaria problemas para um procedimento remoto que alterasse alguma variável de estado local (por exemplo, incrementasse um contador) ou que tivesse algum efeito

colateral visível externamente (por exemplo, lançasse um míssil) cada vez que fosse invocado. Por outro lado, se o procedimento remoto invocado for *idempotente* — invocações múltiplas têm o mesmo efeito que apenas uma — então o mecanismo RPC não precisa suportar a semântica "no máximo uma vez"; uma implementação mais simples (possivelmente mais rápida) será suficiente.

Assim como no caso da confiabilidade, os dois motivos pelos quais um protocolo RPC pode implementar fragmentação e remontagem de mensagens são que isso não é fornecido pela pilha de protocolos subjacente ou que pode ser implementado de forma mais eficiente pelo protocolo RPC. Considere o caso em que o RPC é implementado sobre UDP/IP e depende do IP para fragmentação e remontagem. Se mesmo um fragmento de uma mensagem não chegar dentro de um determinado período de tempo, o IP descarta os fragmentos que chegaram e a mensagem é efetivamente perdida. Eventualmente, o protocolo RPC (assumindo que implemente confiabilidade) expiraria e retransmitiria a mensagem. Em contraste, considere um protocolo RPC que implementa sua própria fragmentação e remontagem e realiza ACKs ou NACKs (reconhecimento negativo) agressivos de fragmentos individuais. Fragmentos perdidos seriam detectados e retransmitidos mais rapidamente, e apenas os fragmentos perdidos seriam retransmitidos, não a mensagem inteira.

Protocolos síncronos versus assíncronos

Uma maneira de caracterizar um protocolo é se ele é *síncrono* ou *assíncrono* . O significado preciso desses termos depende de onde na hierarquia do protocolo você os utiliza. Na camada de transporte, é mais preciso considerá-los como definindo os extremos de um espectro, em vez de duas alternativas mutuamente exclusivas. O atributo-chave de qualquer ponto ao longo do espectro é o quanto o processo de envio sabe após o retorno da operação de envio de uma mensagem. Em outras palavras, se assumirmos que um programa aplicativo invoca uma `send`operação em um protocolo de transporte, então exatamente o que o aplicativo sabe sobre o sucesso da operação quando `send`ela retorna?

Na extremidade *assíncrona* do espectro, o aplicativo não sabe absolutamente nada quando `send`retorna. Não só não sabe se a mensagem foi recebida pelo seu par, como também não tem certeza se a mensagem saiu com sucesso da máquina local. Na extremidade *síncrona* do espectro, a `send`operação normalmente retorna uma mensagem de resposta. Ou seja, o aplicativo não só sabe que a mensagem enviada foi recebida pelo seu par, como também sabe que o par retornou uma resposta. Assim, os protocolos síncronos implementam a abstração de solicitação/resposta, enquanto os protocolos assíncronos são usados se o remetente quiser transmitir muitas mensagens sem ter que esperar por uma resposta. Usando essa definição, os protocolos RPC geralmente são protocolos síncronos.

Embora não os tenhamos discutido neste capítulo, existem pontos interessantes entre esses dois extremos. Por exemplo, o protocolo de transporte pode ser implementado `send`de forma a bloquear (não retornar) até que a mensagem seja recebida com sucesso na máquina remota, mas retornar antes que o par do remetente naquela máquina a tenha efetivamente processado e respondido. Isso às vezes é chamado de *protocolo de datagrama confiável* .

5.3.2 Implementações de RPC (SunRPC, DCE, gRPC)

Agora, voltamos nossa discussão para alguns exemplos de implementação de protocolos RPC. Eles servirão para destacar algumas das diferentes decisões de projeto tomadas por projetistas de protocolos. Nosso primeiro exemplo é o SunRPC, um protocolo RPC amplamente utilizado, também conhecido como Open Network Computing RPC (ONC RPC). Nosso segundo exemplo, que chamaremos de DCE-RPC, faz parte do Distributed Computing Environment (DCE). O DCE é um conjunto de padrões e softwares para a construção de sistemas distribuídos, definido pela Open Software Foundation (OSF), um consórcio de empresas de informática que originalmente incluía IBM, Digital Equipment Corporation e Hewlett-Packard; hoje, a OSF atende pelo nome de The Open Group. Nosso terceiro exemplo é o gRPC, um mecanismo RPC popular que o Google tornou de código aberto, com base em um

mecanismo RPC que eles vêm usando internamente para implementar serviços de nuvem em seus data centers.

Esses três exemplos representam escolhas alternativas interessantes de design no espaço de soluções RPC, mas para que você não pense que elas são as únicas opções, descrevemos três outros mecanismos semelhantes ao RPC (WSDL, SOAP e REST) no contexto de serviços web no Capítulo 9.

SunRPC

O SunRPC tornou-se um padrão *de fato* graças à sua ampla distribuição em estações de trabalho Sun e ao papel central que desempenha no popular Sistema de Arquivos de Rede (NFS) da Sun. Posteriormente, a IETF o adotou como um protocolo padrão de internet sob o nome ONC RPC.

O SunRPC pode ser implementado sobre vários protocolos de transporte diferentes. A Figura 141 ilustra o gráfico do protocolo quando o SunRPC é implementado em UDP. Como observamos anteriormente nesta seção, um especialista em camadas pode desaprovar a ideia de executar um protocolo de transporte sobre outro protocolo de transporte, ou argumentar que o RPC deve ser algo diferente de um protocolo de transporte, visto que aparece "acima" da camada de transporte. Pragmaticamente, a decisão de projeto de executar o RPC sobre uma camada de transporte existente faz bastante sentido, como ficará evidente na discussão a seguir.

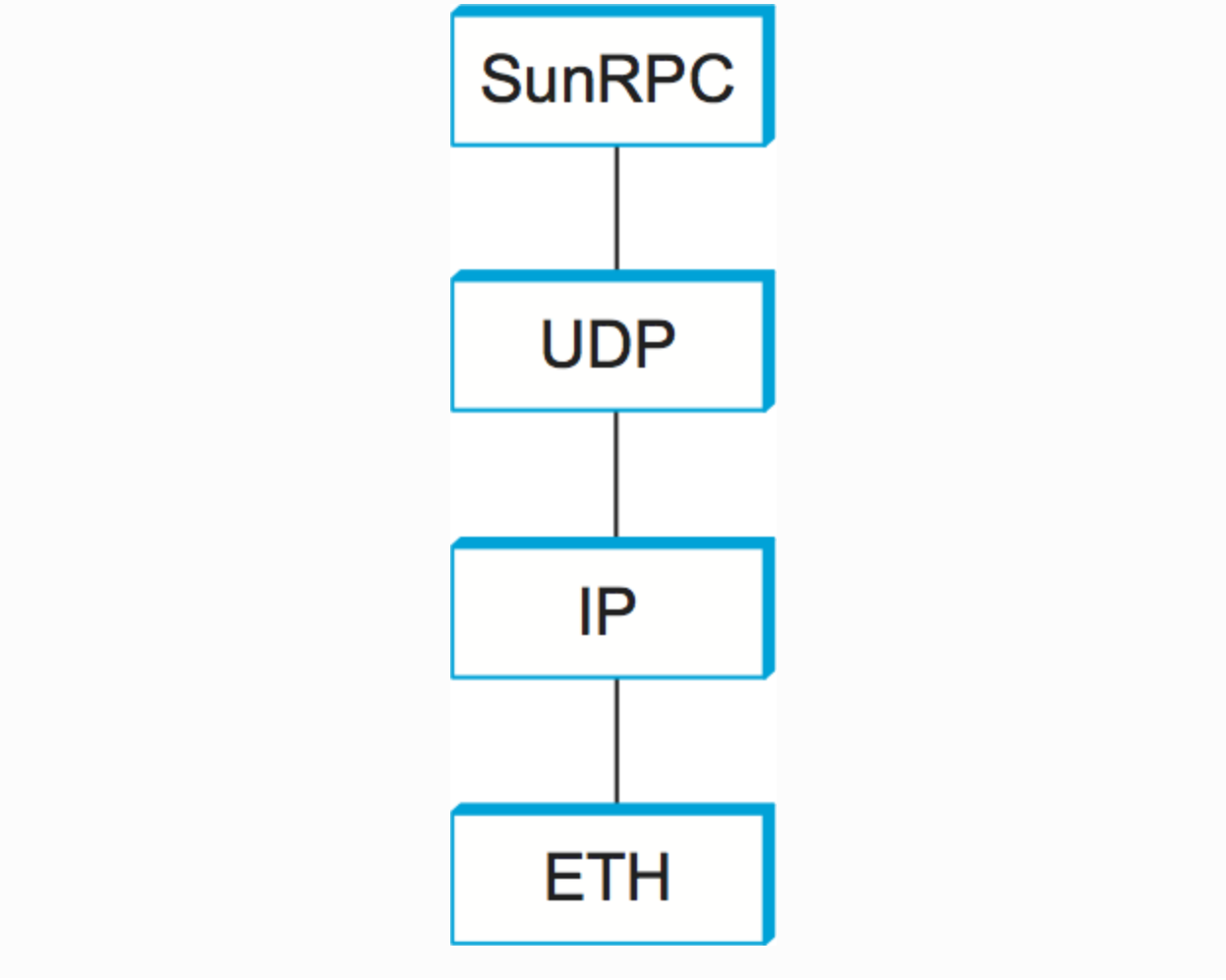

Figura 141. *Gráfico de protocolo para SunRPC sobre UDP.*

O SunRPC usa identificadores de duas camadas para identificar procedimentos remotos: um número de programa de 32 bits e um número de procedimento de 32 bits. (Há também um número de versão de 32 bits, mas o ignoraremos na discussão a seguir.) Por exemplo, o servidor NFS recebeu o número de programa `x00100003`e, dentro desse programa , `getattr`está procedure `1`, `setattr`is procedure `2`, `read`is procedure `6`, `write`is procedure `8`e assim por diante. O número do programa e o número do procedimento são transmitidos no cabeçalho da mensagem de solicitação SunRPC, cujos campos são mostrados na Figura 142. O servidor — que pode suportar vários números de programa — é responsável por chamar o procedimento especificado do programa especificado. Uma solicitação SunRPC realmente representa uma solicitação para chamar o programa e o procedimento especificados na máquina específica para a

qual a solicitação foi enviada, mesmo que o mesmo número de programa possa ser implementado em outras máquinas na mesma rede. Portanto, o endereço da máquina do servidor (por exemplo, um endereço IP) é uma terceira camada implícita do endereço RPC.

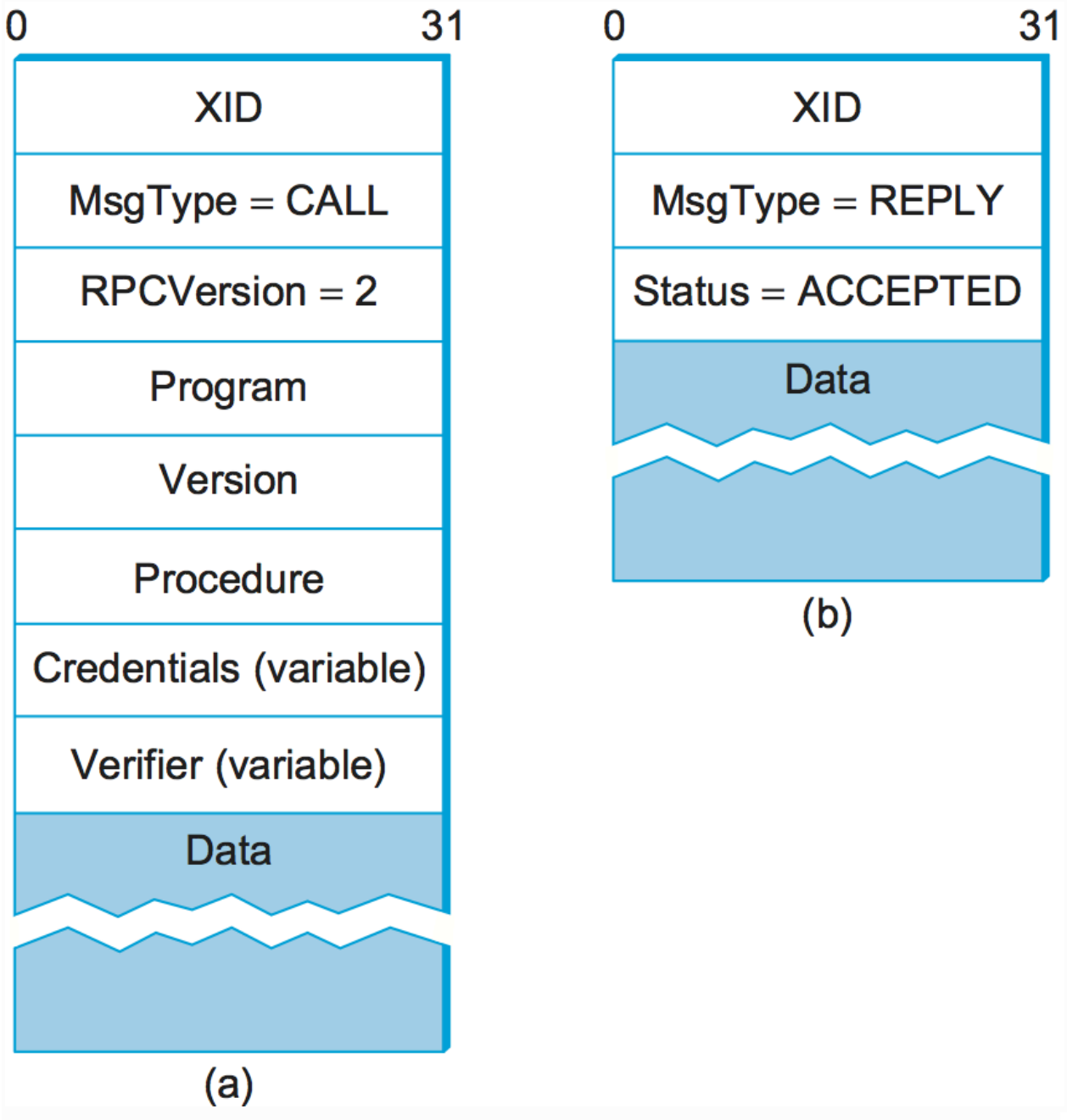

Figura 142. *Formatos de cabeçalho SunRPC: (a) solicitação; (b) resposta.*

Números de programa diferentes podem pertencer a servidores diferentes na mesma máquina. Esses servidores diferentes têm chaves demux de camada de transporte diferentes (por exemplo, portas UDP), a maioria das quais não são números bem conhecidos, mas sim atribuídas dinamicamente. Essas chaves demux são chamadas de *seletores de transporte* . Como um cliente SunRPC que deseja se comunicar com um programa específico pode determinar qual seletor de transporte usar para alcançar o servidor correspondente? A solução é atribuir um endereço bem conhecido a *apenas um* programa na máquina remota e deixar que esse programa cuide da tarefa de informar aos clientes qual seletor de transporte usar para alcançar qualquer outro programa na máquina. A versão original desse programa SunRPC é chamada de *Mapeador de Portas* e suporta apenas UDP e TCP como protocolos subjacentes. Seu número de programa é `x00100000`e sua porta bem conhecida é `111`. O RPCBIND, que evoluiu do Mapeador de Portas, suporta protocolos de transporte subjacentes arbitrários. Conforme cada servidor SunRPC inicia, ele chama um procedimento de registro RPCBIND, na própria máquina do servidor, para registrar seu seletor de transporte e os números de programa que ele suporta. Um cliente remoto pode então chamar um procedimento de consulta RPCBIND para procurar no seletor de transporte um número de programa específico.

Para tornar isso mais concreto, considere um exemplo usando o Mapeador de Portas com UDP. Para enviar uma mensagem de solicitação ao `read`procedimento do NFS, um cliente primeiro envia uma mensagem de solicitação ao Mapeador de Portas na porta UDP conhecida `111`, solicitando que o procedimento `3`seja invocado para mapear o número do programa `x00100003`para a porta UDP onde o programa NFS reside atualmente. O cliente então envia uma mensagem de solicitação SunRPC com o número do programa `x00100003`e o número do procedimento `6`para essa porta UDP, e o módulo SunRPC escutando nessa porta chama o `read`procedimento NFS. O cliente também armazena em cache o mapeamento do número do programa para a porta, de modo que não precise retornar ao Mapeador de Portas sempre que quiser se comunicar com o programa NFS .

1

Na prática, o NFS é um programa tão importante que recebeu sua própria porta UDP, mas, para fins de ilustração, estamos fingindo que não é o caso.

Para associar uma mensagem de resposta à solicitação correspondente, de modo que o resultado do RPC possa ser retornado ao chamador correto, os cabeçalhos das mensagens de solicitação e resposta incluem um `XID`campo (ID da transação), como na Figura 142. A `XID`é um ID de transação exclusivo usado apenas por uma solicitação e a resposta correspondente. Após o servidor responder com sucesso a uma determinada solicitação, ele não se lembra do campo `XID`. Por esse motivo, o SunRPC não garante a semântica "no máximo uma vez".

Os detalhes da semântica do SunRPC dependem do protocolo de transporte subjacente. Ele não implementa sua própria confiabilidade, portanto, só é confiável se o transporte subjacente for confiável. (É claro que qualquer aplicativo executado no SunRPC também pode optar por implementar seus próprios mecanismos de confiabilidade acima do nível do SunRPC.) A capacidade de enviar mensagens de solicitação e resposta maiores que a MTU da rede também depende do transporte subjacente. Em outras palavras, o SunRPC não faz nenhuma tentativa de melhorar o transporte subjacente em termos de confiabilidade e tamanho da mensagem. Como o SunRPC pode ser executado em muitos protocolos de transporte diferentes, isso lhe confere considerável flexibilidade sem complicar o design do próprio protocolo RPC.

Voltando ao formato de cabeçalho SunRPC da Figura 142 , a mensagem de solicitação contém campos de comprimento variável `Credentials`e `Verifier`, ambos usados pelo cliente para se autenticar no servidor — ou seja, para comprovar que o cliente tem o direito de invocar o servidor. A forma como um cliente se autentica em um servidor é uma questão geral que deve ser abordada por qualquer protocolo que queira fornecer um nível razoável de segurança. Este tópico é discutido com mais detalhes em outro capítulo.

DCE-RPC

O DCE-RPC é o protocolo RPC no núcleo do sistema DCE e foi a base do mecanismo RPC subjacente ao DCOM e ao ActiveX da Microsoft. Ele pode ser usado com o compilador stub de Representação de Dados de Rede (NDR) descrito em outro capítulo, mas também serve como o protocolo RPC subjacente à Arquitetura do Corretor de Solicitação de Objetos Comuns (CORBA), que é um padrão da indústria para a construção de sistemas distribuídos e orientados a objetos.

O DCE-RPC, assim como o SunRPC, pode ser implementado sobre vários protocolos de transporte, incluindo UDP e TCP. Ele também é semelhante ao SunRPC, pois define um esquema de endereçamento de dois níveis: o protocolo de transporte desmultiplexa para o servidor correto, o DCE-RPC despacha para um procedimento específico exportado por esse servidor e os clientes consultam um "serviço de mapeamento de endpoint" (semelhante ao Port Mapper do SunRPC) para saber como alcançar um servidor específico. Ao contrário do SunRPC, no entanto, o DCE-RPC implementa a semântica de chamada "no máximo uma vez". (Na verdade, o DCE-RPC suporta múltiplas semânticas de chamada, incluindo uma semântica idempotente semelhante à do SunRPC, mas "no máximo uma vez" é o comportamento padrão.) Existem algumas outras diferenças entre as duas abordagens, que destacaremos nos parágrafos seguintes.

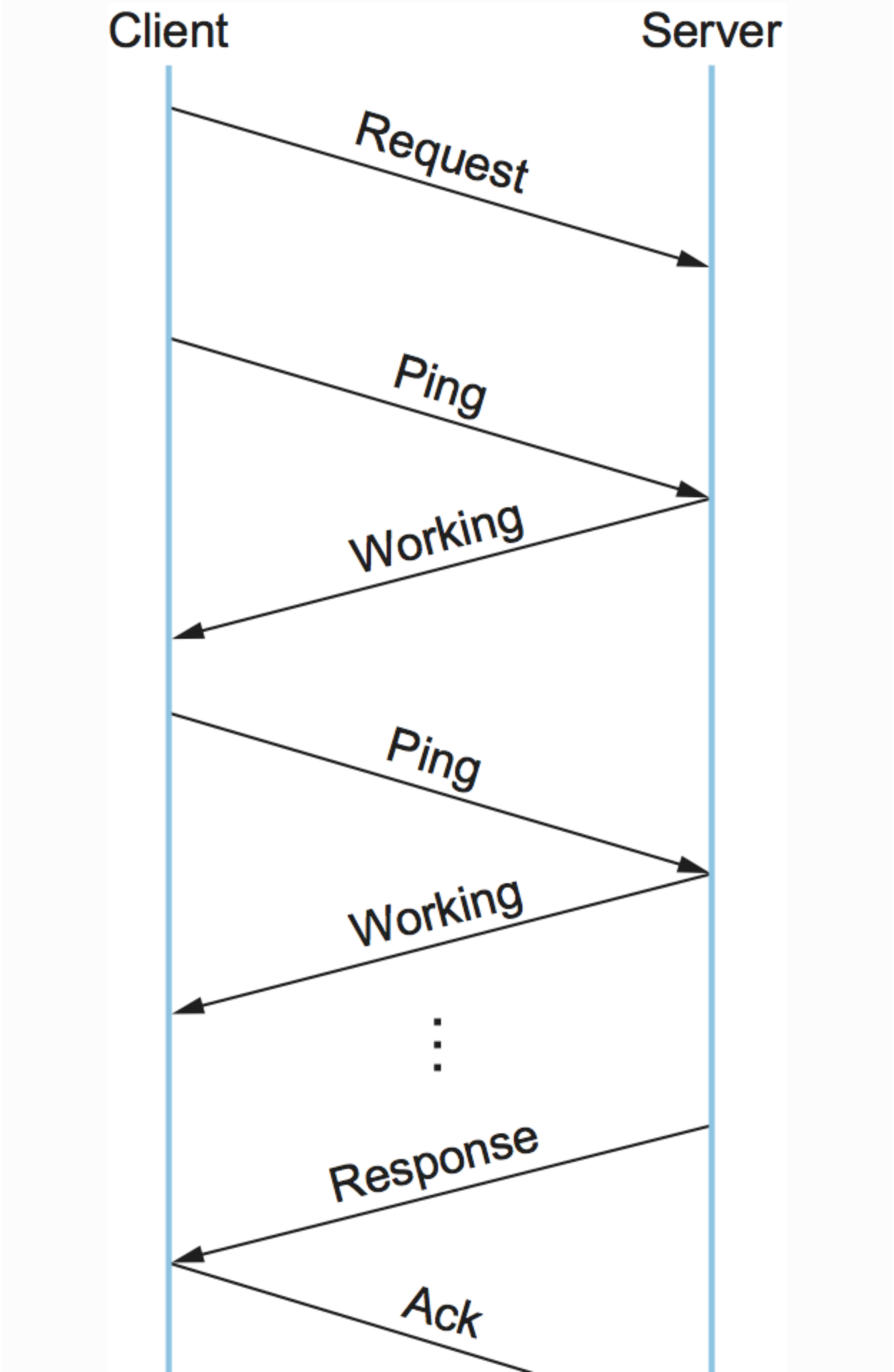
Client
Server
Request
Ping
Working
Ping
Working
Response
Ack

Figura 143. *Troca típica de mensagens DCE-RPC.*

A Figura 143 apresenta uma linha do tempo para a troca típica de mensagens, onde cada mensagem é rotulada por seu tipo DCE-RPC. O cliente envia uma`Request`mensagem, o servidor eventualmente responde com uma `Response`mensagem e o cliente confirma (`Ack`) a resposta. Em vez de o servidor confirmar as mensagens de solicitação, no entanto, o cliente envia periodicamente uma`Ping`mensagem ao servidor, que responde com uma`Working`mensagem para indicar que o procedimento remoto ainda está em andamento. Se a resposta do servidor for recebida com razoável rapidez, nenhum`Ping`s será enviado. Embora não sejam mostrados na figura, outros tipos de mensagens também são suportados. Por exemplo, o cliente pode enviar uma`Quit`mensagem ao servidor, solicitando que ele aborte uma chamada anterior que ainda está em andamento; o servidor responde com uma`Quack` mensagem (confirmação de encerramento). Além disso, o servidor pode responder a uma `Request`mensagem com uma`Reject`mensagem (indicando que uma chamada foi rejeitada) e pode responder a uma`Ping`mensagem com uma `Nocall`mensagem (indicando que o servidor nunca ouviu falar do chamador).

Cada transação de solicitação/resposta no DCE-RPC ocorre no contexto de uma *atividade* . Uma atividade é um canal lógico de solicitação/resposta entre um par de participantes. A qualquer momento, pode haver apenas uma transação de mensagem ativa em um determinado canal. Assim como a abordagem de canal lógico concorrente descrita acima, os programas de aplicação precisam abrir vários canais se quiserem ter mais de uma transação de solicitação/resposta entre eles ao mesmo tempo. A atividade à qual uma mensagem pertence é identificada pelo campo da mensagem `ActivityId`. Um `SequenceNum`campo então distingue entre chamadas feitas como parte da mesma atividade; ele tem a mesma finalidade que `XID` o campo (id da transação) do SunRPC. Ao contrário do SunRPC, o DCE-RPC mantém o controle do último número de sequência usado como parte de uma atividade específica, de modo a garantir a semântica "no máximo uma vez". Para distinguir entre respostas enviadas antes e depois da reinicialização de uma máquina servidora, o DCE-RPC usa um `ServerBoot`campo para armazenar o ID de inicialização da máquina.

Outra escolha de design feita no DCE-RPC que difere do SunRPC é o suporte à fragmentação e remontagem no protocolo RPC. Como observado acima, mesmo que um protocolo subjacente, como o IP, forneça fragmentação/remontagem, um algoritmo mais sofisticado implementado como parte do RPC pode resultar em recuperação mais rápida e menor consumo de largura de banda quando fragmentos são perdidos. O `FragmentNum`campo identifica exclusivamente cada fragmento que compõe uma determinada mensagem de solicitação ou resposta. Cada fragmento DCE-RPC recebe um número de fragmento exclusivo (0, 1, 2, 3 e assim por diante). Tanto o cliente quanto o servidor implementam um mecanismo de confirmação seletiva, que funciona da seguinte maneira: (Descrevemos o mecanismo em termos de um cliente enviando uma mensagem de solicitação fragmentada ao servidor; o mesmo mecanismo se aplica quando um servidor envia uma resposta fragmentada ao cliente.)

Primeiro, cada fragmento que compõe a mensagem de solicitação contém um único `FragmentNum`e um sinalizador que indica se este pacote é um fragmento de uma chamada (`frag`) ou o último fragmento de uma chamada (); mensagens de solicitação que cabem em um único pacote carregam um sinalizador. O servidor sabe que recebeu a mensagem de solicitação completa quando tem o pacote e não há lacunas nos números de fragmentos. Segundo, em resposta a cada fragmento que chega, o servidor envia uma `Fack`mensagem (confirmação de fragmento) para o cliente. Esta confirmação identifica o maior número de fragmento que o servidor recebeu com sucesso. Em outras palavras, a confirmação é cumulativa, muito parecida com o TCP. Além disso, no entanto, o servidor confirma seletivamente quaisquer números de fragmentos maiores que tenha recebido fora de ordem. Ele faz isso com um vetor de bits que identifica esses fragmentos fora de ordem em relação ao maior fragmento em ordem que recebeu. Finalmente, o cliente responde retransmitindo os fragmentos ausentes.

A Figura 144 ilustra como tudo isso funciona. Suponha que o servidor tenha recebido com sucesso fragmentos até o número 20, além dos fragmentos 23, 25 e 26. O servidor responde com um`Fack` que identifica o fragmento 20 como o fragmento mais

alto na ordem, além de um vetor de bits (`SelAck`) com o terceiro (23 = 20 + 3), o quinto (25 = 20 + 5) e o sexto (26 = 20 + 6) bits ativados. Para suportar um vetor de bits (quase) arbitrariamente longo, o tamanho do vetor (medido em palavras de 32 bits) é fornecido no`SelAckLen`campo.

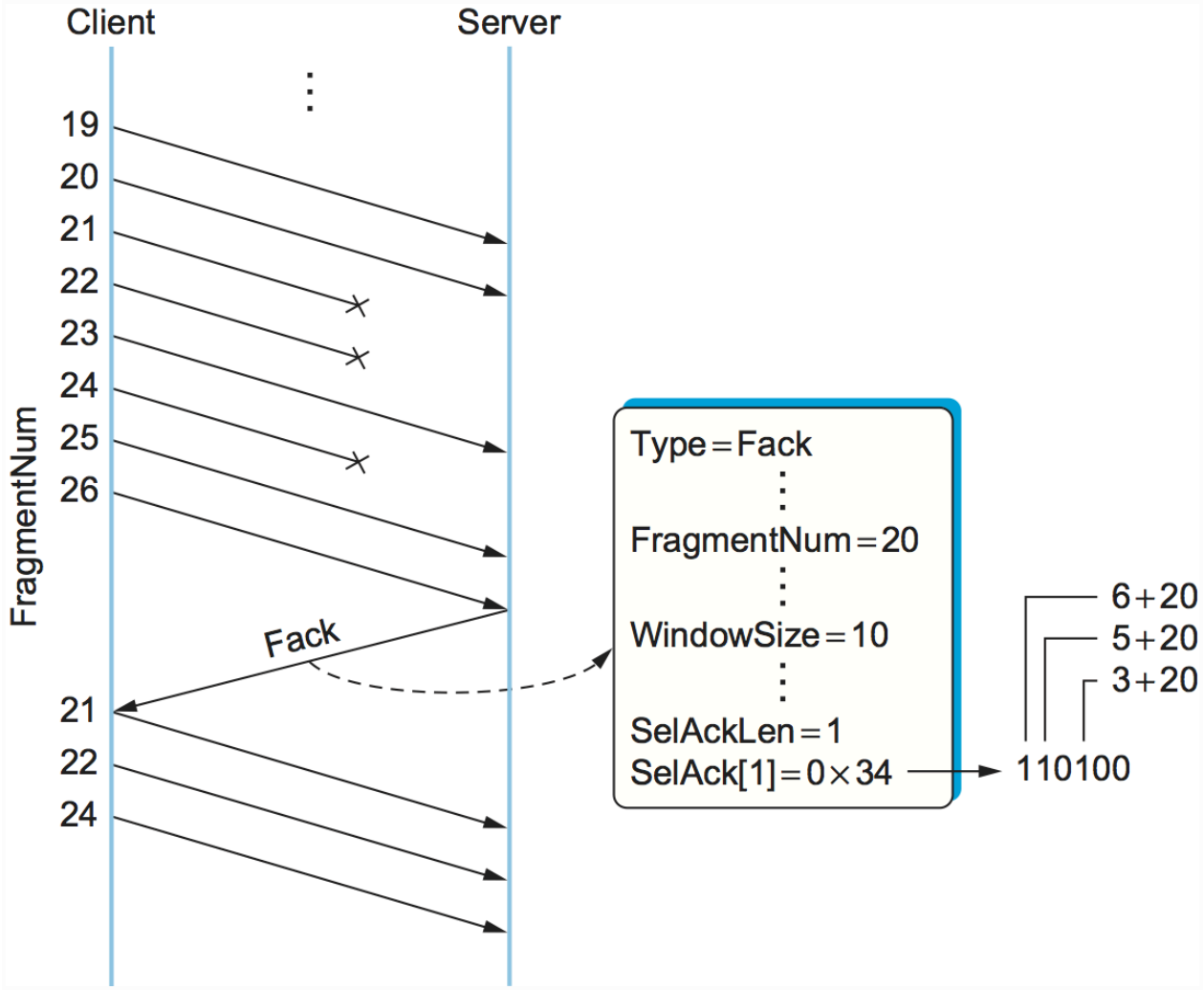

Figura 144. *Fragmentação com confirmações seletivas.*

Dado o suporte do DCE-RPC para mensagens muito grandes — o `FragmentNum` campo tem 16 bits de comprimento, o que significa que pode suportar fragmentos de 64K — não é apropriado que o protocolo exploda todos os fragmentos que compõem uma mensagem o mais rápido possível, pois isso pode sobrecarregar o receptor. Em vez disso, o DCE-RPC implementa um algoritmo de controle de fluxo muito semelhante ao

do TCP. Especificamente, cada `Fack`mensagem não apenas confirma os fragmentos recebidos, mas também informa ao remetente quantos fragmentos ele pode enviar. Este é o propósito do `WindowSize`campo na Figura 144 , que tem exatamente o mesmo propósito que o campo do TCP, `AdvertisedWindow`exceto que conta fragmentos em vez de bytes. O DCE-RPC também implementa um mecanismo de controle de congestionamento semelhante ao do TCP. Dada a complexidade do controle de congestionamento, talvez não seja surpreendente que alguns protocolos RPC o evitem, evitando a fragmentação.

Em resumo, os projetistas têm uma ampla gama de opções disponíveis ao projetar um protocolo RPC. O SunRPC adota uma abordagem mais minimalista e adiciona relativamente pouco ao transporte subjacente, além do essencial para localizar o procedimento correto e identificar mensagens. O DCE-RPC adiciona mais funcionalidades, com a possibilidade de melhorar o desempenho em alguns ambientes, ao custo de maior complexidade.

gRPC

Apesar de suas origens no Google, gRPC não significa Google RPC. O "g" representa algo diferente em cada versão. Na versão 1.10, significava "glamorous" (glamouroso) e, na 1.18, "goose" (ganso). De acordo com as Perguntas Frequentes oficiais do gRPC, agora é uma sigla recursiva: gRPC significa "gRPC Remote Procedure Call" (Chamada de Procedimento Remoto gRPC). Googlers são pessoas selvagens e malucas. Mesmo assim, o gRPC é popular porque disponibiliza a todos — como código aberto — uma década de experiência no Google usando RPC para criar serviços de nuvem escaláveis.

Antes de entrar em detalhes, existem algumas diferenças importantes entre o gRPC e os outros dois exemplos que acabamos de abordar. A principal é que o gRPC foi projetado para serviços em nuvem, e não para o paradigma cliente/servidor mais simples que o precedeu. A diferença reside essencialmente em um nível extra de indireção. No mundo cliente/servidor, o cliente invoca um método em um processo

servidor específico em execução em uma máquina servidora específica. Presume-se que um processo servidor seja suficiente para atender chamadas de todos os processos clientes que possam chamá-lo.

Com serviços em nuvem, o cliente invoca um método em um *serviço* que, para suportar chamadas de muitos clientes arbitrariamente ao mesmo tempo, é implementado por um número escalável de processos de servidor, cada um potencialmente em execução em uma máquina servidora diferente. É aqui que a nuvem entra em jogo: os datacenters disponibilizam um número aparentemente infinito de máquinas servidoras para escalar os serviços em nuvem. Quando usamos o termo "escalável", queremos dizer que o número de processos de servidor idênticos que você escolhe criar depende da carga de trabalho (ou seja, o número de clientes que desejam o serviço a qualquer momento) e esse número pode ser ajustado dinamicamente ao longo do tempo. Outro detalhe é que os serviços em nuvem normalmente não criam um novo processo, por si só, mas sim, eles iniciam um novo *contêiner* , que é essencialmente um processo encapsulado dentro de um ambiente isolado que inclui todos os pacotes de software que o processo precisa para ser executado. O Docker é o exemplo canônico atual de uma plataforma de contêiner.

5.4 Transporte em Tempo Real (RTP)

Nos primórdios da comutação de pacotes, a maioria das aplicações concentrava-se na transferência de arquivos, embora já em 1981 experimentos estivessem em andamento para transportar tráfego em tempo real, como amostras de voz digitalizadas. Chamamos uma aplicação de "tempo real" quando ela possui fortes requisitos para a entrega oportuna de informações. Voz sobre IP (VoIP) é um exemplo clássico de aplicação em tempo real, pois não é possível manter uma conversa com alguém facilmente se a resposta demorar mais do que uma fração de segundo. Como veremos em breve, as aplicações em tempo real impõem algumas demandas específicas ao

protocolo de transporte que não são bem atendidas pelos protocolos discutidos até agora neste capítulo.

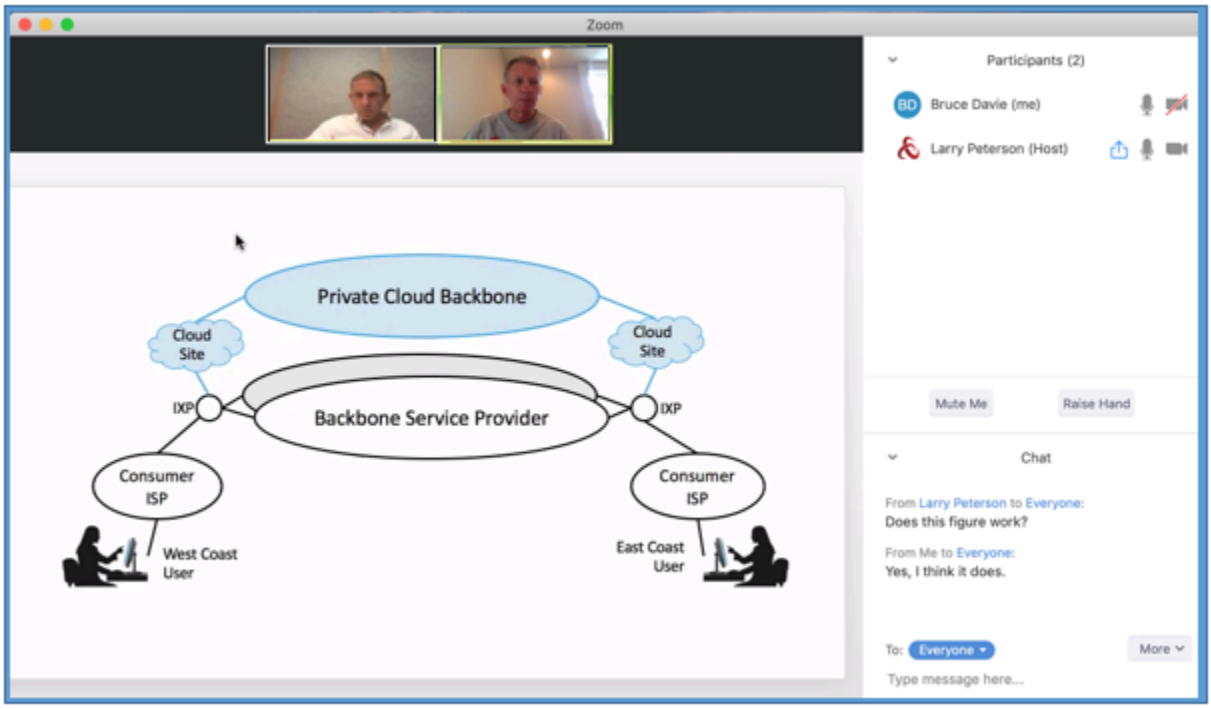

Figura 147. *Interface de usuário de uma ferramenta de videoconferência.*

Aplicações multimídia — aquelas que envolvem vídeo, áudio e dados — às vezes são divididas em duas classes: aplicações *interativas* e aplicações *de streaming* . A Figura 147 mostra os autores usando um exemplo de ferramenta de conferência típica da classe interativa. Juntamente com o VoIP, esses são os tipos de aplicações com os requisitos de tempo real mais rigorosos.

Aplicações de streaming normalmente entregam fluxos de áudio ou vídeo de um servidor para um cliente e são tipificadas por produtos comerciais como o Spotify. O streaming de vídeo, tipificado pelo YouTube e Netflix, tornou-se uma das formas dominantes de tráfego na internet. Como as aplicações de streaming não permitem interação entre pessoas, elas impõem requisitos de tempo real um pouco menos rigorosos aos protocolos subjacentes. A pontualidade ainda é importante, no entanto —

por exemplo, você quer que um vídeo comece a ser reproduzido logo após clicar em "reproduzir" e, uma vez que ele comece a ser reproduzido, pacotes atrasados farão com que ele pare ou criem algum tipo de degradação visual. Portanto, embora as aplicações de streaming não sejam estritamente em tempo real, elas ainda têm o suficiente em comum com aplicações multimídia interativas para justificar a consideração de um protocolo comum para ambos os tipos de aplicação.

Já deve estar claro que os projetistas de um protocolo de transporte para aplicações multimídia e em tempo real enfrentam um verdadeiro desafio: definir os requisitos de forma suficientemente ampla para atender às necessidades de aplicações muito diferentes. Eles também devem estar atentos às interações entre diferentes aplicações, como a sincronização de fluxos de áudio e vídeo. Veremos a seguir como essas preocupações afetaram o projeto do principal protocolo de transporte em tempo real em uso atualmente: *o Protocolo de Transporte em Tempo Real* (RTP).

Grande parte do RTP, na verdade, deriva da funcionalidade do protocolo originalmente incorporada ao próprio aplicativo. Dois dos primeiros aplicativos desse tipo foram o `vic`RTP e `vat`o RTP, o primeiro suportando vídeo em tempo real e o último suportando áudio em tempo real. Ambos os aplicativos originalmente rodavam diretamente sobre UDP, enquanto os projetistas descobriam quais recursos eram necessários para lidar com a natureza em tempo real da comunicação. Posteriormente, eles perceberam que esses recursos poderiam ser úteis para muitos outros aplicativos e definiram um protocolo com eles. Esse protocolo foi eventualmente padronizado como RTP.

O RTP pode ser executado sobre muitos protocolos de camada inferior, mas ainda é comumente executado sobre UDP. Isso nos leva à pilha de protocolos mostrada na Figura 148. Observe que, portanto, estamos executando um protocolo de transporte sobre um protocolo de transporte. Não há regra contra isso e, na verdade, faz muito sentido, visto que o UDP fornece um nível mínimo de funcionalidade e a

demultiplexação básica baseada em números de porta é exatamente o que o RTP precisa como ponto de partida. Portanto, em vez de recriar números de porta no RTP, o RTP terceiriza a função de demultiplexação para o UDP.

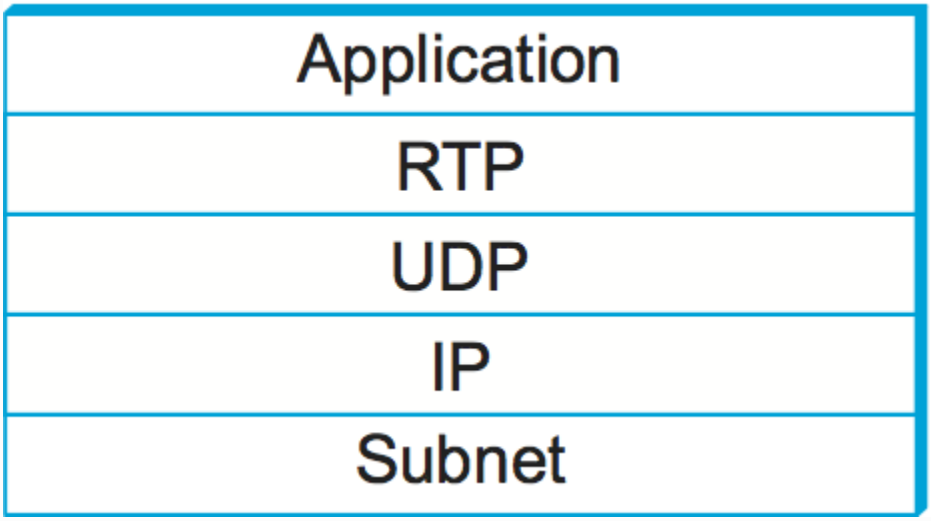

Figura 148. *Pilha de protocolos para aplicações multimídia usando RTP.*

5.4.1 Requisitos

O requisito mais básico para um protocolo multimídia de uso geral é permitir que aplicações semelhantes interoperem entre si. Por exemplo, deve ser possível que duas aplicações de audioconferência implementadas independentemente se comuniquem. Isso sugere imediatamente que as aplicações devem usar o mesmo método de codificação e compressão de voz; caso contrário, os dados enviados por uma parte serão incompreensíveis para a parte receptora. Como existem diversos esquemas de codificação para voz, cada um com suas próprias compensações entre qualidade, requisitos de largura de banda e custo computacional, provavelmente seria uma má ideia decretar que apenas um desses esquemas possa ser usado. Em vez disso, nosso protocolo deve fornecer uma maneira de um remetente informar ao destinatário qual esquema de codificação deseja usar e, possivelmente, negociar até que um esquema disponível para ambas as partes seja identificado.

Assim como no caso do áudio, existem muitos esquemas de codificação de vídeo diferentes. Assim, vemos que a primeira função comum que o RTP pode fornecer é a capacidade de comunicar a escolha do esquema de codificação. Observe que isso também serve para identificar o tipo de aplicação (por exemplo, áudio ou vídeo); uma vez que sabemos qual algoritmo de codificação está sendo usado, sabemos também que tipo de dados está sendo codificado.

Outro requisito importante é permitir que o destinatário de um fluxo de dados determine a relação temporal entre os dados recebidos. Aplicações em tempo real precisam colocar os dados recebidos em um *buffer de reprodução* para suavizar o jitter que pode ter sido introduzido no fluxo de dados durante a transmissão pela rede. Portanto, algum tipo de registro de data e hora dos dados será necessário para permitir que o destinatário os reproduza no momento apropriado.

Relacionada à cronometragem de um único fluxo de mídia está a questão da sincronização de múltiplas mídias em uma conferência. O exemplo óbvio disso seria a sincronização de um fluxo de áudio e vídeo originados do mesmo remetente. Como veremos a seguir, este é um problema um pouco mais complexo do que a determinação do tempo de reprodução de um único fluxo.

Outra função importante a ser fornecida é a indicação de perda de pacotes. Observe que uma aplicação com limites de latência restritos geralmente não pode usar um transporte confiável como o TCP, pois a retransmissão de dados para corrigir a perda provavelmente faria com que o pacote chegasse tarde demais para ser útil. Portanto, a aplicação deve ser capaz de lidar com pacotes perdidos, e o primeiro passo para lidar com eles é perceber que eles estão de fato perdidos. Por exemplo, uma aplicação de vídeo que utiliza codificação MPEG pode tomar ações diferentes quando um pacote é perdido, dependendo se o pacote veio de um quadro I, um quadro B ou um quadro P.

A perda de pacotes também é um indicador potencial de congestionamento. Como aplicações multimídia geralmente não operam sobre TCP, elas também perdem os recursos de prevenção de congestionamento do TCP. No entanto, muitas aplicações

multimídia são capazes de responder ao congestionamento — por exemplo, alterando os parâmetros do algoritmo de codificação para reduzir a largura de banda consumida. Claramente, para que isso funcione, o receptor precisa notificar o remetente da ocorrência de perdas para que este possa ajustar seus parâmetros de codificação.

Outra função comum em aplicações multimídia é o conceito de indicação de limite de quadro. Um quadro, neste contexto, é específico da aplicação. Por exemplo, pode ser útil notificar um aplicativo de vídeo de que um determinado conjunto de pacotes corresponde a um único quadro. Em uma aplicação de áudio, é útil marcar o início de um "surto de fala", que é uma coleção de sons ou palavras seguidos de silêncio. O receptor pode então identificar os silêncios entre os surtos de fala e usá-los como oportunidades para mover o ponto de reprodução. Isso segue a observação de que ligeiros encurtamentos ou alongamentos dos espaços entre as palavras não são perceptíveis aos usuários, enquanto encurtamentos ou alongamentos das próprias palavras são perceptíveis e incômodos.

Uma função final que poderíamos incluir no protocolo é uma forma de identificar remetentes mais amigável do que um endereço IP. Conforme ilustrado na Figura 147 , aplicativos de conferência de áudio e vídeo podem exibir strings como as mostradas em seus painéis de controle e, portanto, o protocolo do aplicativo deve suportar a associação de tal string a um fluxo de dados.

Além da funcionalidade exigida do nosso protocolo, observamos um requisito adicional: ele deve fazer uso razoavelmente eficiente da largura de banda. Em outras palavras, não queremos introduzir muitos bits extras que precisam ser enviados com cada pacote na forma de um cabeçalho longo. A razão para isso é que os pacotes de áudio, que são um dos tipos mais comuns de dados multimídia, tendem a ser pequenos, de modo a reduzir o tempo necessário para preenchê-los com amostras. Pacotes de áudio longos significariam alta latência devido à packetização, o que tem um efeito negativo na qualidade percebida das conversas. (Este foi um dos fatores na escolha do comprimento das células ATM.) Como os próprios pacotes de dados são curtos, um cabeçalho grande significaria que uma quantidade relativamente grande de largura de

banda do link seria usada pelos cabeçalhos, reduzindo assim a capacidade disponível para dados "úteis". Veremos vários aspectos do projeto do RTP que foram influenciados pela necessidade de manter o cabeçalho curto.

Pode-se argumentar se cada recurso descrito acima *realmente* precisa estar em um protocolo de transporte em tempo real, e provavelmente encontraremos mais alguns que poderiam ser adicionados. A ideia principal aqui é facilitar a vida dos desenvolvedores de aplicações, fornecendo-lhes um conjunto útil de abstrações e blocos de construção para suas aplicações. Por exemplo, ao implementar um mecanismo de carimbo de tempo no RTP, poupamos cada desenvolvedor de uma aplicação de tempo real de inventar a sua própria. Também aumentamos as chances de duas aplicações de tempo real diferentes interoperarem.

5.4.2 Projeto RTP

Agora que vimos a longa lista de requisitos para o nosso protocolo de transporte para multimídia, vamos analisar os detalhes do protocolo que foi especificado para atender a esses requisitos. Este protocolo, RTP, foi desenvolvido no IETF e é amplamente utilizado. O padrão RTP, na verdade, define um par de protocolos: RTP e o Protocolo de Controle de Transporte em Tempo Real (RTCP). O primeiro é usado para a troca de dados multimídia, enquanto o último é usado para enviar periodicamente informações de controle associadas a um determinado fluxo de dados. Ao executar sobre UDP, o fluxo de dados RTP e o fluxo de controle RTCP associado usam portas consecutivas da camada de transporte. Os dados RTP usam um número de porta par e as informações de controle RTCP usam o número de porta imediatamente superior (ímpar).

Como o RTP é projetado para suportar uma ampla variedade de aplicações, ele fornece um mecanismo flexível pelo qual novas aplicações podem ser desenvolvidas sem revisar repetidamente o próprio protocolo RTP. Para cada classe de aplicação (por exemplo, áudio), o RTP define um *perfil* e um ou mais *formatos* . O perfil fornece uma variedade de informações que garantem um entendimento comum dos campos no

cabeçalho RTP para aquela classe de aplicação, como ficará aparente quando examinarmos o cabeçalho em detalhes. A especificação do formato explica como os dados que seguem o cabeçalho RTP devem ser interpretados. Por exemplo, o cabeçalho RTP pode ser seguido apenas por uma sequência de bytes, cada um dos quais representa uma única amostra de áudio obtida em um intervalo definido após a anterior. Alternativamente, o formato dos dados pode ser muito mais complexo; um fluxo de vídeo codificado em MPEG, por exemplo, precisaria ter uma boa dose de estrutura para representar todos os diferentes tipos de informação.

Conclusão principal

O design do RTP incorpora um princípio arquitetônico conhecido como *Enquadramento em Nível de Aplicação* (ALF). Esse princípio foi proposto por Clark e Tennenhouse em 1990 como uma nova maneira de projetar protocolos para aplicações multimídia emergentes. Eles reconheceram que essas novas aplicações provavelmente não seriam bem atendidas por protocolos existentes, como o TCP, e que, além disso, poderiam não ser bem atendidas por qualquer tipo de protocolo "tamanho único". No cerne desse princípio está a crença de que uma aplicação entende melhor suas próprias necessidades. Por exemplo, uma aplicação de vídeo MPEG sabe como se recuperar melhor de quadros perdidos e como reagir de forma diferente se um quadro I ou um quadro B for perdido. A mesma aplicação também entende melhor como segmentar os dados para transmissão — por exemplo, é melhor enviar os dados de quadros diferentes em datagramas diferentes, de modo que um pacote perdido corrompa apenas um quadro, não dois. É por esse motivo que o RTP deixa muitos dos detalhes do protocolo para os documentos de perfil e formato específicos de uma aplicação. [Próximo]

Formato do cabeçalho

A Figura 149 mostra o formato de cabeçalho usado pelo RTP. Os primeiros 12 bytes estão sempre presentes, enquanto os identificadores de origem contribuintes são

usados apenas em determinadas circunstâncias. Após esse cabeçalho, pode haver extensões de cabeçalho opcionais, conforme descrito abaixo. Por fim, o cabeçalho é seguido pelo conteúdo útil do RTP, cujo formato é determinado pela aplicação. A intenção desse cabeçalho é conter apenas os campos que provavelmente serão usados por muitas aplicações diferentes, uma vez que qualquer coisa muito específica de uma única aplicação seria transportada de forma mais eficiente no conteúdo útil do RTP apenas para essa aplicação.

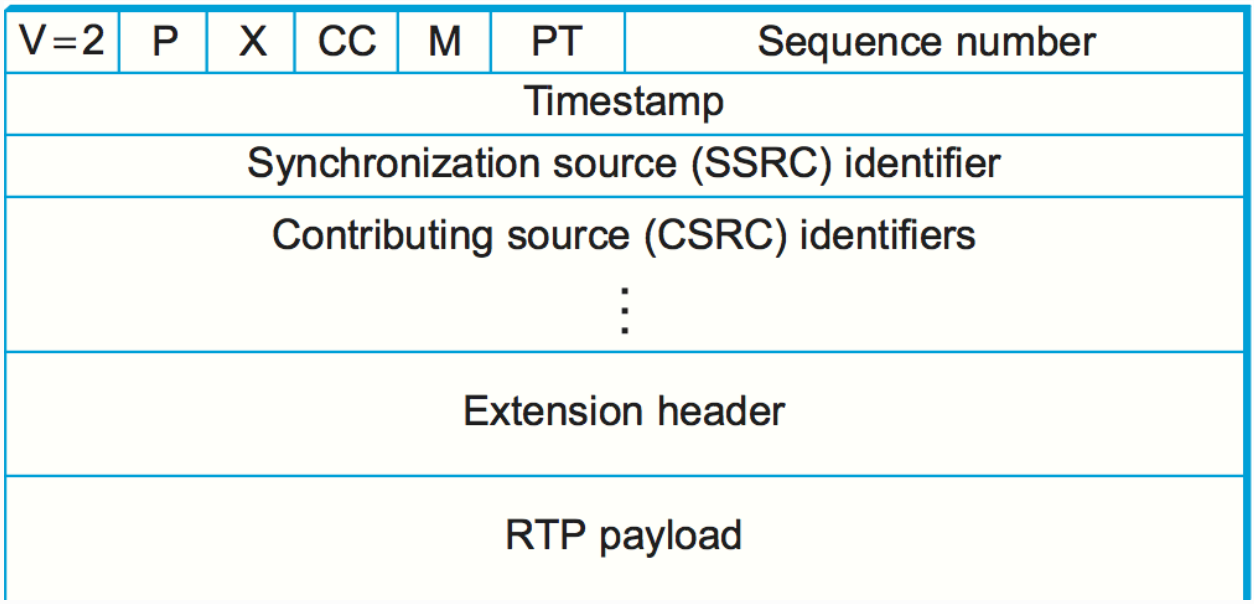

Figura 149. *Formato do cabeçalho RTP.*

Os dois primeiros bits são um identificador de versão, que contém o valor 2 na versão RTP implantada no momento da escrita. Você pode pensar que os projetistas do protocolo foram bastante ousados ao pensar que 2 bits seriam suficientes para conter todas as versões futuras do RTP, mas lembre-se de que os bits são escassos no cabeçalho RTP. Além disso, o uso de perfis para diferentes aplicações torna menos provável que muitas revisões do protocolo RTP base sejam necessárias. De qualquer forma, se for necessário outra versão do RTP além da versão 2, seria possível considerar uma mudança no formato do cabeçalho para que mais de uma versão futura seja possível. Por exemplo, um novo cabeçalho RTP com o valor 3 no campo de versão poderia ter um campo "subversão" em algum outro lugar do cabeçalho.

O próximo bit é o bit *padding* (`P`), que é definido em circunstâncias nas quais a carga útil RTP foi preenchida por algum motivo. Dados RTP podem ser preenchidos para preencher um bloco de um determinado tamanho, conforme exigido por um algoritmo de criptografia, por exemplo. Nesse caso, o comprimento completo do cabeçalho RTP, dados e preenchimento seria transmitido pelo cabeçalho do protocolo de camada inferior (por exemplo, o cabeçalho UDP), e o último byte do preenchimento conteria uma contagem de quantos bytes devem ser ignorados. Isso é ilustrado na Figura 150. Observe que essa abordagem para preenchimento remove qualquer necessidade de um campo de comprimento no cabeçalho RTP (atendendo assim ao objetivo de manter o cabeçalho curto); no caso comum de nenhum preenchimento, o comprimento é deduzido do protocolo de camada inferior.

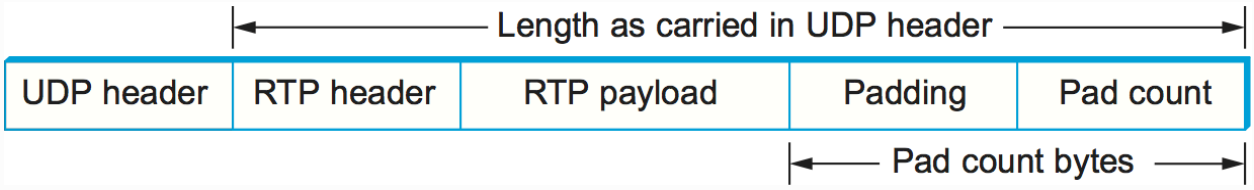

Figura 150. *Preenchimento de um pacote RTP.*

O bit *de extensão* (`X`) é usado para indicar a presença de um cabeçalho de extensão, que seria definido para uma aplicação específica e seguiria o cabeçalho principal. Esses cabeçalhos raramente são usados, pois geralmente é possível definir um cabeçalho específico de carga útil como parte da definição do formato de carga útil para uma aplicação específica.

O `X`bit é seguido por um campo de 4 bits que conta o número de *fontes contribuintes* , se houver alguma incluída no cabeçalho. As fontes contribuintes são discutidas abaixo.

Observamos acima a necessidade frequente de algum tipo de indicação de quadro; isso é fornecido pelo bit marcador, que tem um uso específico para cada perfil. Para uma aplicação de voz, ele poderia ser definido no início de um surto de fala, por exemplo. O campo de tipo de carga útil de 7 bits vem a seguir; ele indica o tipo de

dados multimídia transportados neste pacote. Um possível uso desse campo seria permitir que uma aplicação alternasse de um esquema de codificação para outro com base em informações sobre a disponibilidade de recursos na rede ou em feedback sobre a qualidade da aplicação. O uso exato do tipo de carga útil também é determinado pelo perfil da aplicação.

Observe que o tipo de carga útil geralmente não é usado como uma chave de demultiplexação para direcionar dados para diferentes aplicações (ou para diferentes fluxos dentro de uma única aplicação, como o fluxo de áudio e vídeo para uma videoconferência). Isso ocorre porque essa demultiplexação normalmente é fornecida em uma camada inferior (por exemplo, por UDP, conforme descrito em uma seção anterior). Portanto, dois fluxos de mídia usando RTP normalmente usariam números de porta UDP diferentes.

O número de sequência é usado para permitir que o receptor de um fluxo RTP detecte pacotes ausentes ou mal ordenados. O remetente simplesmente incrementa o valor em um para cada pacote transmitido. Observe que o RTP não faz nada ao detectar um pacote perdido, ao contrário do TCP, que corrige a perda (por retransmissão) e a interpreta como uma indicação de congestionamento (o que pode fazer com que ele reduza o tamanho da janela). Em vez disso, cabe ao aplicativo decidir o que fazer quando um pacote é perdido, pois essa decisão provavelmente depende muito do aplicativo. Por exemplo, um aplicativo de vídeo pode decidir que a melhor coisa a fazer quando um pacote é perdido é reproduzir o último quadro recebido corretamente. Alguns aplicativos também podem decidir modificar seus algoritmos de codificação para reduzir as necessidades de largura de banda em resposta à perda, mas isso não é uma função do RTP. Não seria sensato para o RTP decidir que a taxa de envio deve ser reduzida, pois isso poderia tornar o aplicativo inútil.

A função do campo timestamp é permitir que o receptor reproduza amostras em intervalos apropriados e sincronize diferentes fluxos de mídia. Como diferentes aplicações podem exigir diferentes granularidades de temporização, o RTP em si não especifica as unidades em que o tempo é medido. Em vez disso, o timestamp é apenas

um contador de "tiques", onde o tempo entre os tiques depende da codificação em uso. Por exemplo, um aplicativo de áudio que amostra dados uma vez a cada 125 µs poderia usar esse valor como sua resolução de clock. A granularidade do clock é um dos detalhes especificados no perfil RTP ou no formato de carga útil de um aplicativo.

O valor do carimbo de data/hora no pacote é um número que representa o horário em que a *primeira* amostra do pacote foi gerada. O carimbo de data/hora não reflete a hora do dia; apenas as diferenças entre os carimbos de data/hora são relevantes. Por exemplo, se o intervalo de amostragem for de 125 µs e a primeira amostra do pacote n+1 tiver sido gerada 10 ms após a primeira amostra do pacote n, então o número de instantes de amostragem entre essas duas amostras será

$$\text{TimeBetweenPackets/TimePerSample}$$

$$= (10 \times 10^{-3}) / (125 \times 10^{-6}) = 80$$

Supondo que a granularidade do relógio seja a mesma que o intervalo de amostragem, o registro de data e hora no pacote n+1 seria maior que o do pacote n em 80. Observe que menos de 80 amostras podem ter sido enviadas devido a técnicas de compressão, como detecção de silêncio, e ainda assim o registro de data e hora permite que o receptor reproduza as amostras com a relação temporal correta.

A fonte de sincronização (SSRC) é um número de 32 bits que identifica exclusivamente uma única fonte de um fluxo RTP. Em uma conferência multimídia, cada remetente escolhe um SSRC aleatório e espera-se que resolva conflitos no improvável caso de duas fontes escolherem o mesmo valor. Ao tornar o identificador de fonte algo diferente do endereço de rede ou de transporte da fonte, o RTP garante a independência do protocolo de camada inferior. Ele também permite que um único nó com múltiplas fontes (por exemplo, várias câmeras) distinga essas fontes. Quando um único nó gera diferentes fluxos de mídia (por exemplo, áudio e vídeo), não é necessário usar o mesmo SSRC em cada fluxo, pois existem mecanismos no RTCP (descritos abaixo) para permitir a sincronização intermediária.

A fonte contribuinte (CSRC) é usada apenas quando vários fluxos RTP passam por um mixer. Um mixer pode ser usado para reduzir os requisitos de largura de banda de uma conferência, recebendo dados de várias fontes e enviando-os como um único fluxo. Por exemplo, os fluxos de áudio de vários palestrantes simultâneos podem ser decodificados e recodificados como um único fluxo de áudio. Nesse caso, o mixer se lista como a fonte de sincronização, mas também lista as fontes contribuintes — os valores SSRC dos palestrantes que contribuíram para o pacote em questão.

5.4.3 Protocolo de Controle

O RTCP fornece um fluxo de controle associado a um fluxo de dados para uma aplicação multimídia. Esse fluxo de controle fornece três funções principais:

1. Feedback sobre o desempenho do aplicativo e da rede
2. Uma maneira de correlacionar e sincronizar diferentes fluxos de mídia que vieram do mesmo remetente
3. Uma maneira de transmitir a identidade de um remetente para exibição em uma interface de usuário.

A primeira função pode ser útil para detectar e responder a congestionamentos. Alguns aplicativos conseguem operar em taxas diferentes e podem usar dados de desempenho para decidir usar um esquema de compressão mais agressivo para reduzir o congestionamento, por exemplo, ou enviar um fluxo de maior qualidade quando há pouco congestionamento. O feedback de desempenho também pode ser útil no diagnóstico de problemas de rede.

Você pode pensar que a segunda função já é fornecida pelo ID da fonte de sincronização (SSRC) do RTP, mas na verdade não é. Como já observado, várias câmeras de um único nó podem ter valores de SSRC diferentes. Além disso, não há exigência de que um fluxo de áudio e vídeo do mesmo nó use o mesmo SSRC. Como podem ocorrer colisões de valores de SSRC, pode ser necessário alterar o valor de SSRC de um fluxo. Para lidar com esse problema, o RTCP utiliza o conceito de um

nome canônico (CNAME) atribuído a um remetente, que é então associado aos vários valores de SSRC que podem ser usados por esse remetente por meio de mecanismos RTCP.

A simples correlação entre dois fluxos é apenas parte do problema da sincronização intermediária. Como fluxos diferentes podem ter relógios completamente diferentes (com granularidades diferentes e até mesmo diferentes níveis de imprecisão, ou desvio), é necessário encontrar uma maneira de sincronizar os fluxos entre si com precisão. O RTCP resolve esse problema transmitindo informações de tempo que correlacionam a hora real do dia com os carimbos de tempo dependentes da taxa de relógio, transportados em pacotes de dados RTP.

O RTCP define vários tipos diferentes de pacotes, incluindo

- Relatórios de remetente, que permitem que remetentes ativos em uma sessão relatem estatísticas de transmissão e recepção
- Relatórios do receptor, que os receptores que não são remetentes usam para relatar estatísticas de recepção
- Descrições de origem, que contêm CNAMEs e outras informações de descrição do remetente
- Pacotes de controle específicos do aplicativo

Esses diferentes tipos de pacotes RTCP são enviados pelo protocolo de camada inferior, que, como observamos, normalmente é o UDP. Vários pacotes RTCP podem ser compactados em uma única PDU do protocolo de nível inferior. É necessário que pelo menos dois pacotes RTCP sejam enviados em cada PDU de nível inferior: um deles é um pacote de relatório; o outro é um pacote de descrição da fonte. Outros pacotes podem ser incluídos até os limites de tamanho impostos pelos protocolos de camada inferior.

Antes de analisarmos mais detalhadamente o conteúdo de um pacote RTCP, observamos que existe um problema potencial com cada membro de um grupo

multicast enviando tráfego de controle periódico. A menos que tomemos medidas para limitá-lo, esse tráfego de controle tem o potencial de ser um consumidor significativo de largura de banda. Em uma audioconferência, por exemplo, é provável que não mais do que dois ou três remetentes enviem dados de áudio a qualquer instante, já que não faz sentido que todos falem ao mesmo tempo. Mas não existe um limite social para que todos enviem tráfego de controle, e isso pode ser um problema grave em uma conferência com milhares de participantes. Para lidar com esse problema, o RTCP possui um conjunto de mecanismos pelos quais os participantes reduzem a frequência de relatórios à medida que o número de participantes aumenta. Essas regras são um tanto complexas, mas o objetivo básico é: limitar a quantidade total de tráfego RTCP a uma pequena porcentagem (tipicamente 5%) do tráfego de dados RTP. Para atingir esse objetivo, os participantes devem saber quanta largura de banda de dados provavelmente estará em uso (por exemplo, a quantidade necessária para enviar três fluxos de áudio) e o número de participantes. Eles aprendem o primeiro por meios externos ao RTP (conhecido como *gerenciamento de sessão* , discutido no final desta seção) e o segundo a partir dos relatórios RTCP de outros participantes. Como os relatórios RTCP podem ser enviados a uma taxa muito baixa, pode ser possível obter apenas uma contagem aproximada do número atual de destinatários, mas isso normalmente é suficiente. Além disso, recomenda-se alocar mais largura de banda RTCP para remetentes ativos, partindo do princípio de que a maioria dos participantes gostaria de ver os relatórios deles — por exemplo, para descobrir quem está falando.

Depois que um participante determina quanta largura de banda pode consumir com o tráfego RTCP, ele começa a enviar relatórios periódicos na taxa apropriada. Os relatórios do remetente e do destinatário diferem apenas no fato de que os primeiros incluem algumas informações adicionais sobre o remetente. Ambos os tipos de relatórios contêm informações sobre os dados recebidos de todas as fontes no período de relatório mais recente.

As informações extras em um relatório do remetente consistem em

- Um registro de data e hora contendo a hora real do dia em que este relatório foi gerado
- O carimbo de data/hora RTP correspondente ao momento em que o relatório foi gerado
- Contagens cumulativas de pacotes e bytes enviados por este remetente desde que iniciou a transmissão

Observe que as duas primeiras quantidades podem ser usadas para habilitar a sincronização de diferentes fluxos de mídia da mesma fonte, mesmo que esses fluxos usem granularidades de relógio diferentes em seus fluxos de dados RTP, pois isso fornece a chave para converter a hora do dia em registros de data e hora RTP.

Os relatórios de remetente e destinatário contêm um bloco de dados por fonte ouvida desde o último relatório. Cada bloco contém as seguintes estatísticas para a fonte em questão:

- Seu SSRC
- A fração de pacotes de dados desta fonte que foram perdidos desde o envio do último relatório (calculada pela comparação do número de pacotes recebidos com o número de pacotes esperados; este último valor pode ser determinado a partir dos números de sequência RTP)
- Número total de pacotes perdidos desta fonte desde a primeira vez que foi ouvido
- Maior número de sequência recebido desta fonte (estendido para 32 bits para contabilizar o encapsulamento do número de sequência)
- Jitter estimado entre chegadas para a origem (calculado pela comparação do espaçamento entre chegadas dos pacotes recebidos com o espaçamento esperado no momento da transmissão)
- Último registro de data e hora real recebido via RTCP para esta fonte
- Atraso desde o último relatório do remetente recebido via RTCP para esta fonte

Como você pode imaginar, os destinatários dessas informações podem aprender todo tipo de informação sobre o estado da sessão. Em particular, eles podem ver se outros destinatários estão recebendo qualidade muito melhor de algum remetente do que eles, o que pode ser um indício de que uma reserva de recursos precisa ser feita ou de que há um problema na rede que precisa ser resolvido. Além disso, se um remetente perceber que muitos destinatários estão sofrendo altas perdas de seus pacotes, ele pode decidir reduzir sua taxa de envio ou usar um esquema de codificação mais resiliente a perdas.

O aspecto final do RTCP que consideraremos é o pacote de descrição da fonte. Tal pacote contém, no mínimo, o SSRC do remetente e o CNAME do remetente. O nome canônico é derivado de tal forma que todos os aplicativos que geram fluxos de mídia que podem precisar ser sincronizados (por exemplo, fluxos de áudio e vídeo gerados separadamente pelo mesmo usuário) escolherão o mesmo CNAME, mesmo que escolham valores de SSRC diferentes. Isso permite que um receptor identifique o fluxo de mídia que veio do mesmo remetente. O formato mais comum do CNAME é `user@host`, onde `host`é o nome de domínio totalmente qualificado da máquina remetente. Assim, um aplicativo iniciado pelo usuário cujo nome de usuário está sendo `jdoe` executado na máquina `cicada.cs.princeton.edu`usaria a string `jdoe@cicada.cs.princeton.edu`como seu CNAME. O número grande e variável de bytes usados nessa representação a tornaria uma má escolha para o formato de um SSRC, uma vez que o SSRC é enviado com cada pacote de dados e deve ser processado em tempo real. Permitir que CNAMEs sejam vinculados a valores de SSRC em mensagens RTCP periódicas permite um formato compacto e eficiente para o SSRC.

Outros itens podem ser incluídos no pacote de descrição da fonte, como o nome real e o endereço de e-mail do usuário. Eles são usados nas telas da interface do usuário e para contatar os participantes, mas são menos essenciais para a operação do RTP do que o CNAME.

Assim como o TCP, o RTP e o RTCP são um par de protocolos bastante complexo. Essa complexidade advém, em grande parte, do desejo de facilitar a vida dos projetistas de aplicações. Como há um número infinito de aplicações possíveis, o desafio ao projetar um protocolo de transporte é torná-lo genérico o suficiente para atender às necessidades amplamente variadas de muitas aplicações diferentes, sem tornar o próprio protocolo impossível de implementar. O RTP tem se mostrado muito bem-sucedido nesse aspecto, formando a base para muitas aplicações multimídia em tempo real executadas na internet atualmente.

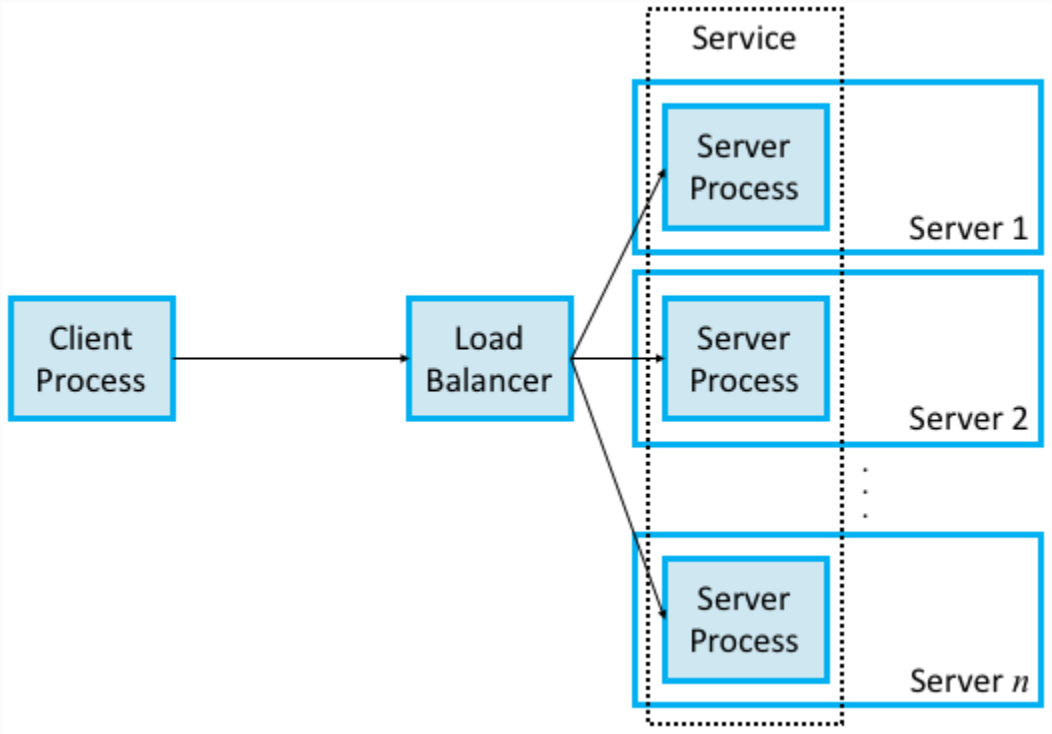

Figura 145. *Usando RPC para invocar um serviço de nuvem escalável.*

Voltando à afirmação de que um serviço é essencialmente um nível extra de indireção sobreposto a um servidor, tudo isso significa que o chamador identifica o serviço que deseja invocar, e um *balanceador de carga* direciona essa invocação para um dos muitos processos de servidor disponíveis (contêineres) que implementam esse serviço,

conforme mostrado na Figura 145. O balanceador de carga pode ser implementado de diferentes maneiras, incluindo um dispositivo de hardware, mas normalmente é implementado por um processo proxy executado em uma máquina virtual (também hospedada na nuvem) em vez de um dispositivo físico.

Há um conjunto de melhores práticas para implementar o código do servidor real que eventualmente responde a essa solicitação, e algumas ferramentas adicionais de nuvem para criar/destruir contêineres e balancear solicitações entre eles. O Kubernetes é o exemplo canônico atual de um sistema de gerenciamento de contêineres, e a *arquitetura de microsserviços* é o que chamamos de melhores práticas na construção de serviços dessa maneira nativa da nuvem. Ambos são tópicos interessantes, mas estão além do escopo deste livro.

O que nos interessa aqui é o protocolo de transporte no cerne do gRPC. Novamente, há um grande distanciamento dos dois protocolos de exemplo anteriores, não em termos dos problemas fundamentais que precisam ser resolvidos, mas em termos da abordagem do gRPC para solucioná-los. Em resumo, o gRPC "terceiriza" muitos dos problemas para outros protocolos, deixando que o gRPC essencialmente empacotasse esses recursos em um formato fácil de usar. Vejamos os detalhes.

Primeiro, o gRPC é executado sobre TCP em vez de UDP, o que significa que ele terceiriza os problemas de gerenciamento de conexão e transmissão confiável de mensagens de solicitação e resposta de tamanho arbitrário. Segundo, o gRPC, na verdade, é executado sobre uma versão segura do TCP chamada *Transport Layer Security* (TLS) — uma camada fina que fica acima do TCP na pilha de protocolos — o que significa que ele terceiriza a responsabilidade de proteger o canal de comunicação para que adversários não possam espionar ou sequestrar a troca de mensagens. Terceiro, o gRPC, na verdade, é executado sobre HTTP/2 (que por sua vez é sobreposto ao TCP e ao TLS), o que significa que o gRPC terceiriza ainda dois outros problemas: (1) codificação/compactação eficiente de dados binários em uma mensagem, (2) multiplexação de múltiplas chamadas de procedimento remoto em uma única conexão TCP. Em outras palavras, o gRPC codifica o identificador do método

remoto como um URI, os parâmetros de solicitação para o método remoto como conteúdo na mensagem HTTP e o valor de retorno do método remoto na resposta HTTP. A pilha gRPC completa é ilustrada na Figura 146 , que também inclui os elementos específicos da linguagem. (Um ponto forte do gRPC é o amplo conjunto de linguagens de programação que ele suporta, com apenas um pequeno subconjunto mostrado na Figura 146.)

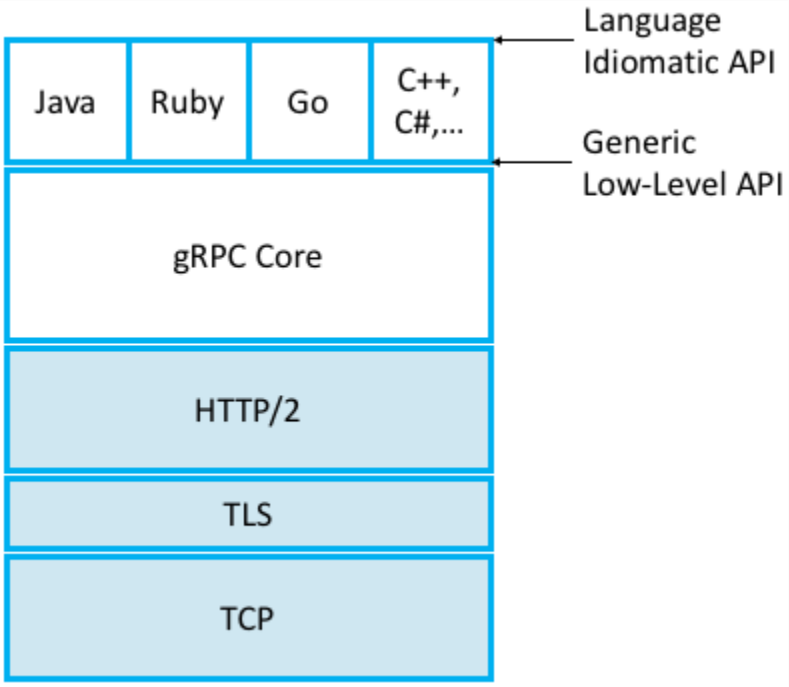

Figura 146. *Núcleo gRPC empilhado sobre HTTP, TLS e TCP e suportando uma coleção de linguagens.*

Discutimos TLS no Capítulo 8 (no contexto de uma ampla gama de tópicos de segurança) e HTTP no Capítulo 9 (no contexto do que tradicionalmente são vistos como protocolos de nível de aplicação). Mas nos encontramos em um interessante ciclo de dependência: RPC é uma variante de protocolo de transporte usado para implementar aplicações distribuídas, HTTP é um exemplo de protocolo de nível de aplicação e, ainda assim, o gRPC roda sobre HTTP, e não o contrário.

A explicação resumida é que a estratificação fornece uma maneira conveniente para os humanos entenderem sistemas complexos, mas o que realmente estamos tentando fazer é resolver um conjunto de problemas (por exemplo, transferir mensagens de tamanho arbitrário de forma confiável, identificar remetentes e destinatários, corresponder mensagens de solicitação com mensagens de resposta e assim por diante) e a maneira como essas soluções são agrupadas em protocolos, e esses protocolos são então sobrepostos uns aos outros, é consequência de mudanças incrementais ao longo do tempo. Pode-se argumentar que é um acidente histórico. Se a Internet tivesse começado com um mecanismo RPC tão onipresente quanto o TCP, o HTTP poderia ter sido implementado sobre ele (como quase todos os outros protocolos de nível de aplicação descritos no Capítulo 9) e o Google teria gasto seu tempo aprimorando *esse* protocolo em vez de inventar um próprio (como eles e outros têm feito com o TCP). O que aconteceu, em vez disso, foi que a web se tornou o aplicativo matador da internet, o que significou que seu protocolo de aplicação (HTTP) passou a ter suporte universal no restante da infraestrutura da internet: firewalls, balanceadores de carga, criptografia, autenticação, compressão e assim por diante. Como todos esses elementos de rede foram projetados para funcionar bem com HTTP, este se tornou efetivamente o protocolo universal de transporte de solicitação/resposta da internet.

Voltando às características únicas do gRPC, o maior valor que ele traz é incorporar *streaming* ao mecanismo RPC, ou seja, o gRPC suporta quatro padrões diferentes de solicitação/resposta:

1. RPC simples: o cliente envia uma única mensagem de solicitação e o servidor responde com uma única mensagem de resposta.
2. RPC de Streaming do Servidor: O cliente envia uma única mensagem de solicitação e o servidor responde com um fluxo de mensagens de resposta. O cliente conclui a execução assim que recebe todas as respostas do servidor.

3. RPC de streaming do cliente: o cliente envia um fluxo de solicitações ao servidor, e o servidor envia de volta uma única resposta, normalmente (mas não necessariamente) após receber todas as solicitações do cliente.
4. Streaming bidirecional RPC: a chamada é iniciada pelo cliente, mas depois disso, o cliente e o servidor podem ler e gravar solicitações e respostas em qualquer ordem; os fluxos são completamente independentes.

Essa liberdade extra na forma como o cliente e o servidor interagem significa que o protocolo de transporte gRPC precisa enviar metadados e mensagens de controle adicionais — além das mensagens de solicitação e resposta — entre os dois pares. Exemplos incluem códigos `Error`e `Status`(para indicar sucesso ou por que algo falhou), `Timeouts`(para indicar quanto tempo um cliente está disposto a esperar por uma resposta), `PING`(um aviso de keep-alive para indicar que um lado ou outro ainda está em execução), `EOS` (aviso de fim de fluxo para indicar que não há mais solicitações ou respostas) e `GOAWAY`(um aviso dos servidores aos clientes para indicar que eles não aceitarão mais novos fluxos). Ao contrário de muitos outros protocolos neste livro, onde mostramos o formato do cabeçalho do protocolo, a maneira como essas informações de controle são transmitidas entre os dois lados é amplamente ditada pelo protocolo de transporte subjacente, neste caso o HTTP/2. Por exemplo, como veremos no Capítulo 9, o HTTP já inclui um conjunto de campos de cabeçalho e códigos de resposta dos quais o gRPC se aproveita.

Talvez você queira ler a discussão sobre HTTP no Capítulo 9 antes de continuar, mas o que se segue é bastante simples. Uma simples solicitação RPC (sem streaming) pode incluir a seguinte mensagem HTTP do cliente para o servidor:

```
HEADERS (flags = END_HEADERS)

:method = POST

:scheme = http

:path = /google.pubsub.v2.PublisherService/CreateTopic
```

```
:authority = pubsub.googleapis.com

grpc-timeout = 1S

content-type = application/grpc+proto

grpc-encoding = gzip

authorization = Bearer y235.wef315yfh138vh31hv93hv8h3v

DATA (flags = END_STREAM)

<Length-Prefixed Message>
```

levando à seguinte mensagem de resposta do servidor para o cliente:

```
HEADERS (flags = END_HEADERS)

:status = 200

grpc-encoding = gzip

content-type = application/grpc+proto

DATA

<Length-Prefixed Message>

HEADERS (flags = END_STREAM, END_HEADERS)

grpc-status = 0 # OK

trace-proto-bin = jher831yy13JHy3hc
```

Neste exemplo, `HEADERS`e `DATA`são duas mensagens de controle HTTP padrão, que efetivamente delimitam entre “o cabeçalho da mensagem” e “a carga útil da mensagem”. Especificamente, cada linha a seguir `HEADERS`(mas antes de `DATA`) é um par que compõe o cabeçalho (pense em cada linha como análoga a um campo de cabeçalho); os pares que começam com dois pontos (por exemplo,) fazem parte do padrão HTTP (por exemplo, status indica sucesso); e os pares que não começam com dois pontos são personalizações específicas do gRPC (por exemplo, indica que os dados na mensagem a seguir foram compactados usando e indica que o cliente definiu um tempo limite de um segundo).`attribute = value:status = 200200grpc-encoding = gzipgzipgrpc-timeout = 1S`

Há uma última parte a explicar. A linha do cabeçalho

```
content-type = application/grpc+proto
```

indica que o corpo da mensagem (conforme delimitado pela `DATA`linha) é significativo apenas para o programa aplicativo (ou seja, o método do servidor) do qual este cliente está solicitando o serviço. Mais especificamente, a `+proto`string especifica que o destinatário será capaz de interpretar os bits na mensagem de acordo com uma especificação de interface *do Buffer de Protocolo* (abreviado como `proto`). Buffers de Protocolo são a maneira do gRPC especificar como os parâmetros passados ao servidor são codificados em uma mensagem, que por sua vez é usada para gerar os stubs que ficam entre o mecanismo RPC subjacente e as funções reais que estão sendo chamadas (veja a Figura 138). Este é um tópico que abordaremos no Capítulo 7.

Conclusão principal

A questão fundamental é que mecanismos complexos como o RPC, antes empacotados como um pacote monolítico de software (como o SunRPC e o

DCE-RPC), hoje são construídos pela montagem de uma variedade de partes menores, cada uma das quais resolve um problema específico. O gRPC é um exemplo dessa abordagem e uma ferramenta que permite sua adoção posterior. A arquitetura de microsserviços mencionada anteriormente nesta subseção aplica a estratégia de "construído a partir de pequenas partes" a aplicativos de nuvem inteiros (por exemplo, Uber, Lyft, Netflix, Yelp, Spotify), onde o gRPC é frequentemente o mecanismo de comunicação usado por essas pequenas partes para trocar mensagens entre si. [Próximo]

Perspectiva: HTTP é a nova cintura estreita

A Internet tem sido descrita como tendo uma arquitetura *de cintura estreita* , com um protocolo universal no meio (IP), ampliando-se para suportar muitos protocolos de transporte e aplicação acima dele (por exemplo, TCP, UDP, RTP, SunRPC, DCE-RPC, gRPC, SMTP, HTTP, SNMP) e capaz de rodar sobre muitas tecnologias de rede abaixo (por exemplo, Ethernet, PPP, WiFi, SONET, ATM). Essa estrutura geral tem sido fundamental para a onipresença da Internet: ao manter a camada IP, com a qual todos devem concordar, mínima, milhares de flores puderam florescer tanto acima quanto abaixo. Essa é agora uma estratégia amplamente compreendida por qualquer plataforma que tente alcançar a adoção universal.

Mas algo mais aconteceu nos últimos 30 anos. Ao não abordar todos os problemas que a internet eventualmente enfrentaria à medida que crescia (por exemplo, segurança, congestionamento, mobilidade, capacidade de resposta em tempo real e assim por diante), tornou-se necessário introduzir uma série de recursos adicionais na arquitetura da internet. Ter endereços IP universais e o modelo de serviço de melhor esforço era

uma condição necessária para a adoção, mas não uma base suficiente para todas as aplicações que as pessoas queriam construir.

Ainda estamos para ver algumas dessas soluções — capítulos futuros descreverão como a Internet gerencia o congestionamento (Capítulo 6), fornece segurança (Capítulo 8) e suporta aplicativos multimídia em tempo real (Capítulos 7 e 9) — mas é informativo aproveitar esta oportunidade para reconciliar o valor de uma cintura estreita universal com a evolução que inevitavelmente acontece em qualquer sistema de longa duração: o "ponto fixo" em torno do qual o resto da arquitetura evolui mudou-se para um novo ponto na pilha de software. Em suma, o HTTP se tornou a nova cintura estreita; a única parte compartilhada/assumida da infraestrutura global que torna todo o resto possível. Isso não aconteceu da noite para o dia ou por proclamação, embora alguns tenham previsto que aconteceria. A cintura estreita subiu lentamente na pilha de protocolos como consequência de uma evolução (para misturar geociências e metáforas biológicas).

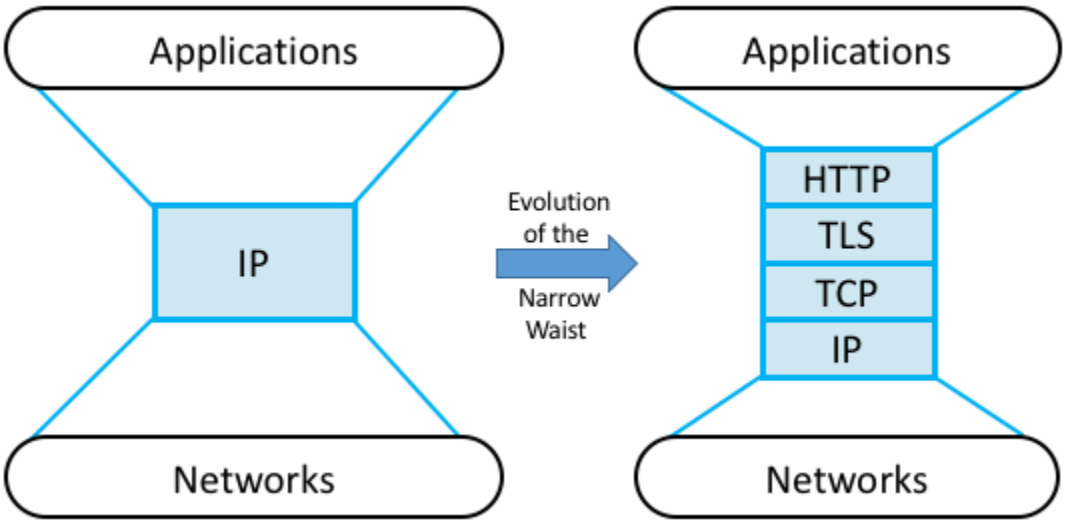

Figura 151. *HTTP (mais TLS, TCP e IP) formando a cintura estreita da arquitetura da Internet atual.*

Colocar o rótulo de "cintura estreita" exclusivamente no HTTP é uma simplificação exagerada. Na verdade, é um esforço conjunto, com a combinação HTTP/TLS/TCP/IP servindo agora como a plataforma comum da internet.

- O HTTP fornece identificadores globais de objetos (URIs) e uma interface GET/PUT simples.
- O TLS fornece segurança de comunicação de ponta a ponta.
- O TCP fornece gerenciamento de conexão, transmissão confiável e controle de congestionamento.
- O IP fornece endereços de host globais e uma camada de abstração de rede.

Em outras palavras, mesmo que você tenha a liberdade de inventar seu próprio algoritmo de controle de congestionamento, o TCP resolve esse problema muito bem, então faz sentido reutilizá-lo. Da mesma forma, mesmo que você tenha a liberdade de inventar seu próprio protocolo RPC, o HTTP oferece um protocolo perfeitamente funcional (que, por vir com segurança comprovada, tem a vantagem de não ser bloqueado por firewalls corporativos), então, novamente, faz sentido reutilizá-lo em vez de reinventar a roda.

De forma um pouco menos óbvia, o HTTP também fornece uma boa base para lidar com a mobilidade. Se o recurso que você deseja acessar foi movido, você pode fazer com que o HTTP retorne uma *resposta de redirecionamento* que aponta o cliente para um novo local. Da mesma forma, o HTTP permite a injeção de *proxies de cache* entre o cliente e o servidor, possibilitando a replicação de conteúdo popular em vários locais e poupando os clientes do atraso de percorrer toda a Internet para recuperar alguma informação. (Ambas as capacidades são discutidas na Seção 9.1 .) Por fim, o HTTP tem sido usado para fornecer multimídia em tempo real, em uma abordagem conhecida como *streaming adaptativo* . (Veja como na Seção 7.2 .)

Perspectiva mais ampla

Para continuar lendo sobre a cloudificação da Internet, consulte Perspectiva: Engenharia de tráfego definida por software .

Para saber mais sobre a centralidade do HTTP, recomendamos: HTTP: Uma cintura estreita evolutiva para a Internet do futuro , janeiro de 2012.