Capítulo 7: Dados de ponta a ponta

Problema: O que fazemos com os dados?

Da perspectiva da rede, os programas aplicativos enviam mensagens entre si. Cada uma dessas mensagens é apenas uma sequência de bytes não interpretada. Da perspectiva do aplicativo, no entanto, essas mensagens contêm vários tipos de *dados* — matrizes de números inteiros, quadros de vídeo, linhas de texto, imagens digitais e assim por diante. Em outras palavras, esses bytes têm significado. Agora, consideramos o problema de como codificar melhor os diferentes tipos de dados que os programas aplicativos desejam trocar em sequências de bytes. Em muitos aspectos, isso é semelhante ao problema de codificação de sequências de bytes em sinais eletromagnéticos que vimos em um capítulo anterior.

Voltando à nossa discussão sobre codificação, existem essencialmente duas preocupações. A primeira é que o receptor consiga extrair do sinal a mesma mensagem que o transmissor enviou; este é o problema do enquadramento. A segunda é tornar a codificação o mais eficiente possível. Ambas as preocupações também estão presentes na codificação de dados de aplicação em mensagens de rede.

Para que o receptor extraia a mensagem enviada pelo transmissor, os dois lados precisam concordar com um formato de mensagem, frequentemente chamado de *formato de apresentação* . Se o remetente quiser enviar ao receptor uma matriz de inteiros, por exemplo, os dois lados precisam concordar com a aparência de cada inteiro (quantos bits ele tem, em que ordem os bytes são organizados e se o bit mais significativo vem primeiro ou por último, por exemplo) e quantos elementos estão na matriz. A primeira seção descreve várias codificações de dados de computador tradicionais, como inteiros, números de ponto flutuante, cadeias de caracteres, matrizes

e estruturas. Formatos bem estabelecidos também existem para dados multimídia: o vídeo, por exemplo, é normalmente transmitido em um dos formatos criados pelo Moving Picture Experts Group (MPEG), e as imagens estáticas geralmente são transmitidas no formato Joint Photographic Experts Group (JPEG). As questões específicas que surgem na codificação de dados multimídia são discutidas na próxima seção.

Os tipos de dados multimídia exigem que pensemos tanto na apresentação quanto *na compressão* . Os formatos conhecidos para transmissão e armazenamento de áudio e vídeo lidam com ambas as questões: garantir que o que foi gravado, fotografado ou ouvido pelo remetente possa ser interpretado corretamente pelo destinatário, e fazê-lo de forma a não sobrecarregar a rede com enormes quantidades de dados multimídia.

A compressão e, de forma mais geral, a eficiência da codificação têm uma história rica, que remonta ao trabalho pioneiro de Shannon sobre a teoria da informação na década de 1940. Na verdade, há duas forças opostas em ação aqui. Em uma direção, você gostaria da maior redundância possível nos dados para que o receptor seja capaz de extrair os dados corretos, mesmo que erros sejam introduzidos na mensagem. Os códigos de detecção e correção de erros que vimos em um capítulo anterior adicionam informações redundantes às mensagens exatamente para esse propósito. Na outra direção, gostaríamos de remover o máximo de redundância possível dos dados para que possamos codificá-los no menor número possível de bits. Acontece que os dados multimídia oferecem uma riqueza de oportunidades para compressão devido à maneira como nossos sentidos e cérebros processam sinais visuais e auditivos. Não ouvimos frequências altas tão bem quanto as mais baixas, e não percebemos detalhes finos tanto quanto a imagem maior, especialmente se a imagem estiver em movimento.

A compressão é importante para os projetistas de redes por diversos motivos, não apenas porque raramente temos largura de banda em abundância em todos os pontos

da rede. Por exemplo, a maneira como projetamos um algoritmo de compressão afeta nossa sensibilidade a dados perdidos ou atrasados e, portanto, pode influenciar o design de mecanismos de alocação de recursos e protocolos ponta a ponta. Por outro lado, se a rede subjacente não for capaz de garantir uma quantidade fixa de largura de banda durante uma videoconferência, podemos optar por projetar algoritmos de compressão que se adaptem às mudanças nas condições da rede.

Por fim, um aspecto importante tanto da formatação de apresentação quanto da compactação de dados é que elas exigem que os hosts de envio e recebimento processem cada byte de dados na mensagem. É por esse motivo que a formatação e a compactação de apresentação são, às vezes, chamadas de funções *de manipulação de dados* . Isso contrasta com a maioria dos protocolos que vimos até agora, que processam uma mensagem sem nunca consultar seu conteúdo. Devido a essa necessidade de ler, calcular e gravar cada byte de dados em uma mensagem, as manipulações de dados afetam a taxa de transferência de ponta a ponta na rede. Em alguns casos, essas manipulações podem ser o fator limitante.

7.1 Formatação de apresentação

Uma das transformações mais comuns de dados de rede é da representação usada pelo programa aplicativo para um formato adequado para transmissão por uma rede e *vice-versa* . Essa transformação é normalmente chamada de *formatação de apresentação* . Conforme ilustrado na Figura 179 , o programa emissor traduz os dados que deseja transmitir da representação que usa internamente para uma mensagem que pode ser transmitida pela rede; ou seja, os dados são *codificados* em uma mensagem. No lado receptor, o aplicativo traduz essa mensagem que chega para uma representação que ele pode então processar; ou seja, a mensagem é *decodificada* . Esse processo às vezes é chamado de *marshalling de argumentos* ou *serialização* . Essa terminologia vem do mundo da Chamada de Procedimento Remoto (RPC), onde

o cliente pensa que está invocando um procedimento com um conjunto de argumentos, mas esses argumentos são então "reunidos e ordenados de forma apropriada e eficaz" para formar uma mensagem de rede.

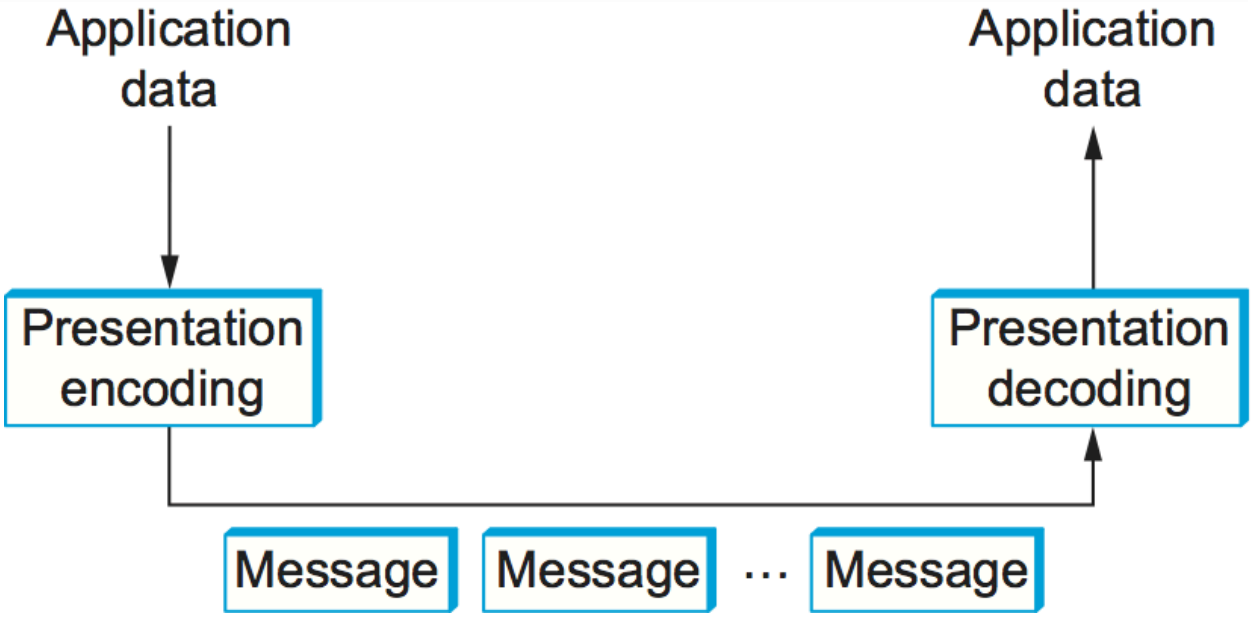

Figura 179. *A formatação da apresentação envolve a codificação e decodificação de dados do aplicativo.*

Você pode se perguntar o que torna esse problema desafiador. Um dos motivos é que os computadores representam dados de maneiras diferentes. Por exemplo, alguns computadores representam números de ponto flutuante no formato padrão IEEE 754, enquanto algumas máquinas mais antigas ainda usam seu próprio formato não padrão. Mesmo para algo tão simples como números inteiros, diferentes arquiteturas usam tamanhos diferentes (por exemplo, 16 bits, 32 bits, 64 bits). Para piorar a situação, em algumas máquinas, os números inteiros são representados na forma *big-endian* (o bit mais significativo de uma palavra — o "big end" — está no byte com o menor endereço), enquanto em outras máquinas os números inteiros são representados na forma *little-endian* (o bit menos significativo — o "little end" — está no byte com o menor endereço). Por exemplo, os processadores PowerPC são máquinas big-endian,

e a família Intel x86 é uma arquitetura little-endian. Hoje, muitas arquiteturas (por exemplo, ARM) suportam ambas as representações (e por isso são chamadas de *bi-endian*), mas a questão é que nunca se pode ter certeza de como o host com o qual se está se comunicando armazena inteiros. As representações big-endian e little-endian do inteiro 34.677.374 são apresentadas na Figura 180 .

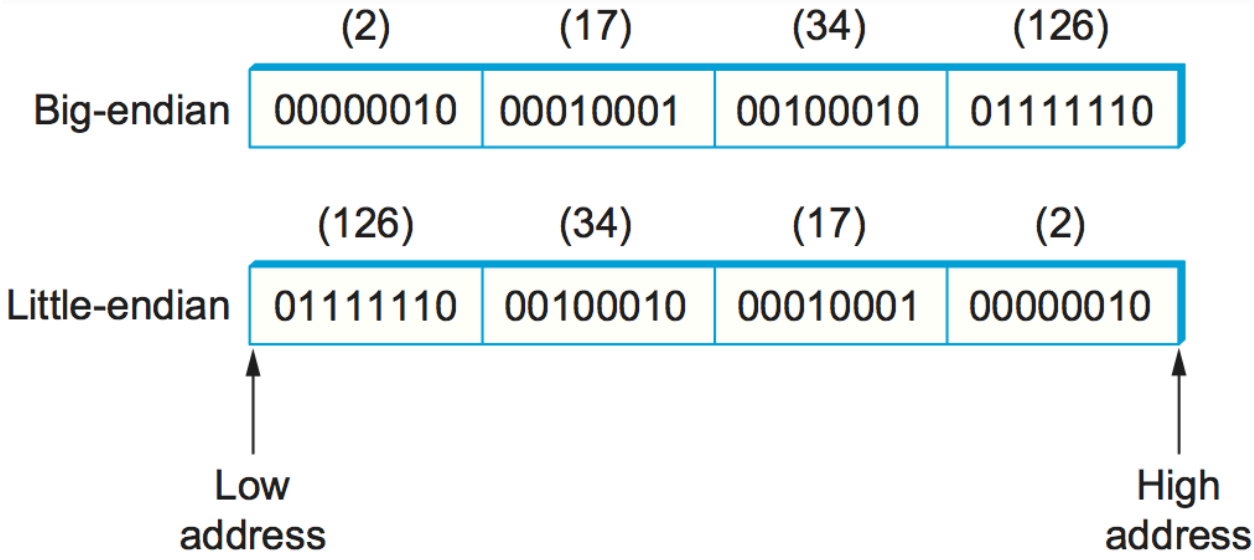

Figura 180. *Ordem de bytes big-endian e little-endian para o inteiro 34.677.374*

Outro motivo pelo qual o marshaling é difícil é que os programas aplicativos são escritos em linguagens diferentes e, mesmo quando se usa uma única linguagem, pode haver mais de um compilador. Por exemplo, os compiladores têm bastante liberdade na forma como organizam estruturas (registros) na memória, como, por exemplo, a quantidade de preenchimento que colocam entre os campos que compõem a estrutura. Portanto, não seria possível simplesmente transmitir uma estrutura de uma máquina para outra, mesmo que ambas as máquinas tivessem a mesma arquitetura e o programa fosse escrito na mesma linguagem, porque o compilador na máquina de destino poderia alinhar os campos da estrutura de forma diferente.

7.1.1 Taxonomia

Embora o agrupamento de argumentos não seja ciência de foguetes — é uma questão de ajustes de bits —, há um número surpreendente de opções de design que você deve considerar. Começamos apresentando uma taxonomia simples para sistemas de agrupamento de argumentos. A seguinte não é de forma alguma a única taxonomia viável, mas é suficiente para cobrir a maioria das alternativas interessantes.

Tipos de dados

A primeira questão é quais tipos de dados o sistema suportará. Em geral, podemos classificar os tipos suportados por um mecanismo de marshaling de argumentos em três níveis. Cada nível complica a tarefa enfrentada pelo sistema de marshaling.

No nível mais baixo, um sistema de marshaling opera em algum conjunto de *tipos base* . Normalmente, os tipos base incluem inteiros, números de ponto flutuante e caracteres. O sistema também pode suportar tipos ordinais e booleanos. Conforme descrito acima, a implicação do conjunto de tipos base é que o processo de codificação deve ser capaz de converter cada tipo base de uma representação para outra — por exemplo, converter um inteiro de big-endian para little-endian.

No próximo nível, estão *os tipos planos* — estruturas e arrays. Embora os tipos planos possam, à primeira vista, não parecer complicar o marshaling de argumentos, a realidade é que o fazem. O problema é que os compiladores usados para compilar programas aplicativos às vezes inserem preenchimento entre os campos que compõem a estrutura para alinhá-los aos limites das palavras. O sistema de marshaling normalmente *compacta* as estruturas de forma que elas não contenham preenchimento.

No nível mais alto, o sistema de marshalling pode ter que lidar com *tipos complexos* — aqueles que são construídos usando ponteiros. Ou seja, a estrutura de dados que um programa deseja enviar para outro pode não estar contida em uma única estrutura,

mas pode, em vez disso, envolver ponteiros de uma estrutura para outra. Uma árvore é um bom exemplo de um tipo complexo que envolve ponteiros. Claramente, o codificador de dados deve preparar a estrutura de dados para transmissão pela rede porque ponteiros são implementados por endereços de memória, e só porque uma estrutura reside em um determinado endereço de memória em uma máquina não significa que residirá no mesmo endereço em outra máquina. Em outras palavras, o sistema de marshalling deve *serializar* (achatar) estruturas de dados complexas.

Em resumo, dependendo da complexidade do sistema de tipos, a tarefa de agrupamento de argumentos geralmente envolve a conversão dos tipos base, o empacotamento das estruturas e a linearização das estruturas de dados complexas, tudo para formar uma mensagem contígua que pode ser transmitida pela rede. A Figura 181 ilustra essa tarefa.

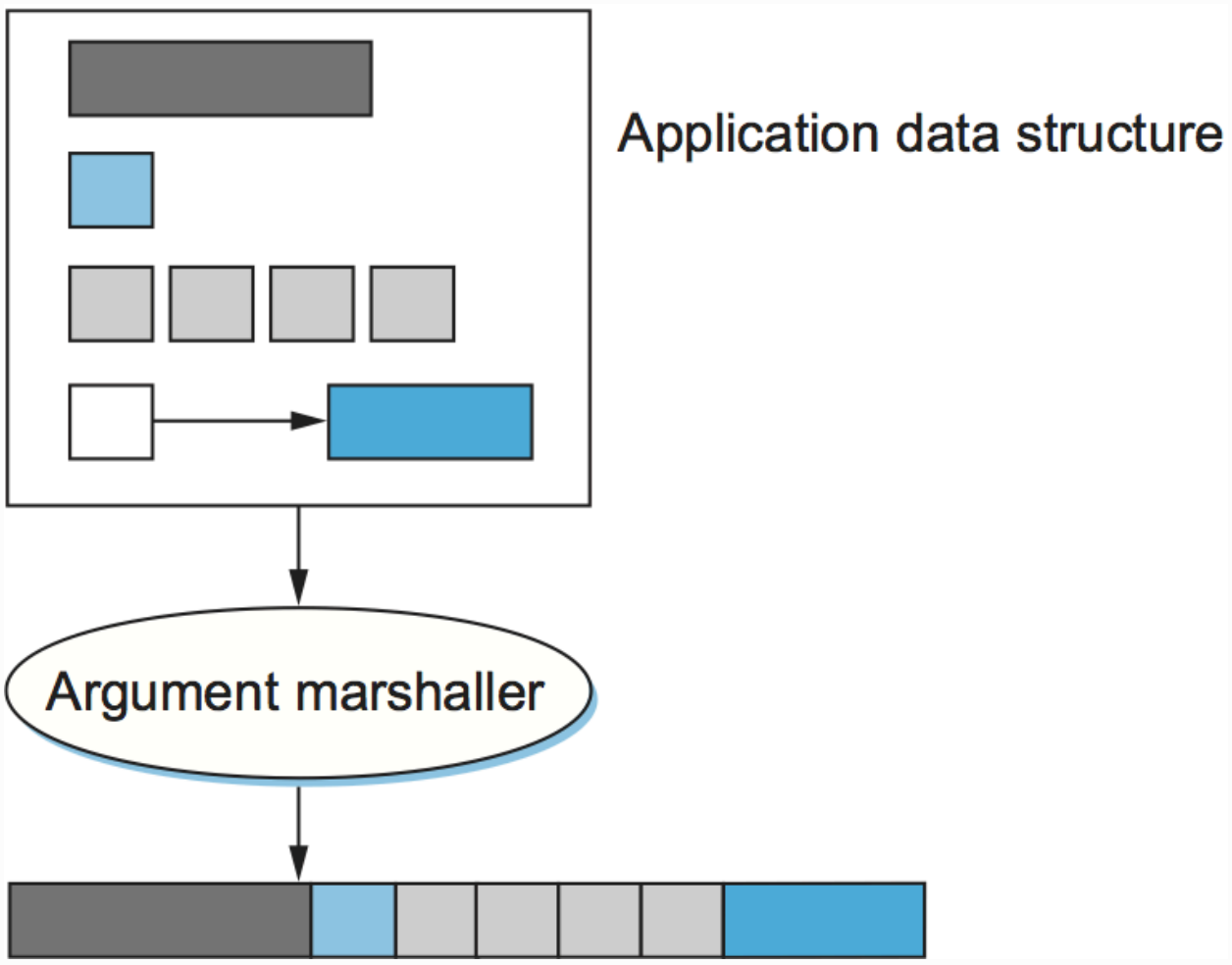

Figura 181. *Argumento de marshalling: conversão, empacotamento e linearização*

Estratégia de Conversão

Uma vez estabelecido o sistema de tipos, a próxima questão é qual estratégia de conversão o marshaller de argumentos usará. Há duas opções gerais: *forma intermediária canônica* e *receptor-faz-direito* . Consideraremos cada uma delas, separadamente.

A ideia da forma intermediária canônica é estabelecer uma representação externa para cada tipo; o host remetente traduz de sua representação interna para essa representação externa antes de enviar os dados, e o receptor traduz dessa

representação externa para sua representação local ao receber os dados. Para ilustrar a ideia, considere dados inteiros; outros tipos são tratados de maneira semelhante. Você pode declarar que o formato big-endian será usado como representação externa para inteiros. O host remetente deve traduzir cada inteiro que envia para o formato big-endian, e o host receptor deve traduzir inteiros big-endian para qualquer representação que use. (Isso é o que é feito na Internet para cabeçalhos de protocolo.) É claro que um determinado host pode já usar o formato big-endian, caso em que nenhuma conversão é necessária.

A alternativa, "recebedor-faz-direito", faz com que o remetente transmita os dados em seu próprio formato interno; o remetente não converte os tipos base, mas geralmente precisa compactar e compactar estruturas de dados mais complexas. O destinatário é então responsável por traduzir os dados do formato do remetente para seu próprio formato local. O problema com essa estratégia é que cada host deve estar preparado para converter dados de todas as outras arquiteturas de máquina. Em redes, isso é conhecido como uma *solução N por N* : cada uma das N arquiteturas de máquina deve ser capaz de lidar com todas as N arquiteturas. Em contraste, em um sistema que usa uma forma intermediária canônica, cada host precisa saber apenas como converter entre sua própria representação e uma única outra representação — a externa.

Usar um formato externo comum é claramente a coisa certa a se fazer, certo? Essa certamente tem sido a sabedoria convencional na comunidade de redes por mais de 30 anos. A resposta, no entanto, não é tão simples. Acontece que não existem muitas representações diferentes para as várias classes base ou, dito de outra forma, N não é tão grande. Além disso, o caso mais comum é que duas máquinas do mesmo tipo estejam se comunicando. Nessa situação, parece tolo traduzir dados da representação dessa arquitetura para alguma representação externa estrangeira, apenas para ter que traduzir os dados de volta para a representação da mesma arquitetura no receptor.

Uma terceira opção, embora não conheçamos nenhum sistema existente que a explore, é usar o método "receptor-faz-direito" se o remetente souber que o destino tem a mesma arquitetura; o remetente usaria alguma forma intermediária canônica se as

duas máquinas usarem arquiteturas diferentes. Como um remetente aprenderia a arquitetura do destinatário? Ele poderia obter essa informação de um servidor de nomes ou usando primeiro um caso de teste simples para verificar se o resultado apropriado ocorre.

Etiquetas

A terceira questão na manipulação de argumentos é como o receptor sabe que tipo de dados está contido na mensagem que recebe. Existem duas abordagens comuns: dados *marcados* e *não marcados* . A abordagem marcada é mais intuitiva, por isso a descreveremos primeiro.

Uma tag é qualquer informação adicional incluída em uma mensagem — além da representação concreta dos tipos base — que ajuda o receptor a decodificar a mensagem. Há várias tags possíveis que podem ser incluídas em uma mensagem. Por exemplo, cada item de dados pode ser complementado com uma tag *de tipo* . Uma tag de tipo indica que o valor a seguir é um inteiro, um número de ponto flutuante ou qualquer outro. Outro exemplo é uma tag *de comprimento* . Essa tag é usada para indicar o número de elementos em uma matriz ou o tamanho de um inteiro. Um terceiro exemplo é uma tag *de arquitetura* , que pode ser usada em conjunto com a estratégia "o receptor acerta" para especificar a arquitetura na qual os dados contidos na mensagem foram gerados. A Figura 182 mostra como um inteiro simples de 32 bits pode ser codificado em uma mensagem marcada.

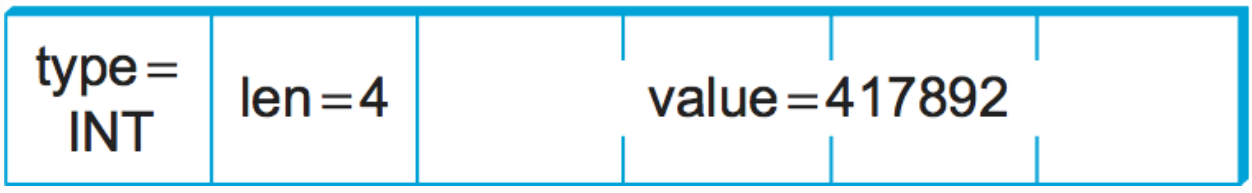

Figura 182. *Um inteiro de 32 bits codificado em uma mensagem marcada.*

A alternativa, claro, é não usar tags. Como o receptor sabe como decodificar os dados neste caso? Ele sabe porque foi programado para saber. Em outras palavras, se você chamar um procedimento remoto que recebe dois inteiros e um número de ponto flutuante como argumentos, não há razão para o procedimento remoto inspecionar tags para saber o que acabou de receber. Ele simplesmente assume que a mensagem contém dois inteiros e um número flutuante e a decodifica de acordo. Observe que, embora isso funcione na maioria dos casos, o único ponto em que falha é ao enviar matrizes de comprimento variável. Nesse caso, uma tag de comprimento é comumente usada para indicar o comprimento da matriz.

Vale ressaltar também que a abordagem sem tags significa que a formatação da apresentação é verdadeiramente de ponta a ponta. Não é possível que algum agente intermediário interprete a mensagem a menos que os dados estejam marcados. Por que um agente intermediário precisaria interpretar uma mensagem, você pode se perguntar? Coisas mais estranhas já aconteceram, principalmente devido a soluções *ad hoc* para problemas inesperados para os quais o sistema não foi projetado. Projetos de rede inadequados estão além do escopo deste livro.

Tocos

Um stub é o trecho de código que implementa o marshalling de argumentos. Stubs são normalmente usados para oferecer suporte ao RPC. No lado do cliente, o stub marshaling os argumentos do procedimento em uma mensagem que pode ser transmitida por meio do protocolo RPC. No lado do servidor, o stub converte a mensagem de volta em um conjunto de variáveis que podem ser usadas como argumentos para chamar o procedimento remoto. Stubs podem ser interpretados ou compilados.

Em uma abordagem baseada em compilação, cada procedimento possui um stub de cliente e servidor personalizado. Embora seja possível escrever stubs manualmente, eles normalmente são gerados por um compilador de stubs, com base em uma descrição da interface do procedimento. Essa situação é ilustrada na Figura 183. Como

o stub é compilado, ele geralmente é muito eficiente. Em uma abordagem baseada em interpretação, o sistema fornece stubs genéricos de cliente e servidor, cujos parâmetros são definidos por uma descrição da interface do procedimento. Como é fácil alterar essa descrição, os stubs interpretados têm a vantagem de serem flexíveis. Stubs compilados são mais comuns na prática.

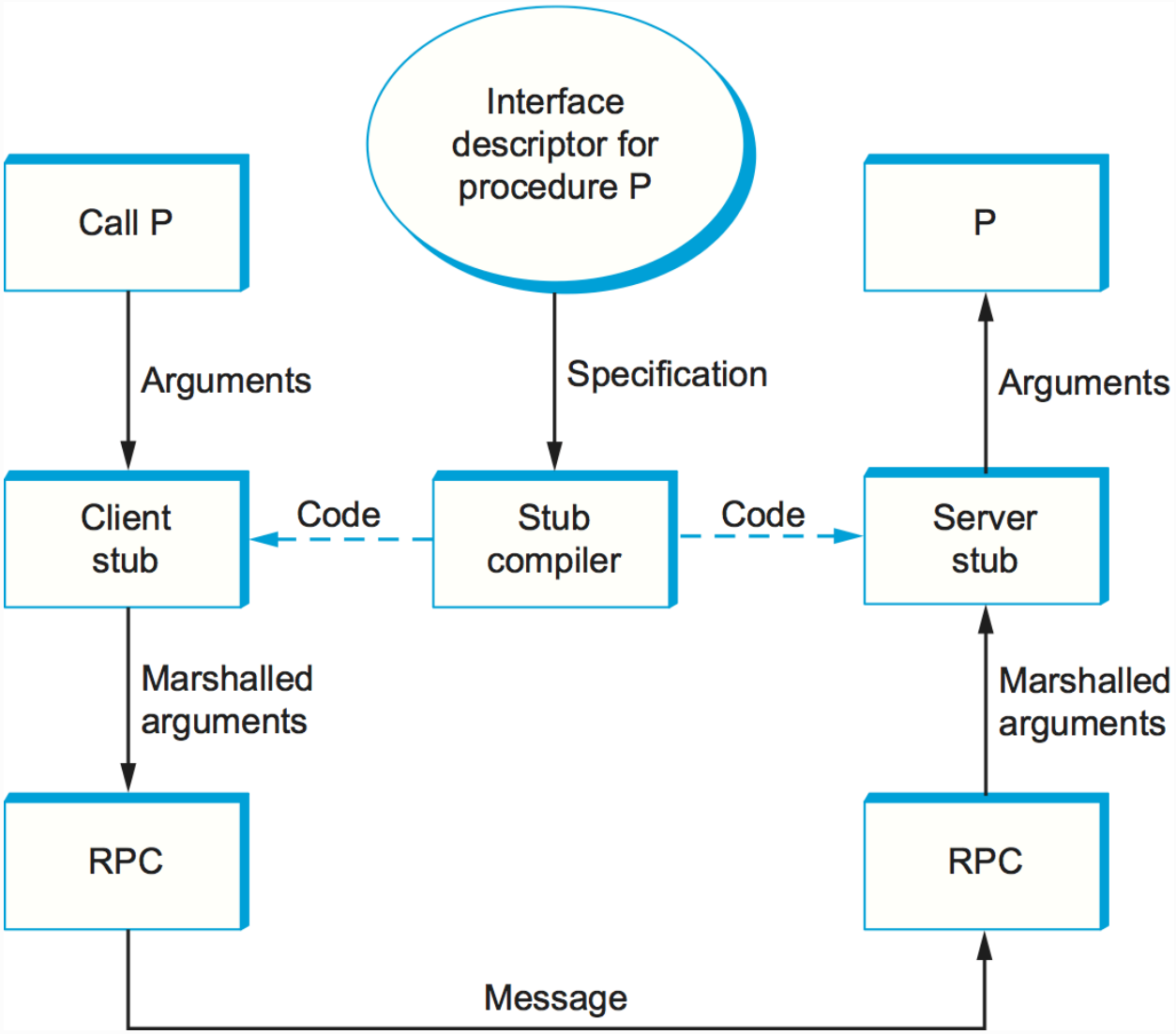

Figura 183. *O compilador stub recebe a descrição da interface como entrada e gera stubs de cliente e servidor.*

7.1.2 Exemplos (XDR, ASN.1, NDR, ProtoBufs)

Agora, descreveremos brevemente quatro representações populares de dados de rede em termos dessa taxonomia. Usamos o tipo de base inteiro para ilustrar como cada sistema funciona.

XDR

Representação Externa de Dados (XDR) é o formato de rede usado com o SunRPC. Na taxonomia recém-apresentada, XDR

- Suporta todo o sistema tipo C, com exceção de ponteiros de função
- Define uma forma intermediária canônica
- Não usa tags (exceto para indicar comprimentos de matriz)
- Usa stubs compilados

Um inteiro XDR é um item de dados de 32 bits que codifica um inteiro em C. É representado na notação de complemento de dois, com o byte mais significativo do inteiro em C no primeiro byte do inteiro XDR e o byte menos significativo do inteiro em C no quarto byte do inteiro XDR. Ou seja, o XDR usa o formato big-endian para inteiros. O XDR suporta inteiros com e sem sinal, assim como o C.

XDR representa arrays de comprimento variável especificando primeiro um inteiro sem sinal (4 bytes) que fornece o número de elementos no array, seguido por essa quantidade de elementos do tipo apropriado. XDR codifica os componentes de uma estrutura na ordem em que são declarados na estrutura. Tanto para arrays quanto para estruturas, o tamanho de cada elemento/componente é representado em um múltiplo de 4 bytes. Tipos de dados menores são preenchidos com 0s até 4 bytes. A exceção a essa regra de "preenchimento de 4 bytes" é feita para caracteres, que são codificados um por byte.

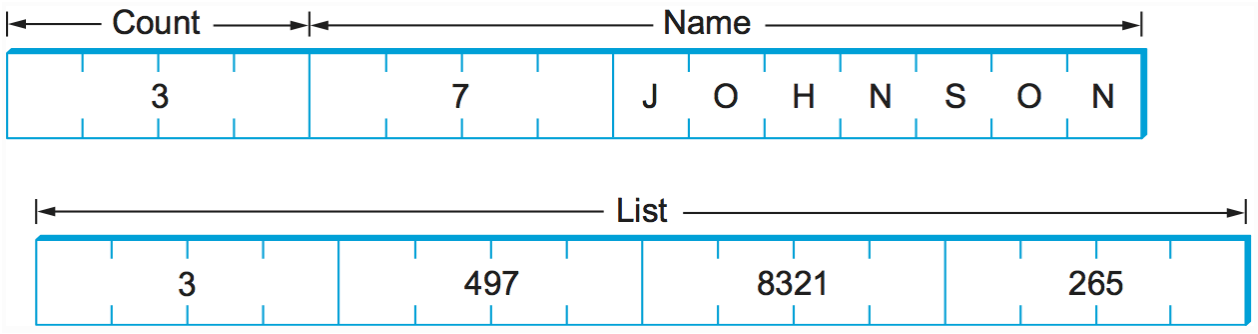

Figura 184. *Exemplo de codificação de uma estrutura em XDR.*

O fragmento de código a seguir fornece um exemplo de estrutura C (`item`) e a rotina XDR que codifica/decodifica essa estrutura (`xdr_item`). A Figura 184 descreve esquematicamente a representação on-the-wire dessa estrutura pelo XDR quando o campo `name`tem sete caracteres e a matriz `list`contém três valores.

Neste exemplo, `xdr_array`, `xdr_int`e `xdr_string`são três funções primitivas fornecidas pelo XDR para codificar e decodificar matrizes, inteiros e cadeias de caracteres, respectivamente. Argumento `xdrs`é uma variável de contexto que o XDR usa para rastrear onde está na mensagem que está sendo processada; inclui um sinalizador que indica se esta rotina está sendo usada para codificar ou decodificar a mensagem. Em outras palavras, rotinas como `xdr_item`são usadas tanto no cliente quanto no servidor. Observe que o programador do aplicativo pode escrever a rotina `xdr_item`manualmente ou usar um compilador de stub chamado `rpcgen`(não mostrado) para gerar esta rotina de codificação/decodificação. Neste último caso, `rpcgen`recebe o procedimento remoto que define a estrutura de dados `item`como entrada e gera o stub correspondente.

```
#define MAXNAME 256;


#define MAXLIST 100;
```

```
struct item {

    int      count;

    char     name[MAXNAME];

    int      list[MAXLIST];

};




bool_t

xdr_item(XDR *xdrs, struct item *ptr)

{

    return(xdr_int(xdrs, &ptr->count) &&

        xdr_string(xdrs, &ptr->name, MAXNAME) &&

        xdr_array(xdrs, &ptr->list, &ptr->count, MAXLIST,

                  sizeof(int), xdr_int));
```

```
}
```

O desempenho exato do XDR depende, é claro, da complexidade dos dados. Em um caso simples de um array de inteiros, onde cada inteiro precisa ser convertido de uma ordem de bytes para outra, são necessárias, em média, três instruções para cada byte, o que significa que a conversão de todo o array provavelmente será limitada pela largura de banda da memória da máquina. Conversões mais complexas, que exigem significativamente mais instruções por byte, serão limitadas pela CPU e, portanto, executarão a uma taxa de dados menor que a largura de banda da memória.

ASN.1

A Notação de Sintaxe Abstrata Um (ASN.1) é um padrão ISO que define, entre outras coisas, uma representação para dados enviados por uma rede. A parte específica da representação da ASN.1 é chamada de *Regras Básicas de Codificação* (BER). A ASN.1 suporta o sistema de tipos C sem ponteiros de função, define uma forma intermediária canônica e utiliza tags de tipo. Seus stubs podem ser interpretados ou compilados. Uma das razões da fama da BER da ASN.1 é que ela é usada pelo Protocolo Simples de Gerenciamento de Rede (SNMP) padrão da Internet.

ASN.1 representa cada item de dados com um triplo do formato

```
(tag, length, value)
```

O `tag`campo é normalmente de 8 bits, embora o ASN.1 permita a definição de tags multibyte. O `length`campo especifica quantos bytes compõem o campo `value`; discutiremos `length`mais a seguir. Tipos de dados compostos, como estruturas, podem ser construídos aninhando tipos primitivos, como ilustrado na Figura 185 .

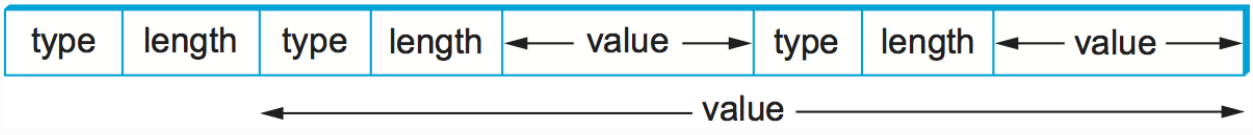

Figura 185. *Tipos compostos criados por meio de aninhamento em ASN.1 BER.*

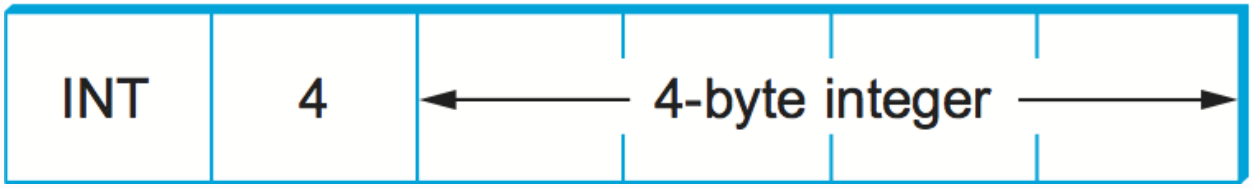

Figura 186. *Representação ASN.1 BER para um inteiro de 4 bytes.*

Se o `value`tiver 127 bytes ou menos, então o `length`será especificado em um único byte. Assim, por exemplo, um inteiro de 32 bits é codificado como um 1 byte `type`, um 1 byte `length`e os 4 bytes que codificam o inteiro, conforme ilustrado na Figura 186. O `value`próprio , no caso de um inteiro, é representado na notação de complemento de dois e na forma big-endian, assim como no XDR. Tenha em mente que, embora o `value`do inteiro seja representado exatamente da mesma maneira no XDR e no ASN.1, a representação XDR não tem nem as tags the `type`nem as `length`associadas a esse inteiro. Essas duas tags ocupam espaço na mensagem e, mais importante, exigem processamento durante o marshalling e o unmarshalling. Esta é uma das razões pelas quais o ASN.1 não é tão eficiente quanto o XDR. Outra é que o próprio fato de cada valor de dados ser precedido por um `length`campo significa que é improvável que o valor de dados caia em um limite de byte natural (por exemplo, um inteiro começando em um limite de palavra). Isso complica o processo de codificação/decodificação.

Se o campo `value`tiver 128 bytes ou mais, múltiplos bytes serão usados para especificá-lo `length`. Neste ponto, você pode estar se perguntando por que um byte pode especificar um comprimento de até 127 bytes em vez de 256. O motivo é que 1 bit do `length`campo é usado para indicar o tamanho do `length`campo. Um 0 no oitavo bit

indica um `length`campo de 1 byte. Para especificar um campo mais longo `length`, o oitavo bit é definido como 1, e os outros 7 bits indicam quantos bytes adicionais compõem o campo `length`. A Figura 187 ilustra um campo simples de 1 byte `length`e um multibyte `length`.

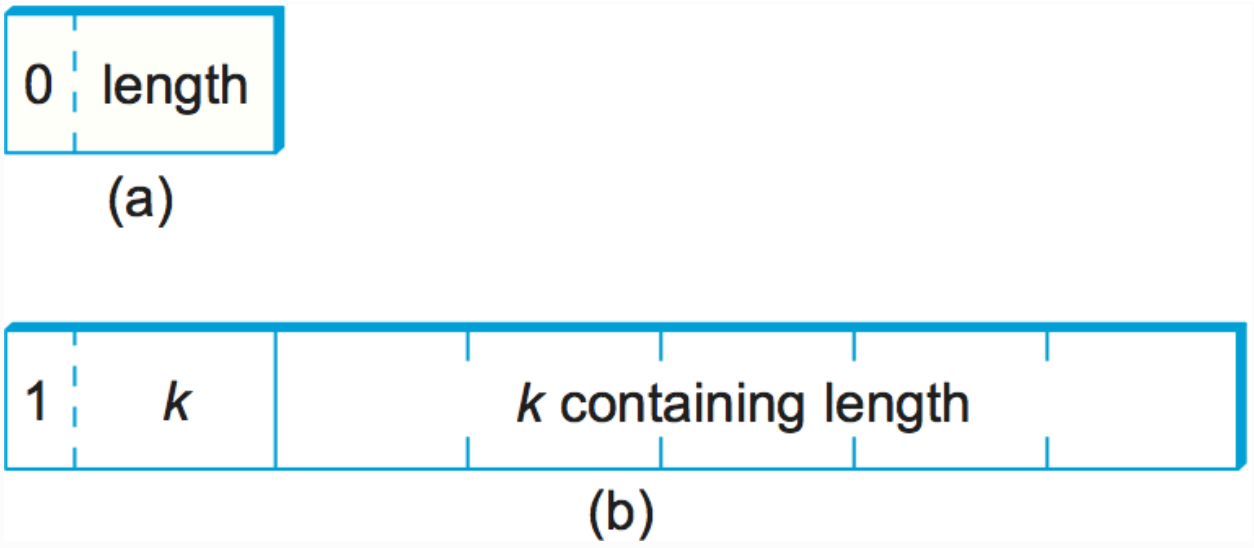

Figura 187. *Representação ASN.1 BER para comprimento: (a) 1 byte; (b) multibyte.*

NDR

A Representação de Dados em Rede (NDR) é o padrão de codificação de dados usado no Ambiente de Computação Distribuída (DCE). Ao contrário do XDR e do ASN.1, a NDR utiliza a abordagem "receiver-makes-right" (o receptor faz o direito). Ela faz isso inserindo uma tag de arquitetura no início de cada mensagem; itens de dados individuais não são marcados. A NDR usa um compilador para gerar stubs. Este compilador recebe a descrição de um programa escrito na Linguagem de Definição de Interface (IDL) e gera os stubs necessários. A IDL se parece muito com C e, portanto, suporta essencialmente o sistema do tipo C.

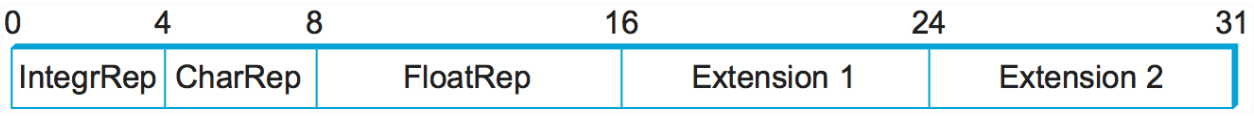

Figura 188. *Tag de arquitetura do NDR.*

A Figura 188 ilustra a tag de definição de arquitetura de 4 bytes incluída na frente de cada mensagem codificada por NDR. O primeiro byte contém dois campos de 4 bits. O primeiro campo, `IntegrRep`, define o formato para todos os inteiros contidos na mensagem. Um 0 neste campo indica inteiros big-endian e um 1 indica inteiros little-endian. O`CharRep`campo indica qual formato de caractere é usado: 0 significa ASCII (American Standard Code for Information Interchange) e 1 significa EBCDIC (uma alternativa mais antiga ao ASCII definida pela IBM). Em seguida, o`FloatRep`byte define qual representação de ponto flutuante está sendo usada: 0 significa IEEE 754, 1 significa VAX, 2 significa Cray e 3 significa IBM. Os 2 bytes finais são reservados para uso futuro. Observe que, em casos simples, como matrizes de inteiros, o NDR realiza a mesma quantidade de trabalho que o XDR e, portanto, consegue atingir o mesmo desempenho.

ProtoBufs

Buffers de Protocolo (Protobufs, abreviado) fornecem uma maneira neutra em termos de linguagem e plataforma para serializar dados estruturados, comumente usados com gRPC. Eles usam uma estratégia de marcação com uma forma intermediária canônica, onde o stub em ambos os lados é gerado a partir de um `.proto`arquivo compartilhado. Esta especificação usa uma sintaxe simples semelhante à linguagem C, como ilustra o exemplo a seguir:

```
message Person {

    required string name = 1;

    required int32 id = 2;

    optional string email = 3;
```

```
    enum PhoneType {

        MOBILE = 0;

        HOME = 1;

        WORK = 2;

    }



    message PhoneNumber {

        required string number = 1;

        optional PhoneType type = 2 [default = HOME];

    }



    required PhoneNumber phone = 4;
```

```
}
```

onde `message`poderia ser interpretado aproximadamente como equivalente a em C. O restante do exemplo é bastante intuitivo, exceto que cada campo recebe um identificador numérico para garantir exclusividade caso a especificação mude ao longo do tempo, e cada campo pode ser anotado como sendo ou .`typedef structrequiredoptional`

A maneira como os Protobufs codificam números inteiros é inovadora. Eles usam uma técnica chamada *varints* (números inteiros de comprimento variável), na qual cada byte de 8 bits usa o bit mais significativo para indicar se há mais bytes no inteiro e os sete bits inferiores para codificar a representação em complemento de dois do próximo grupo de sete bits no valor. O grupo menos significativo é o primeiro na serialização.

Isso significa que um inteiro pequeno (menor que 128) pode ser codificado em um único byte (por exemplo, o inteiro 2 é codificado como), enquanto para um inteiro maior que 128, mais bytes são necessários. Por exemplo, 365 seria codificado como`0000 0010`

```
1110 1101 0000 0010
```

Para ver isso, primeiro remova o bit mais significativo de cada byte, pois ele está lá para nos dizer se chegamos ao fim do inteiro. Neste exemplo, o `1`bit mais significativo do primeiro byte indica que há mais de um byte na variante:

```
1110 1101 0000 0010
```

```
→ 110 1101  000 0010
```

Como as variantes armazenam números com o grupo menos significativo primeiro, você inverte os dois grupos de sete bits. Em seguida, concatena-os para obter o valor final:

```
000 0010  110 1101

→  000 0010 || 110 1101

→  101101101

→  256 + 64 + 32 + 8 + 4 + 1 = 365
```

Para a especificação de mensagens mais ampla, você pode pensar no fluxo de bytes serializado como uma coleção de pares chave/valor, onde a chave (ou seja, a tag) tem duas subpartes: o identificador exclusivo do campo (ou seja, aqueles números extras no `.proto`arquivo de exemplo) e o *tipo de conexão* do valor (por exemplo, `Varint`é o único tipo de conexão de exemplo que vimos até agora). Outros tipos de conexão suportados incluem `32-bit`and `64-bit`(para inteiros de comprimento fixo) e and `length-delimited`(para strings e mensagens incorporadas). Este último informa quantos bytes de comprimento a mensagem incorporada (estrutura) tem, mas é outra `message` especificação no `.proto`arquivo que informa como interpretar esses bytes.

7.1.3 Linguagens de Marcação (XML)

Embora tenhamos discutido o problema da formatação de apresentação sob a perspectiva do RPC — ou seja, como codificar tipos de dados primitivos e estruturas de dados compostas para que possam ser enviados de um programa cliente para um programa servidor — o mesmo problema básico ocorre em outros contextos. Por exemplo, como um servidor web descreve uma página web para que qualquer número de navegadores diferentes saiba o que exibir na tela? Neste caso específico, a resposta é a Linguagem de Marcação de Hipertexto (HTML), que indica que determinadas sequências de caracteres devem ser exibidas em negrito ou itálico, que tipo e tamanho de fonte devem ser usados e onde as imagens devem ser posicionadas.

A disponibilidade de todos os tipos de aplicações e dados da Web também criou uma situação em que diferentes aplicações da Web precisam se comunicar entre si e entender os dados umas das outras. Por exemplo, um site de comércio eletrônico pode precisar se comunicar com o site de uma transportadora para permitir que um cliente rastreie uma encomenda sem precisar sair do site. Isso rapidamente começa a se assemelhar ao RPC, e a abordagem adotada na Web hoje para permitir essa comunicação entre servidores web se baseia na *Linguagem de Marcação Extensível* (XML) — uma maneira de descrever os dados que estão sendo trocados entre aplicações da Web.

Linguagens de marcação, das quais HTML e XML são exemplos, levam a abordagem de dados marcados ao extremo. Os dados são representados como texto, e as tags de texto, conhecidas como *marcação,* são intercaladas com o texto dos dados para expressar informações sobre eles. No caso do HTML, a marcação indica como o texto deve ser exibido; outras linguagens de marcação, como XML, podem expressar o tipo e a estrutura dos dados.

XML é, na verdade, uma estrutura para definir diferentes linguagens de marcação para diferentes tipos de dados. Por exemplo, XML tem sido usado para definir uma linguagem de marcação aproximadamente equivalente ao HTML, chamada *Extensible HyperText Markup Language* (XHTML). XML define uma sintaxe básica para combinar

marcação com texto de dados, mas o designer de uma linguagem de marcação específica precisa nomear e definir sua marcação. É prática comum referir-se a linguagens individuais baseadas em XML simplesmente como XML, mas enfatizaremos a distinção neste material introdutório.

A sintaxe XML se parece muito com HTML. Por exemplo, um registro de funcionário em uma linguagem hipotética baseada em XML pode se parecer com o seguinte *documento* XML , que pode ser armazenado em um arquivo chamado `employee.xml`. A primeira linha indica a versão do XML que está sendo usada, e as linhas restantes representam quatro campos que compõem o registro do funcionário, o último dos quais (`hiredate`) contém três subcampos. Em outras palavras, a sintaxe XML fornece uma estrutura aninhada de pares de tag/valor, que é equivalente a uma estrutura de árvore para os dados representados (com `employee`como raiz). Isso é semelhante à capacidade do XDR, ASN.1 e NDR de representar tipos compostos, mas em um formato que pode ser processado por programas e lido por humanos. Mais importante ainda, programas como analisadores podem ser usados em diferentes linguagens baseadas em XML, porque as definições dessas linguagens são expressas como dados legíveis por máquina que podem ser inseridos nos programas.

```
<?xml version="1.0"?>

<employee>
  <name>John Doe</name>
  <title>Head Bottle Washer</title>
  <id>123456789</id>
```

```
    <hiredate>

        <day>5</day>

        <month>June</month>

        <year>1986</year>

    </hiredate>

</employee>
```

Embora a marcação e os dados neste documento sejam altamente sugestivos para o leitor humano, é a definição da linguagem de registro de funcionários que, na verdade, determina quais tags são válidas, o que significam e quais tipos de dados implicam. Sem uma definição formal das tags, um leitor humano (ou um computador) não consegue distinguir se `1986`no `year`campo, por exemplo, há uma string, um inteiro, um inteiro sem sinal ou um número de ponto flutuante.

A definição de uma linguagem específica baseada em XML é dada por um *esquema* , que é um termo de banco de dados para uma especificação de como interpretar uma coleção de dados. Diversas linguagens de esquema foram definidas para XML; vamos nos concentrar aqui no padrão líder, conhecido pelo nome nada surpreendente de *Esquema XML* . Um esquema individual definido usando Esquema XML é conhecido como *Documento de Esquema XML* (XSD). A seguir, uma especificação XSD para o exemplo; em outras palavras, ela define a linguagem à qual o documento de exemplo

está em conformidade. Ela pode ser armazenada em um arquivo chamado `employee.xsd`.

```
<?xml version="1.0"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema">

  <element name="employee">

    <complexType>

      <sequence>

        <element name="name" type="string"/>

        <element name="title" type="string"/>

        <element name="id" type="string"/>

        <element name="hiredate">

          <complexType>

            <sequence>

              <element name="day" type="integer"/>
```

```
            <element name="month" type="string"/>

            <element name="year" type="integer"/>

          </sequence>

        </complexType>

      </element>

    </sequence>

  </complexType>

 </element>

</schema>
```

Este XSD se parece superficialmente com o nosso documento de exemplo `employee.xml`, e por um bom motivo: o XML Schema é uma linguagem baseada em XML. Há uma relação óbvia entre este XSD e o documento definido acima. Por exemplo,

```
<element name="title" type="string"/>
```

indica que o valor entre colchetes da marcação `title`deve ser interpretado como uma string. A sequência e o aninhamento dessa linha no XSD indicam que um `title`campo deve ser o segundo item em um registro de funcionário.

Ao contrário de algumas linguagens de esquema, o XML Schema fornece tipos de dados como string, inteiro, decimal e booleano. Ele permite que os tipos de dados sejam combinados em sequências ou aninhados, como em `employee.xsd`, para criar tipos de dados compostos. Portanto, um XSD define mais do que uma sintaxe; ele define seu próprio modelo de dados abstrato. Um documento em conformidade com o XSD representa uma coleção de dados em conformidade com o modelo de dados.

A importância de um XSD definir um modelo de dados abstrato e não apenas uma sintaxe é que pode haver outras maneiras além do XML de representar dados que estejam em conformidade com o modelo. E o XML, afinal, tem algumas deficiências como uma representação on-the-wire: não é tão compacto quanto outras representações de dados e é relativamente lento para analisar. Várias representações alternativas descritas como binárias estão em uso. A Organização Internacional de Padronização (ISO) publicou uma chamada *Fast Infoset* , enquanto o World Wide Web Consortium (W3C) produziu a proposta *Efficient XML Interchange* (EXI). As representações binárias sacrificam a legibilidade humana em prol de maior compactação e análise mais rápida.

Espaços para nomes XML

O XML precisa resolver um problema comum: conflitos de nomes. O problema surge porque linguagens de esquema, como o XML Schema, suportam modularidade, no sentido de que um esquema pode ser reutilizado como parte de outro. Suponha que dois XSDs sejam definidos independentemente e ambos definam o nome de marcação *idNumber* . Talvez um XSD use esse nome para identificar funcionários de uma empresa e o outro o use para identificar laptops de propriedade da empresa.

Poderíamos reutilizar esses dois XSDs em um terceiro XSD para descrever quais ativos estão associados a quais funcionários, mas, para isso, precisamos de algum mecanismo para distinguir os idNumbers dos funcionários dos idNumbers dos laptops.

A solução do XML para esse problema são *os namespaces XML* . Um namespace é uma coleção de nomes. Cada namespace XML é identificado por um Identificador Uniforme de Recursos (URI). URIs serão descritos em detalhes em um capítulo posterior; por enquanto, tudo o que você realmente precisa saber é que URIs são uma forma de identificador globalmente único. (Uma URL HTTP é um tipo específico de UNI.) Um nome de marcação simples como *idNumber pode ser adicionado a um namespace, desde que seja único dentro desse namespace. Como o namespace é globalmente único e o nome simples é único dentro do namespace, a combinação dos dois resulta em um nome qualificado* globalmente único que não pode entrar em conflito.

Um XSD geralmente especifica um *namespace de destino* com uma linha como a seguinte:

```
targetNamespace="http://www.example.com/employee"
```

é um Identificador Uniforme de Recursos, que identifica um namespace inventado. Todas as novas marcações definidas nesse XSD pertencerão a esse namespace.

Agora, se um XSD quiser referenciar nomes que foram definidos em outros XSDs, ele pode fazê-lo qualificando esses nomes com um prefixo de namespace. Esse prefixo é uma abreviação curta para o URI completo que realmente identifica o namespace. Por exemplo, a linha a seguir atribui `emp`como prefixo de namespace para o namespace do funcionário:

```
xmlns:emp="http://www.example.com/employee"
```

Qualquer marcação desse namespace seria qualificada prefixando-a com `emp:`, como `title`na linha a seguir:

```
<emp:title>Head Bottle Washer</emp:title>
```

Em outras palavras, `emp:title`é um nome qualificado, que não entrará em conflito com o nome `title`de algum outro namespace.

É impressionante como o XML é amplamente utilizado em aplicações que vão desde a comunicação no estilo RPC entre serviços web até ferramentas de produtividade de escritório e mensagens instantâneas. É certamente um dos protocolos centrais dos quais as camadas superiores da internet dependem atualmente.

7.2 Dados multimídia

Dados multimídia, compostos por áudio, vídeo e imagens estáticas, agora compõem a maior parte do tráfego na internet. Parte do que tornou possível a transmissão generalizada de multimídia através de redes são os avanços na tecnologia de compressão. Como os dados multimídia são consumidos principalmente por humanos usando seus sentidos — visão e audição — e processados pelo cérebro humano, existem desafios únicos para compactá-los. Você deve tentar manter as informações que são mais importantes para um ser humano, enquanto se livra de tudo o que não melhora a percepção humana da experiência visual ou auditiva. Portanto, tanto a ciência da computação quanto o estudo da percepção humana entram em jogo. Nesta

seção, examinaremos alguns dos principais esforços na representação e compactação de dados multimídia.

Os usos da compressão não se limitam a dados multimídia, é claro — por exemplo, você pode muito bem ter usado um utilitário como `zip`ou `compress` para compactar arquivos antes de enviá-los por uma rede, ou para descompactar um arquivo de dados após o download. Acontece que as técnicas usadas para compactar dados — que normalmente são *sem perdas* , porque a maioria das pessoas não gosta de perder dados de um arquivo — também aparecem como parte da solução para compressão multimídia. Em contraste, *a compressão com perdas* , comumente usada para dados multimídia, não garante que os dados recebidos sejam exatamente os mesmos que os dados enviados. Como observado acima, isso ocorre porque os dados multimídia geralmente contêm informações de pouca utilidade para o ser humano que as recebe. Nossos sentidos e cérebros só conseguem perceber alguns detalhes. Eles também são muito bons em preencher partes que faltam e até mesmo corrigir alguns erros no que vemos ou ouvimos. E algoritmos com perdas normalmente alcançam taxas de compressão muito melhores do que suas contrapartes sem perdas; eles podem ser uma ordem de magnitude melhores ou mais.

Para ter uma ideia da importância da compressão para a disseminação da multimídia em rede, considere o seguinte exemplo. Uma tela de TV de alta definição tem algo como 1080 × 1920 pixels, cada um com 24 bits de informação de cor, então cada quadro é

$$1080 \times 1920 \times 24 = 50\ Mb$$

Portanto, se você quiser enviar 24 quadros por segundo, isso seria mais de 1 Gbps. Isso é mais do que a maioria dos usuários da internet tem acesso. Em contraste, técnicas modernas de compressão podem levar um sinal de HDTV de qualidade

razoavelmente alta até a faixa de 10 Mbps, uma redução de duas ordens de magnitude e bem ao alcance da maioria dos usuários de banda larga. Ganhos de compressão semelhantes se aplicam a vídeos de qualidade inferior, como clipes do YouTube — os vídeos da web nunca teriam alcançado sua popularidade atual sem a compressão para fazer com que todos esses vídeos divertidos se encaixem na largura de banda das redes atuais.

As técnicas de compressão aplicadas à multimídia têm sido uma área de grande inovação, particularmente a compressão com perdas. No entanto, as técnicas sem perdas também desempenham um papel importante. De fato, a maioria das técnicas com perdas inclui algumas etapas sem perdas, portanto, iniciaremos nossa discussão com uma visão geral da compressão sem perdas.

7.2.1 Técnicas de compressão sem perdas

De muitas maneiras, a compressão é inseparável da codificação de dados. Ao pensar em como codificar um pedaço de dados em um conjunto de bits, podemos também pensar em como codificar os dados no menor conjunto de bits possível. Por exemplo, se você tem um bloco de dados que é composto por 26 símbolos de A a Z, e se todos esses símbolos têm uma chance igual de ocorrer no bloco de dados que você está codificando, então codificar cada símbolo em 5 bits é o melhor que você pode fazer (já que $2^5 = 32$ é a menor potência de 2 acima de 26). Se, no entanto, o símbolo R ocorre 50% do tempo, então seria uma boa ideia usar menos bits para codificar o R do que qualquer um dos outros símbolos. Em geral, se você sabe a probabilidade relativa de que cada símbolo ocorrerá nos dados, então você pode atribuir um número diferente de bits a cada símbolo possível de uma forma que minimize o número de bits necessários para codificar um determinado bloco de dados. Esta é a ideia essencial dos *códigos de Huffman* , um dos primeiros desenvolvimentos importantes na compressão de dados.

Codificação de comprimento de execução

A codificação por comprimento de execução (RLE) é uma técnica de compressão com a simplicidade de uma força bruta. A ideia é substituir ocorrências consecutivas de um determinado símbolo por apenas uma cópia do símbolo, além de uma contagem de quantas vezes esse símbolo ocorre — daí o nome " *comprimento de execução"* . Por exemplo, a string `AAABBCDDDD`seria codificada como `3A2B1C4D`.

A RLE se mostra útil para compactar algumas classes de imagens. Ela pode ser usada nesse contexto comparando valores de pixels adjacentes e codificando apenas as alterações. Para imagens com grandes regiões homogêneas, essa técnica é bastante eficaz. Por exemplo, não é incomum que a RLE consiga taxas de compactação da ordem de 8 para 1 para imagens de texto digitalizadas. A RLE funciona bem nesses arquivos porque eles geralmente contêm uma grande quantidade de espaço em branco que pode ser removido. Para aqueles com idade suficiente para se lembrar da tecnologia, a RLE era o principal algoritmo de compactação usado para transmitir faxes. No entanto, para imagens com até mesmo um pequeno grau de variação local, não é incomum que a compactação aumente o tamanho de bytes da imagem, já que são necessários 2 bytes para representar um único símbolo quando esse símbolo não é repetido.

Modulação por código de pulso diferencial

Outro algoritmo simples de compressão sem perdas é a Modulação por Código de Pulso Diferencial (DPCM). A ideia aqui é primeiro gerar um símbolo de referência e, em seguida, para cada símbolo nos dados, gerar a diferença entre esse símbolo e o símbolo de referência. Por exemplo, usando o símbolo A como símbolo de referência, a string `AAABBCDDDD`seria codificada como `A0001123333`porque A é o mesmo que o símbolo de referência, B tem uma diferença de 1 em relação ao símbolo de referência e assim por diante. Observe que este exemplo simples não ilustra o benefício real do DPCM, que é que, quando as diferenças são pequenas, elas podem ser codificadas com menos bits do que o próprio símbolo. Neste exemplo, o intervalo de diferenças, 0-3, pode ser representado com 2 bits cada, em vez dos 7 ou 8 bits exigidos pelo caractere

completo. Assim que a diferença se torna muito grande, um novo símbolo de referência é selecionado.

O DPCM funciona melhor que o RLE para a maioria das imagens digitais, pois aproveita o fato de que pixels adjacentes geralmente são semelhantes. Devido a essa correlação, a faixa dinâmica das diferenças entre os valores de pixels adjacentes pode ser significativamente menor que a faixa dinâmica da imagem original, e essa faixa pode, portanto, ser representada usando menos bits. Usando o DPCM, medimos taxas de compressão de 1,5 para 1 em imagens digitais. O DPCM também funciona em áudio, pois amostras adjacentes de uma forma de onda de áudio provavelmente têm valores próximos.

Uma abordagem ligeiramente diferente, chamada *codificação delta* , simplesmente codifica um símbolo como a diferença em relação ao anterior. Assim, por exemplo, `AAABBCDDDD`seria representado como `A001011000`. Observe que a codificação delta provavelmente funciona bem para codificar imagens em que pixels adjacentes são semelhantes. Também é possível executar RLE após a codificação delta, pois podemos encontrar longas sequências de 0s se houver muitos símbolos semelhantes próximos uns dos outros.

Métodos baseados em dicionário

O último método de compressão sem perdas que consideramos é a abordagem baseada em dicionário, da qual o algoritmo de compressão Lempel-Ziv (LZ) é o mais conhecido. Os comandos Unix `compress`e `gzip` .NET usam variantes do algoritmo LZ.

A ideia de um algoritmo de compressão baseado em dicionário é construir um dicionário (tabela) de strings de comprimento variável (pense nelas como frases comuns) que você espera encontrar nos dados e então substituir cada uma dessas strings quando elas aparecem nos dados pelo índice correspondente do dicionário. Por exemplo, em vez de trabalhar com caracteres individuais em dados de texto, você poderia tratar cada palavra como uma string e gerar o índice no dicionário para essa

palavra. Para elaborar mais sobre este exemplo, a palavra *compression* tem o índice 4978 em um dicionário específico; é a 4978ª palavra em `/usr/share/dict/words`. Para comprimir um corpo de texto, cada vez que a string "compression" aparece, ela seria substituída por 4978. Como este dicionário específico tem pouco mais de 25.000 palavras, seriam necessários 15 bits para codificar o índice, o que significa que a string "compression" poderia ser representada em 15 bits em vez dos 77 bits exigidos pelo ASCII de 7 bits. Esta é uma taxa de compressão de 5 para 1! Em outro ponto de dados, conseguimos obter uma taxa de compressão de 2 para 1 quando aplicamos o `compress`comando ao código-fonte dos protocolos descritos neste livro.

Claro, isso nos leva à questão de onde vem o dicionário. Uma opção é definir um dicionário estático, de preferência um que seja adaptado aos dados que estão sendo compactados. Uma solução mais geral, e a usada pela compactação LZ, é definir o dicionário de forma adaptativa com base no conteúdo dos dados que estão sendo compactados. Nesse caso, no entanto, o dicionário construído durante a compactação precisa ser enviado junto com os dados para que a metade de descompressão do algoritmo possa realizar seu trabalho. A maneira exata de construir um dicionário adaptativo tem sido objeto de extensa pesquisa.

7.2.2 Representação e compressão de imagens (GIF, JPEG)

Dado o uso ubíquo de imagens digitais — esse uso foi gerado pela invenção de displays gráficos, não de redes de alta velocidade — a necessidade de formatos de representação padrão e algoritmos de compressão para dados de imagens digitais tornou-se essencial. Em resposta a essa necessidade, a ISO definiu um formato de imagem digital conhecido como *JPEG* , em homenagem ao Joint Photographic Experts Group que o projetou. (O "Joint" em JPEG significa um esforço conjunto ISO/ITU.) JPEG é o formato mais amplamente usado para imagens estáticas em uso hoje. No centro da definição do formato está um algoritmo de compressão, que descrevemos abaixo. Muitas técnicas usadas em JPEG também aparecem em MPEG, o conjunto de

padrões para compressão e transmissão de vídeo criado pelo Moving Picture Experts Group.

Antes de nos aprofundarmos nos detalhes do JPEG, observamos que existem algumas etapas para se chegar de uma imagem digital a uma representação compactada dessa imagem que pode ser transmitida, descompactada e exibida corretamente por um receptor. Você provavelmente sabe que imagens digitais são compostas de pixels (daí os megapixels citados em anúncios de câmeras de smartphones). Cada pixel representa um local na grade bidimensional que compõe a imagem e, para imagens coloridas, cada pixel tem um valor numérico que representa uma cor. Existem muitas maneiras de representar cores, chamadas de *espaços de cores* ; a mais familiar é o RGB (vermelho, verde, azul). Você pode pensar na cor como uma grandeza tridimensional — você pode criar qualquer cor a partir de luz vermelha, verde e azul em diferentes quantidades. Em um espaço tridimensional, existem muitas maneiras diferentes e válidas de descrever um determinado ponto (considere coordenadas cartesianas e polares, por exemplo). Da mesma forma, existem várias maneiras de descrever uma cor usando três grandezas, e a alternativa mais comum ao RGB é o YUV. O Y representa a luminância, aproximadamente o brilho geral do pixel, e U e V contêm a crominância, ou informação de cor. Curiosamente, também existem algumas variantes diferentes do espaço de cores YUV. Falaremos mais sobre isso em breve.

A importância desta discussão reside no fato de que a codificação e a transmissão de imagens coloridas (estáticas ou em movimento) exigem um acordo entre as duas extremidades quanto ao espaço de cores. Caso contrário, é claro, o receptor exibiria cores diferentes das capturadas pelo emissor. Portanto, chegar a um acordo sobre uma definição de espaço de cores (e talvez uma maneira de comunicar qual espaço específico está em uso) faz parte da definição de qualquer formato de imagem ou vídeo.

Vejamos o exemplo do Graphical Interchange Format (GIF). O GIF usa o espaço de cores RGB e começa com 8 bits para representar cada uma das três dimensões da cor, totalizando 24 bits. Em vez de enviar esses 24 bits por pixel, no entanto, o GIF primeiro

reduz imagens coloridas de 24 bits para imagens coloridas de 8 bits. Isso é feito identificando as cores usadas na imagem, das quais normalmente haverá consideravelmente menos que 2^{24}, e então escolhendo as 256 cores que mais se aproximam das cores usadas na imagem. Pode haver mais de 256 cores, no entanto, então o truque é tentar não distorcer muito a cor escolhendo 256 cores de forma que nenhum pixel tenha sua cor alterada muito.

As 256 cores são armazenadas em uma tabela, que pode ser indexada com um número de 8 bits, e o valor de cada pixel é substituído pelo índice apropriado. Observe que este é um exemplo de compressão com perdas para qualquer imagem com mais de 256 cores. O GIF então executa uma variante LZ sobre o resultado, tratando sequências comuns de pixels como as strings que compõem o dicionário — uma operação sem perdas. Usando essa abordagem, o GIF às vezes consegue atingir taxas de compressão da ordem de 10:1, mas somente quando a imagem consiste em um número relativamente pequeno de cores discretas. Logotipos gráficos, por exemplo, são bem tratados pelo GIF. Imagens de cenas naturais, que geralmente incluem um espectro mais contínuo de cores, não podem ser comprimidas nessa proporção usando GIF. Também não é muito difícil para o olho humano detectar a distorção causada pela redução de cor com perdas do GIF em alguns casos.

O formato JPEG é consideravelmente mais adequado para imagens fotográficas, como seria de se esperar, dado o nome do grupo que o criou. O JPEG não reduz o número de cores como o GIF. Em vez disso, o JPEG começa transformando as cores RGB (que são as que normalmente se obtém de uma câmera digital) para o espaço YUV. A razão para isso tem a ver com a maneira como o olho percebe as imagens. Existem receptores no olho para brilho e receptores separados para cor. Como somos muito bons em perceber variações de brilho, faz sentido gastar mais bits na transmissão de informações de brilho. Como o componente Y do YUV é, aproximadamente, o brilho do pixel, podemos comprimir esse componente separadamente, e de forma menos agressiva, dos outros dois componentes (crominância).

Como observado acima, YUV e RGB são formas alternativas de descrever um ponto em um espaço tridimensional, e é possível converter de um espaço de cores para outro usando equações lineares. Para um espaço YUV comumente usado para representar imagens digitais, as equações são:

```
Y = 0.299R + 0.587G + 0.114B
```

```
U = (B-Y) x 0.565
```

```
V =  (R-Y) x 0.713
```

Os valores exatos das constantes aqui não são importantes, desde que o codificador e o decodificador concordem sobre quais são. (O decodificador terá que aplicar as transformações inversas para recuperar os componentes RGB necessários para controlar uma exibição.) As constantes são, no entanto, cuidadosamente escolhidas com base na percepção humana da cor. Você pode ver que Y, a luminância, é uma soma dos componentes vermelho, verde e azul, enquanto U e V são componentes de diferença de cor. U representa a diferença entre a luminância e o azul, e V a diferença entre a luminância e o vermelho. Você pode notar que definir R, G e B para seus valores máximos (que seriam 255 para representações de 8 bits) também produzirá um valor de Y = 255, enquanto U e V, neste caso, seriam zero. Ou seja, um pixel totalmente branco é (255,255,255) no espaço RGB e (255,0,0) no espaço YUV.

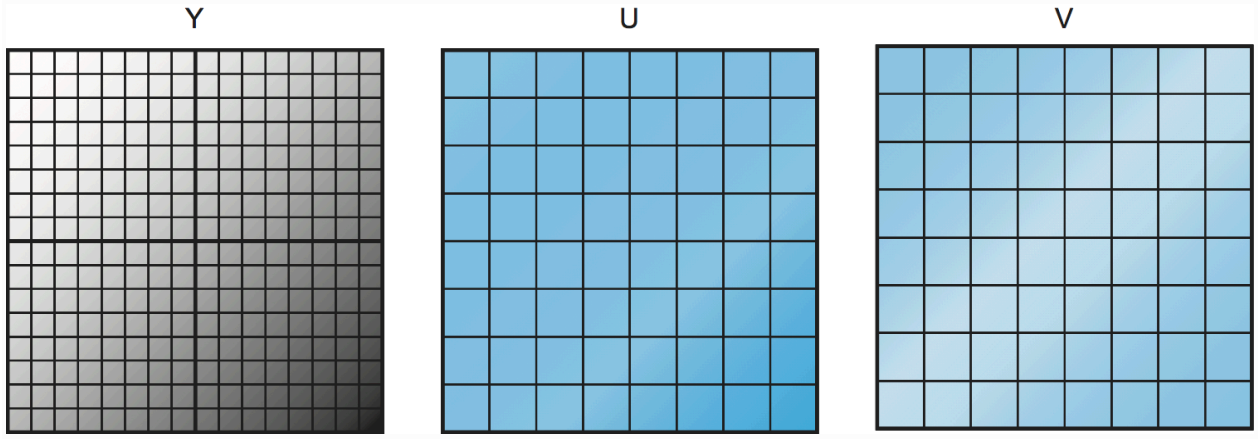

Figura 189. *Subamostragem dos componentes U e V de uma imagem.*

Uma vez que a imagem tenha sido transformada no espaço YUV, podemos agora pensar em comprimir cada um dos três componentes separadamente. Queremos ser mais agressivos na compressão dos componentes U e V, aos quais os olhos humanos são menos sensíveis. Uma maneira de comprimir os componentes U e V é *subamostra-* los. A ideia básica da subamostragem é pegar um número de pixels adjacentes, calcular o valor médio de U ou V para esse grupo de pixels e transmiti-lo, em vez de enviar o valor para cada pixel. A Figura 189 ilustra o ponto. O componente de luminância (Y) não é subamostrado, então o valor Y de todos os pixels será transmitido, conforme indicado pela grade de 16 × 16 pixels à esquerda. No caso de U e V, tratamos cada grupo de quatro pixels adjacentes como um grupo, calculamos a média do valor de U ou V para esse grupo e a transmitimos. Assim, terminamos com uma grade de 8 × 8 de valores de U e V para transmitir. Portanto, neste exemplo, para cada quatro pixels, transmitimos seis valores (quatro Y e um de U e V) em vez dos 12 valores originais (quatro para cada um dos três componentes), o que representa uma redução de 50% nas informações.

Vale ressaltar que você pode ser mais ou menos agressivo na subamostragem, com aumentos correspondentes na compressão e reduções na qualidade. A abordagem de subamostragem mostrada aqui, na qual a crominância é subamostrada por um fator de

dois nas direções horizontal e vertical (e que atende pela identificação 4:2:0), coincide com a abordagem mais comum usada para JPEG e MPEG.

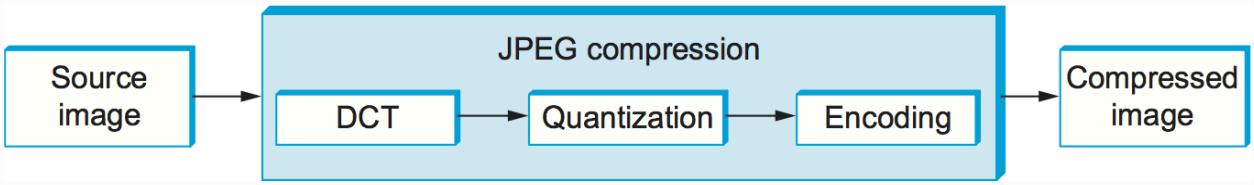

Figura 190. *Diagrama de blocos da compressão JPEG.*

Uma vez feita a subamostragem, temos agora três grades de pixels para lidar, e cada uma é tratada separadamente. A compressão JPEG de cada componente ocorre em três fases, conforme ilustrado na Figura 190. No lado da compressão, a imagem é alimentada através dessas três fases, um bloco 8 × 8 por vez. A primeira fase aplica a transformada discreta de cosseno (DCT) ao bloco. Se você pensar na imagem como um sinal no domínio espacial, então a DCT transforma esse sinal em um sinal equivalente no domínio *da frequência espacial* . Esta é uma operação sem perdas, mas um precursor necessário para a próxima etapa, com perdas. Após a DCT, a segunda fase aplica uma quantização ao sinal resultante e, ao fazê-lo, perde as informações menos significativas contidas naquele sinal. A terceira fase codifica o resultado final, mas ao fazê-lo também adiciona um elemento de compressão sem perdas à compressão com perdas alcançada pelas duas primeiras fases. A descompressão segue essas mesmas três fases, mas em ordem inversa.

Fase DCT

DCT é uma transformação intimamente relacionada à transformada rápida de Fourier (FFT). Ela recebe uma matriz 8 × 8 de valores de pixel como entrada e gera como saída uma matriz 8 × 8 de coeficientes de frequência. Você pode pensar na matriz de entrada como um sinal de 64 pontos definido em duas dimensões espaciais (*x* e *y*); a DCT divide esse sinal em 64 frequências espaciais. Para ter uma noção intuitiva da frequência espacial, imagine-se movendo-se por uma imagem, digamos, na direção *x* .

Você veria o valor de cada pixel variando como alguma função de x . Se esse valor muda lentamente com o aumento de x , então ele tem uma frequência espacial baixa; se muda rapidamente, ele tem uma frequência espacial alta. Portanto, as frequências baixas correspondem às características gerais da imagem, enquanto as frequências altas correspondem aos detalhes finos. A ideia por trás da DCT é separar as características gerais, que são essenciais para a visualização da imagem, dos detalhes finos, que são menos essenciais e, em alguns casos, podem ser mal percebidos pelo olho.

O DCT, juntamente com seu inverso, que recupera os pixels originais e durante a descompressão, são definidos pelas seguintes fórmulas:

$$DCT(i,j)=12NC(i)C(j)\sum x=0N-1\sum y=0N-1pixel(x,y)cos[(2x+1)i\pi 2N]cos[(2y+1)j\pi 2N]pixel(x,y)=12N\sum i=0N-1\sum j=0N-1C(i)C(j)DCT(i,j)cos[(2x+1)i\pi 2N]cos[(2y+1)j\pi 2N]$$

onde

$$C(x)=1/2$$

quando

$$x=0$$

e

$$1$$

quando

$$x>0$$

, e

pixel(x,y)

é o valor de escala de cinza do pixel na posição *(x,y)* no bloco 8 × 8 que está sendo compactado; N = 8 neste caso.

O primeiro coeficiente de frequência, na localização (0,0) na matriz de saída, é chamado de *coeficiente DC* . Intuitivamente, podemos ver que o coeficiente DC é uma medida do valor médio dos 64 pixels de entrada. Os outros 63 elementos da matriz de saída são chamados de *coeficientes AC* . Eles adicionam as informações de frequência espacial mais alta a esse valor médio. Assim, à medida que você vai do primeiro coeficiente de frequência em direção ao 64º coeficiente de frequência, você está se movendo de informações de baixa frequência para informações de alta frequência, dos traços amplos da imagem para detalhes cada vez mais finos. Esses coeficientes de frequência mais alta são cada vez menos importantes para a qualidade percebida da imagem. É a segunda fase do JPEG que decide qual parte de quais coeficientes descartar.

Fase de quantização

A segunda fase do JPEG é onde a compressão se torna com perdas. O DCT em si não perde informações; ele apenas transforma a imagem em um formato que facilita a identificação de quais informações remover. (Embora não seja com perdas, *por si só* , há, é claro, alguma perda de precisão durante a fase DCT devido ao uso da aritmética de ponto fixo.) A quantização é fácil de entender — é simplesmente uma questão de eliminar os bits insignificantes dos coeficientes de frequência.

Para ver como a fase de quantização funciona, imagine que você deseja comprimir alguns números inteiros menores que 100, como 45, 98, 23, 66 e 7. Se você decidir que conhecer esses números truncados para o múltiplo de 10 mais próximo é suficiente para seus propósitos, você pode dividir cada número pelo quantum 10 usando

aritmética de inteiros, resultando em 4, 9, 2, 6 e 0. Cada um desses números pode ser codificado em 4 bits em vez dos 7 bits necessários para codificar os números originais.

Quântico							
3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29

17	19	21	23	25	27	29	31

Em vez de usar o mesmo quantum para todos os 64 coeficientes, o JPEG usa uma tabela de quantização que fornece o quantum a ser usado para cada um dos coeficientes, conforme especificado na fórmula abaixo. Você pode pensar nessa tabela (`Quantum`) como um parâmetro que pode ser definido para controlar quanta informação é perdida e, correspondentemente, quanta compressão é alcançada. Na prática, o padrão JPEG especifica um conjunto de tabelas de quantização que se mostraram eficazes na compressão de imagens digitais; um exemplo de tabela de quantização é fornecido na Tabela 24 . Em tabelas como esta, os coeficientes baixos têm um quantum próximo de 1 (o que significa que pouca informação de baixa frequência é perdida) e os coeficientes altos têm valores maiores (o que significa que mais informação de alta frequência é perdida). Observe que, como resultado dessas tabelas de quantização, muitos dos coeficientes de alta frequência acabam sendo definidos como 0 após a quantização, tornando-os maduros para compressão adicional na terceira fase.

A equação básica de quantização é

```
QuantizedValue(i,j) = IntegerRound(DCT(i,j)/Quantum(i,j))
```

onde

```
IntegerRound(x) =

    Floor(x + 0.5) if x >= 0

    Floor(x - 0.5) if x < 0
```

A descompressão é então simplesmente definida como

```
DCT(i,j) = QuantizedValue(i,j) x Quantum(i,j)
```

Por exemplo, se o coeficiente DC (ou seja, DCT(0,0)) para um bloco específico fosse igual a 25, então a quantização desse valor usando a Tabela 24 resultaria em

```
Floor(25/3+0.5) = 8
```

Durante a descompressão, esse coeficiente seria restaurado como 8 × 3 = 24.

Fase de codificação

A fase final do JPEG codifica os coeficientes de frequência quantizados em um formato compacto. Isso resulta em compressão adicional, mas essa compressão é sem perdas. Começando com o coeficiente DC na posição (0,0), os coeficientes são processados na sequência em zigue-zague mostrada na Figura 191. Ao longo desse zigue-zague, uma forma de codificação de comprimento de execução é usada — RLE é aplicada apenas aos coeficientes 0, o que é significativo porque muitos dos coeficientes posteriores são 0. Os valores individuais dos coeficientes são então codificados usando um código de Huffman. (O padrão JPEG permite que o implementador use uma codificação aritmética em vez do código de Huffman.)

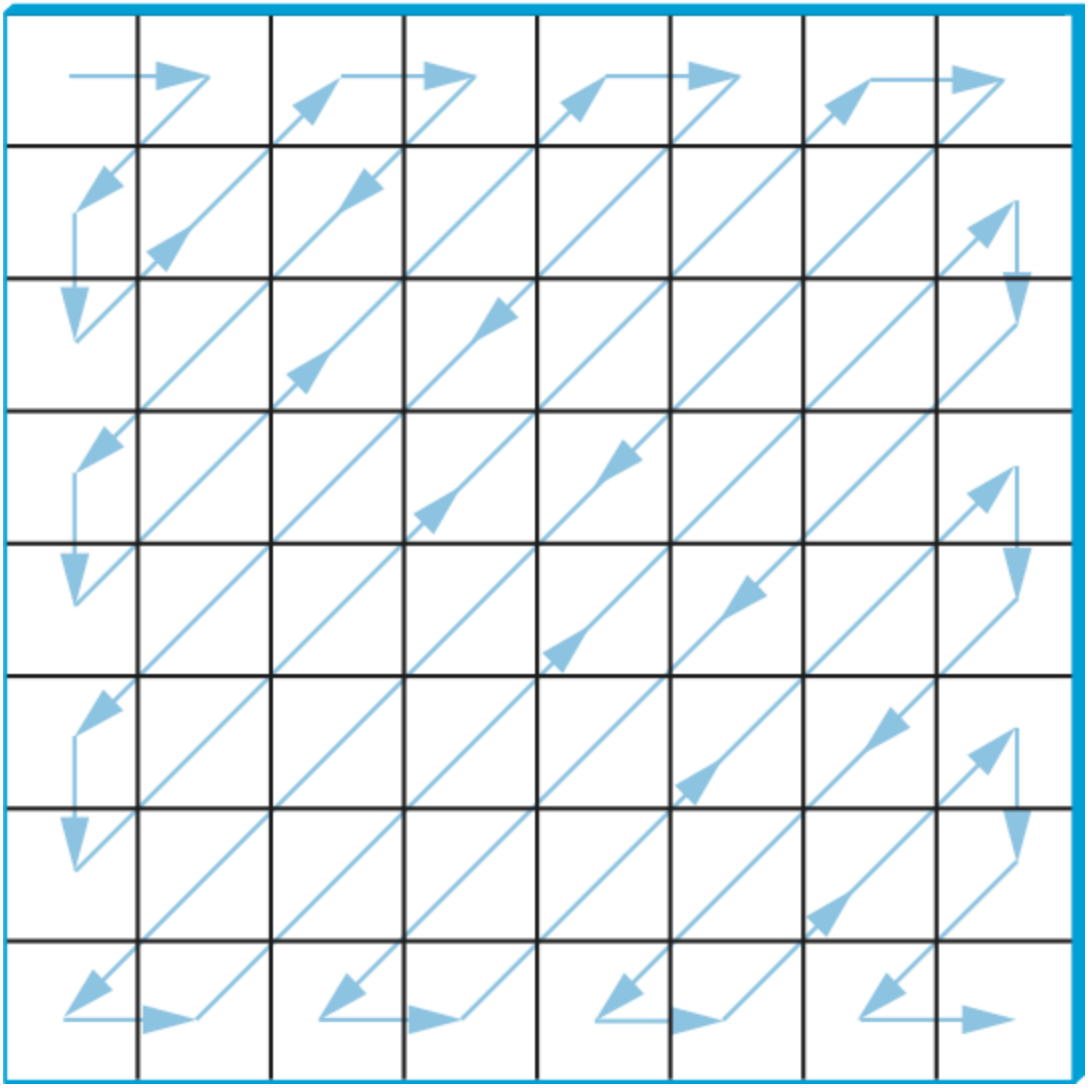

Figura 191. *Percurso em ziguezague de coeficientes de frequência quantizados.*

Além disso, como o coeficiente DC contém uma grande porcentagem das informações sobre o bloco 8 × 8 da imagem de origem, e as imagens normalmente mudam lentamente de bloco para bloco, cada coeficiente DC é codificado como a diferença em relação ao coeficiente DC anterior. Esta é a abordagem de codificação delta descrita em uma seção posterior.

O JPEG inclui diversas variações que controlam a quantidade de compressão obtida em comparação com a fidelidade da imagem. Isso pode ser feito, por exemplo, usando

diferentes tabelas de quantização. Essas variações, somadas ao fato de imagens diferentes terem características diferentes, tornam impossível determinar com precisão as taxas de compressão que podem ser alcançadas com o JPEG. Proporções de 30:1 são comuns, e proporções mais altas certamente são possíveis, mas *os artefatos* (distorção perceptível devido à compressão) tornam-se mais graves em proporções mais altas.

7.2.3 Compressão de vídeo (MPEG)

Agora, voltamos nossa atenção para o formato MPEG, nomeado em homenagem ao Moving Picture Experts Group que o definiu. Em uma primeira aproximação, uma imagem em movimento (ou seja, vídeo) é simplesmente uma sucessão de imagens estáticas — também chamadas de *quadros* ou *imagens* — exibidas em uma determinada taxa de vídeo. Cada um desses quadros pode ser compactado usando a mesma técnica baseada em DCT usada em JPEG. Parar neste ponto seria um erro, no entanto, porque isso não remove a redundância entre quadros presente em uma sequência de vídeo. Por exemplo, dois quadros sucessivos de vídeo conterão informações quase idênticas se não houver muito movimento na cena, portanto, seria desnecessário enviar as mesmas informações duas vezes. Mesmo quando há movimento, pode haver bastante redundância, já que um objeto em movimento pode não mudar de um quadro para o outro; em alguns casos, apenas sua posição muda. O MPEG leva essa redundância entre quadros em consideração. O MPEG também define um mecanismo para codificar um sinal de áudio com o vídeo, mas consideramos apenas o aspecto de vídeo do MPEG nesta seção.

Tipos de quadros

O MPEG recebe uma sequência de quadros de vídeo como entrada e os compacta em três tipos de quadros, chamados *quadros I* (intrapicture), *quadros P* (imagem prevista) e *quadros B* (imagem prevista bidirecional). Cada quadro de entrada é compactado em um desses três tipos de quadros. Os quadros I podem ser considerados quadros de referência; eles são autocontidos, não dependendo de quadros anteriores nem

posteriores. Em uma primeira aproximação, um quadro I é simplesmente a versão compactada em JPEG do quadro correspondente na fonte de vídeo. Os quadros P e B não são autocontidos; eles especificam diferenças relativas de algum quadro de referência. Mais especificamente, um quadro P especifica as diferenças em relação ao quadro I anterior, enquanto um quadro B fornece uma interpolação entre os quadros I ou P anteriores e subsequentes.

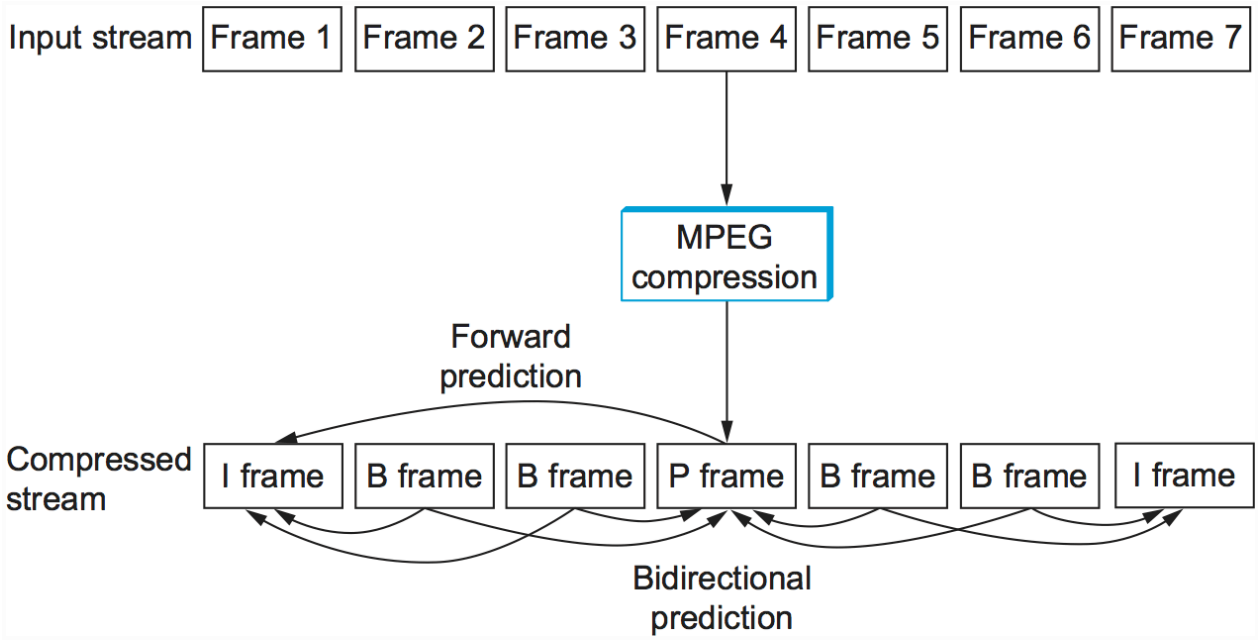

Figura 192. *Sequência de quadros I, P e B gerados por MPEG.*

A Figura 192 ilustra uma sequência de sete quadros de vídeo que, após serem comprimidos por MPEG, resultam em uma sequência de quadros I, P e B. Os dois quadros I são independentes; cada um pode ser descomprimido no receptor independentemente de quaisquer outros quadros. O quadro P depende do quadro I precedente; ele só pode ser descomprimido no receptor se o quadro I precedente também chegar. Cada um dos quadros B depende tanto do quadro I ou P precedente quanto do quadro I ou P subsequente. Ambos os quadros de referência devem chegar ao receptor antes que o MPEG possa descomprimir o quadro B para reproduzir o quadro de vídeo original.

Observe que, como cada quadro B depende de um quadro posterior na sequência, os quadros compactados não são transmitidos em ordem sequencial. Em vez disso, a sequência I BBPBB I mostrada na Figura 192 é transmitida como I PBBIB B. Além disso, o MPEG não define a proporção entre os quadros I e os quadros P e B; essa proporção pode variar dependendo da compressão necessária e da qualidade da imagem. Por exemplo, é permitido transmitir apenas quadros I. Isso seria semelhante a usar JPEG para compactar o vídeo.

Em contraste com a discussão anterior sobre JPEG, a seguinte se concentra na *decodificação* de um fluxo MPEG. É um pouco mais fácil de descrever e é a operação mais frequentemente implementada em sistemas de rede hoje em dia, visto que a codificação MPEG é tão cara que frequentemente é feita offline (ou seja, não em tempo real). Por exemplo, em um sistema de vídeo sob demanda, o vídeo seria codificado e armazenado em disco antecipadamente. Quando um espectador quisesse assistir ao vídeo, o fluxo MPEG seria então transmitido para a máquina do espectador, que decodificaria e exibiria o fluxo em tempo real.

Vamos analisar mais detalhadamente os três tipos de quadros. Como mencionado acima, os quadros I são aproximadamente iguais à versão compactada em JPEG do quadro de origem. A principal diferença é que o MPEG trabalha em unidades de macroblocos de 16 × 16. Para um vídeo colorido representado em YUV, os componentes U e V em cada macrobloco são subamostrados em um bloco de 8 × 8, como discutimos acima no contexto de JPEG. Cada subbloco 2 × 2 no macrobloco é dado por um valor U e um valor V — a média dos quatro valores de pixel. O subbloco ainda tem quatro valores Y. A relação entre um quadro e os macroblocos correspondentes é apresentada na Figura 193 .

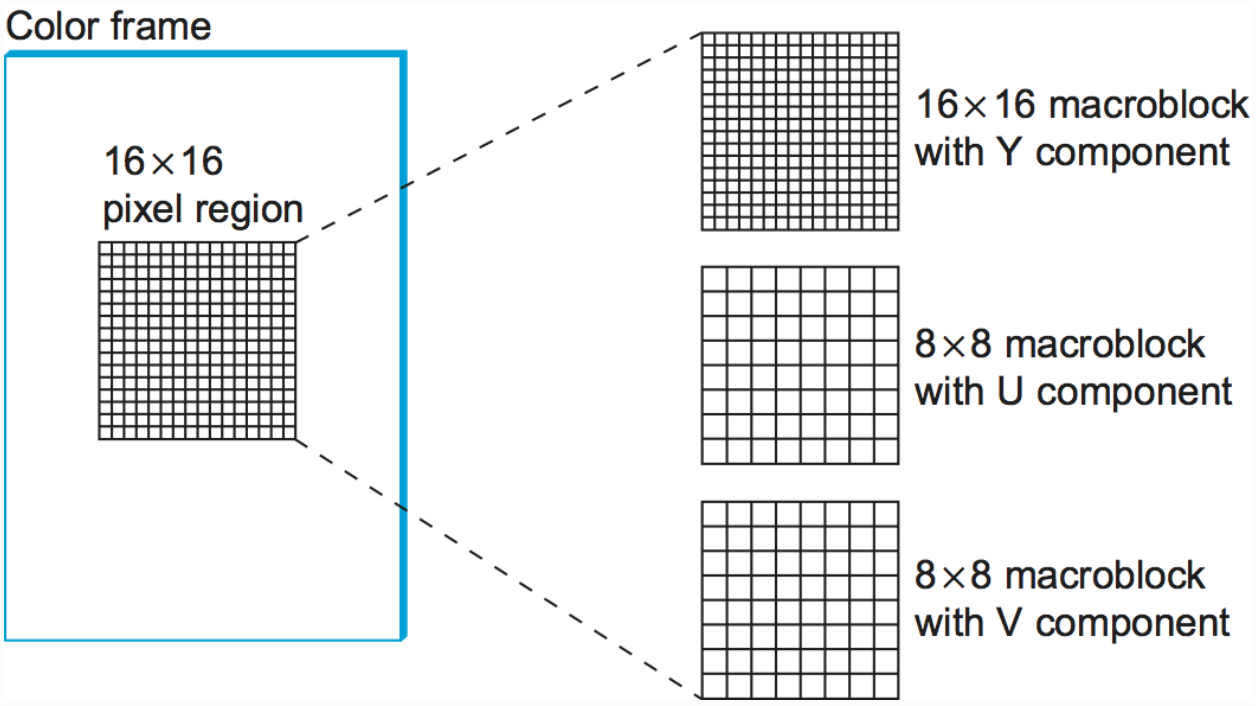

Figura 193. *Cada quadro como uma coleção de macroblocos.*

Os quadros P e B também são processados em unidades de macroblocos. Intuitivamente, podemos ver que as informações que eles carregam para cada macrobloco capturam o movimento no vídeo; ou seja, mostram em que direção e a que distância o macrobloco se moveu em relação ao(s) quadro(s) de referência. A seguir, descrevemos como um quadro B é usado para reconstruir um quadro durante a descompressão; os quadros P são tratados de maneira semelhante, exceto que dependem de apenas um quadro de referência em vez de dois.

Antes de entrarmos nos detalhes de como um quadro B é descomprimido, notamos primeiro que cada macrobloco em um quadro B não é necessariamente definido em relação a um quadro anterior e posterior, como sugerido acima, mas pode simplesmente ser especificado em relação a apenas um ou outro. De fato, um dado macrobloco em um quadro B pode usar a mesma intracodificação usada em um quadro I. Essa flexibilidade existe porque, se o filme estiver mudando muito rapidamente, às vezes faz sentido fornecer a codificação intrapicture em vez de uma codificação

prevista para frente ou para trás. Assim, cada macrobloco em um quadro B inclui um campo de tipo que indica qual codificação é usada para aquele macrobloco. Na discussão a seguir, no entanto, consideramos apenas o caso geral em que o macrobloco usa codificação preditiva bidirecional.

Nesse caso, cada macrobloco em um quadro B é representado com uma tupla 4: (1) uma coordenada para o macrobloco no quadro, (2) um vetor de movimento em relação ao quadro de referência anterior, (3) um vetor de movimento em relação ao quadro de referência subsequente e (4) um delta (

$$\delta$$

) para cada pixel no macrobloco (ou seja, o quanto cada pixel mudou em relação aos dois pixels de referência). Para cada pixel no macrobloco, a primeira tarefa é encontrar o pixel de referência correspondente nos quadros de referência passado e futuro. Isso é feito usando os dois vetores de movimento associados ao macrobloco. Então, o delta para o pixel é adicionado à média desses dois pixels de referência. Em termos mais precisos, se deixarmos F_p e F_f denotarem os quadros de referência passado e futuro, respectivamente, e os vetores de movimento passado/futuro forem dados por (x_p, y_p) e (x_f, y_f), então o pixel na coordenada *(x,y)* no quadro atual (denotado F_c) é computado como

$$Fc(x,y)=(Fp(x+xp,y+yp)+Ff(x+xf,y+yf))/2+\delta(x,y)$$

onde

$$\delta$$

é o delta para o pixel, conforme especificado no quadro B. Esses deltas são codificados da mesma forma que os pixels nos quadros I; ou seja, passam por DCT e depois são quantizados. Como os deltas são tipicamente pequenos, a maioria dos coeficientes DCT é 0 após a quantização; portanto, podem ser efetivamente comprimidos.

A discussão anterior deve ter deixado bastante claro como a codificação seria realizada, com uma exceção. Ao gerar um quadro B ou P durante a compressão, o MPEG deve decidir onde posicionar os macroblocos. Lembre-se de que cada macrobloco em um quadro P, por exemplo, é definido em relação a um macrobloco em um quadro I, mas o macrobloco no quadro P não precisa estar na mesma parte do quadro que o macrobloco correspondente no quadro I — a diferença de posição é dada pelo vetor de movimento. Você gostaria de escolher um vetor de movimento que tornasse o macrobloco no quadro P o mais semelhante possível ao macrobloco correspondente no quadro I, de modo que os deltas para esse macrobloco pudessem ser os menores possíveis. Isso significa que você precisa descobrir onde os objetos na imagem se moveram de um quadro para o próximo. Este é o problema da *estimativa de movimento* , e várias técnicas (heurísticas) para resolver este problema são conhecidas. (Discutiremos artigos que consideram este problema no final deste capítulo.) A dificuldade deste problema é uma das razões pelas quais a codificação MPEG demora mais do que a decodificação em hardware equivalente. O MPEG não especifica nenhuma técnica específica; ele apenas define o formato para codificação dessas informações nos quadros B e P e o algoritmo para reconstrução do pixel durante a descompressão, conforme fornecido acima.

Eficácia e Desempenho

O MPEG normalmente atinge uma taxa de compressão de 90:1, embora taxas de até 150:1 não sejam incomuns. Em termos dos tipos de quadros individuais, podemos esperar uma taxa de compressão de aproximadamente 30:1 para os quadros I (isso é consistente com as taxas obtidas usando JPEG quando a cor de 24 bits é primeiro reduzida para a cor de 8 bits), enquanto as taxas de compressão dos quadros P e B são normalmente de três a cinco vezes menores do que as taxas para o quadro I. Sem primeiro reduzir os 24 bits de cor para 8 bits, a compressão alcançável com MPEG fica normalmente entre 30:1 e 50:1.

O MPEG envolve um processamento caro. A compressão é normalmente feita offline, o que não é um problema para preparar filmes para um serviço de vídeo sob demanda.

Atualmente, o vídeo pode ser comprimido em tempo real usando hardware, mas implementações em software estão rapidamente preenchendo essa lacuna. Quanto à descompressão, placas de vídeo MPEG de baixo custo estão disponíveis, mas elas fazem pouco mais do que a busca de cores YUV, que felizmente é a etapa mais cara. A maior parte da decodificação MPEG é feita em software. Nos últimos anos, os processadores se tornaram rápidos o suficiente para acompanhar as taxas de vídeo de 30 quadros por segundo ao decodificar fluxos MPEG puramente em software — os processadores modernos podem até decodificar fluxos MPEG de vídeo de alta definição (HDTV).

Padrões de codificação de vídeo

Concluímos observando que o MPEG é um padrão em evolução e de complexidade significativa. Essa complexidade advém do desejo de dar ao algoritmo de codificação todo o grau possível de liberdade na forma como codifica um determinado fluxo de vídeo, resultando em diferentes taxas de transmissão de vídeo. Também advém da evolução do padrão ao longo do tempo, com o Moving Picture Experts Group trabalhando arduamente para manter a compatibilidade com versões anteriores (por exemplo, MPEG-1, MPEG-2, MPEG-4). O que descrevemos neste livro são as ideias essenciais subjacentes à compressão baseada em MPEG, mas certamente não todas as complexidades envolvidas em um padrão internacional.

Além disso, o MPEG não é o único padrão disponível para codificação de vídeo. Por exemplo, a ITU-T também definiu a *série H* para codificação de dados multimídia em tempo real. Geralmente, a série H inclui padrões para vídeo, áudio, controle e multiplexação (por exemplo, mixagem de áudio, vídeo e dados em um único fluxo de bits). Dentro da série, o H.261 e o H.263 foram os padrões de codificação de vídeo de primeira e segunda geração. Em princípio, tanto o H.261 quanto o H.263 se parecem muito com o MPEG: utilizam DCT, quantização e compressão entre quadros. As diferenças entre o H.261/H.263 e o MPEG estão nos detalhes.

Hoje, uma parceria entre a ITU-T e o grupo MPEG levou ao padrão conjunto H.264/MPEG-4, que é usado tanto para discos Blu-ray quanto por muitas fontes populares de streaming (por exemplo, YouTube, Vimeo).

7.2.4 Transmitindo MPEG através de uma rede

Como observamos, MPEG e JPEG não são apenas padrões de compressão, mas também definições do formato de vídeo e imagem, respectivamente. Com foco no MPEG, a primeira coisa a ter em mente é que ele define o formato de um *fluxo* de vídeo ; não especifica como esse fluxo é dividido em pacotes de rede. Portanto, o MPEG pode ser usado para vídeos armazenados em disco, bem como vídeos transmitidos por uma conexão de rede orientada a fluxo, como a fornecida pelo TCP.

O que descrevemos abaixo é chamado de *perfil principal* de um fluxo de vídeo MPEG enviado por uma rede. Podemos pensar em um perfil MPEG como análogo a uma "versão", exceto que o perfil não é explicitamente especificado em um cabeçalho MPEG; o receptor precisa deduzir o perfil a partir da combinação de campos de cabeçalho que vê.

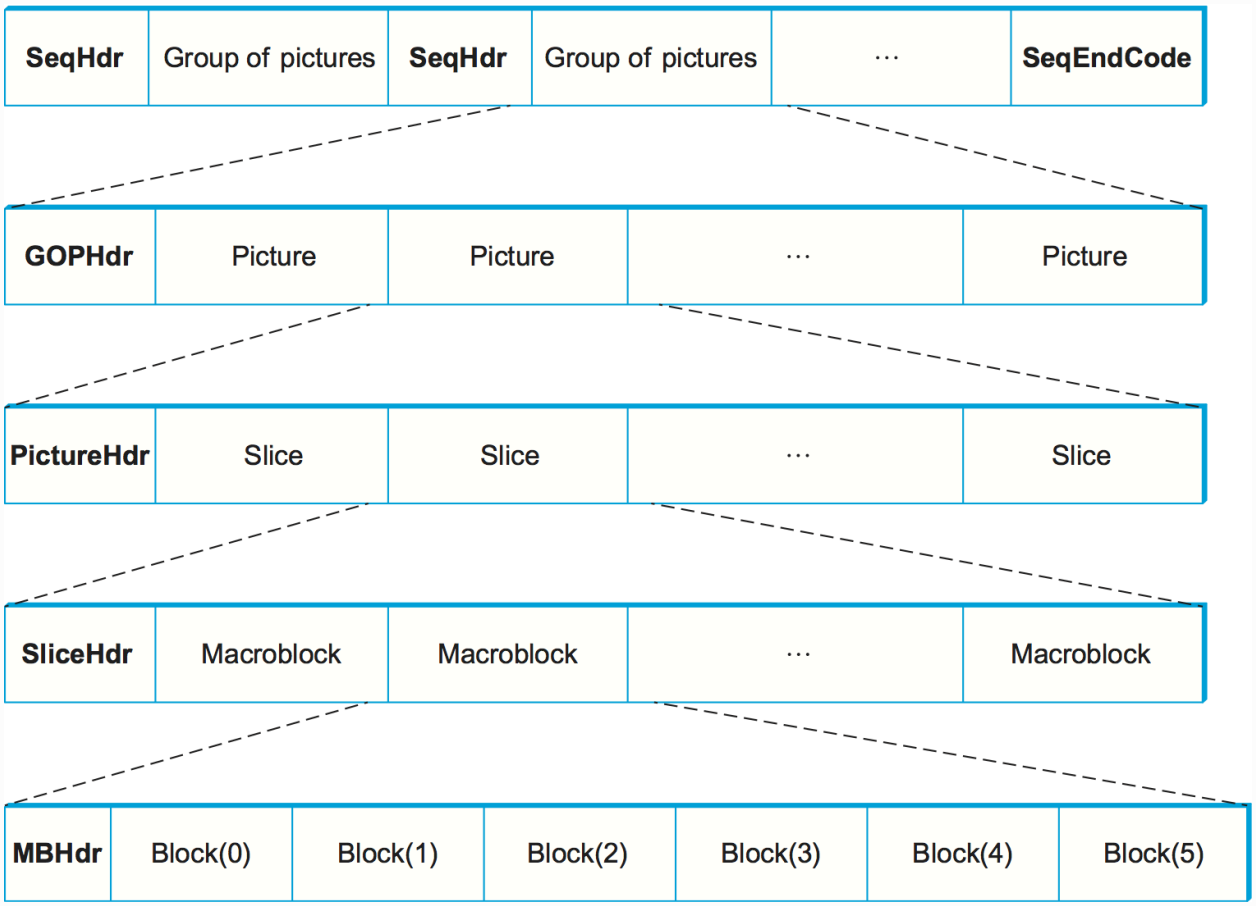

Figura 194. *Formato de um fluxo de vídeo compactado em MPEG.*

Um fluxo MPEG de perfil principal tem uma estrutura aninhada, conforme ilustrado na Figura 194. (Tenha em mente que esta figura esconde *muitos* detalhes confusos.) No nível mais externo, o vídeo contém uma sequência de grupos de imagens (GOP) separados por um `SeqHdr`. A sequência é terminada por um `SeqEndCode`(`0xb7`). O `SeqHdr` que precede cada GOP especifica — entre outras coisas — o tamanho de cada imagem (quadro) no GOP (medido em pixels e macroblocos), o período entre imagens (medido em µs) e duas matrizes de quantização para os macroblocos dentro deste GOP: uma para macroblocos intracodificados (blocos I) e uma para macroblocos intercodificados (blocos B e P). Como essas informações são fornecidas para cada GOP — em vez de uma vez para todo o fluxo de vídeo, como você poderia esperar — é possível alterar a tabela de quantização e a taxa de quadros nos limites do GOP ao

longo do vídeo. Isso torna possível adaptar o fluxo de vídeo ao longo do tempo, como discutiremos a seguir.

Cada GOP é dado por um `GOPHdr`, seguido pelo conjunto de imagens que compõem o GOP. O `GOPHdr`especifica o número de imagens no GOP, bem como informações de sincronização para o GOP (ou seja, quando o GOP deve ser reproduzido, em relação ao início do vídeo). Cada imagem, por sua vez, é dada por um `PictureHdr`e um conjunto de *fatias* que compõem a imagem. (Uma fatia é uma região da imagem, como uma linha horizontal.) O `PictureHdr`identifica o tipo da imagem (I, B ou P) e define uma tabela de quantização específica da imagem. O `SliceHdr` fornece a posição vertical da fatia, além de outra oportunidade de alterar a tabela de quantização — desta vez por um fator de escala constante em vez de fornecer uma tabela totalmente nova. Em seguida, o `SliceHdr`é seguido por uma sequência de macroblocos. Finalmente, cada macrobloco inclui um cabeçalho que especifica o endereço do bloco dentro da imagem, juntamente com dados para os seis blocos dentro do macrobloco: um para o componente U, um para o componente V e quatro para o componente Y. (Lembre-se de que o componente Y é 16 × 16, enquanto os componentes U e V são 8 × 8.)

Deve ficar claro que um dos poderes do formato MPEG é que ele dá ao codificador a oportunidade de alterar a codificação ao longo do tempo. Ele pode alterar a taxa de quadros, a resolução, a combinação de tipos de quadros que definem um GOP, a tabela de quantização e a codificação usada para macroblocos individuais. Como consequência, é possível adaptar a taxa na qual um vídeo é transmitido em uma rede trocando a qualidade da imagem pela largura de banda da rede. Exatamente como um protocolo de rede pode explorar essa adaptabilidade é atualmente objeto de pesquisa (veja a barra lateral).

Outro aspecto interessante do envio de um fluxo MPEG pela rede é exatamente como o fluxo é dividido em pacotes. Se enviado por uma conexão TCP, a empacotamento não é um problema; o TCP decide quando tem bytes suficientes para enviar o próximo datagrama IP. Ao usar vídeo interativamente, no entanto, é raro transmiti-lo por TCP,

uma vez que o TCP possui vários recursos inadequados para aplicações altamente sensíveis à latência (como mudanças abruptas de taxa após a perda de um pacote e a retransmissão de pacotes perdidos). Se estivermos transmitindo vídeo usando UDP, por exemplo, faz sentido quebrar o fluxo em pontos cuidadosamente selecionados, como nos limites de um macrobloco. Isso ocorre porque gostaríamos de limitar os efeitos de um pacote perdido a um único macrobloco, em vez de danificar vários macroblocos com uma única perda. Este é um exemplo de Enquadramento em Nível de Aplicação, que foi discutido em um capítulo anterior.

A empacotamento do fluxo é apenas o primeiro problema no envio de vídeo compactado em MPEG pela rede. A próxima complicação é lidar com a perda de pacotes. Por um lado, se um quadro B for descartado pela rede, é possível simplesmente reproduzir o quadro anterior sem comprometer seriamente o vídeo; 1 quadro em 30 não é grande coisa. Por outro lado, a perda de um quadro I tem consequências graves — nenhum dos quadros B e P subsequentes pode ser processado sem ele. Portanto, a perda de um quadro I resultaria na perda de vários quadros do vídeo. Embora seja possível retransmitir o quadro I ausente, o atraso resultante provavelmente não seria aceitável em uma videoconferência em tempo real. Uma solução para esse problema seria usar as técnicas de Serviços Diferenciados descritas no capítulo anterior para marcar os pacotes que contêm quadros I com menor probabilidade de descarte do que outros pacotes.

Uma observação final é que a forma como você escolhe codificar o vídeo depende de mais do que apenas da largura de banda de rede disponível. Depende também das restrições de latência do aplicativo. Mais uma vez, um aplicativo interativo como videoconferência precisa de latências baixas. O fator crítico é a combinação dos quadros I, P e B no GOP. Considere o seguinte GOP:

IBBBBPBBBBI

O problema que este GOP causa em um aplicativo de videoconferência é que o remetente precisa atrasar a transmissão dos quatro quadros B até que o P ou I que os

segue esteja disponível. Isso ocorre porque cada quadro B depende do quadro P ou I subsequente. Se o vídeo estiver sendo reproduzido a 15 quadros por segundo (ou seja, um quadro a cada 67 ms), isso significa que o primeiro quadro B está atrasado 4 × 67 ms, o que é mais de um quarto de segundo. Esse atraso é adicional a qualquer atraso de propagação imposto pela rede. Um quarto de segundo é muito maior do que o limite de 100 ms que os humanos são capazes de perceber. É por esse motivo que muitos aplicativos de videoconferência codificam vídeo usando JPEG, que é frequentemente chamado de motion-JPEG. (O motion-JPEG também resolve o problema de perda de um quadro de referência, já que todos os quadros podem ser independentes.) Observe, no entanto, que uma codificação entre quadros que depende apenas de quadros anteriores em vez de quadros posteriores não é um problema. Portanto, um GOP de

IPPPPI

funcionaria perfeitamente para videoconferências interativas.

Streaming Adaptável

Como esquemas de codificação como o MPEG permitem uma compensação entre a largura de banda consumida e a qualidade da imagem, existe a oportunidade de adaptar um fluxo de vídeo para corresponder à largura de banda de rede disponível. É isso que serviços de streaming de vídeo como a Netflix fazem atualmente.

Para começar, vamos supor que temos alguma maneira de medir a quantidade de capacidade livre e o nível de congestionamento ao longo de um caminho, por exemplo, observando a taxa de chegada bem-sucedida dos pacotes ao destino. À medida que a largura de banda disponível flutua, podemos enviar essa informação de volta ao codec para que ele ajuste seus parâmetros de codificação para reduzir o congestionamento e enviar mensagens de forma mais agressiva (com uma qualidade de imagem superior) quando a rede estiver ociosa. Isso é análogo ao comportamento do TCP, exceto que, no caso do vídeo, estamos, na verdade, modificando a quantidade total de dados

enviados, em vez de quanto tempo levamos para enviar uma quantidade fixa de dados, já que não queremos introduzir atraso em uma aplicação de vídeo.

No caso de serviços de vídeo sob demanda como a Netflix, não adaptamos a codificação imediatamente, mas codificamos alguns níveis de qualidade de vídeo antecipadamente e os salvamos em arquivos com os nomes correspondentes. O receptor simplesmente altera o nome do arquivo solicitado para corresponder à qualidade que suas medições indicam que a rede será capaz de fornecer. O receptor monitora sua fila de reprodução e solicita uma codificação de qualidade mais alta quando a fila fica muito cheia e uma codificação de qualidade mais baixa quando a fila fica muito vazia.

Como essa abordagem sabe para onde pular no filme caso a qualidade solicitada mude? Na verdade, o receptor nunca pede ao remetente para transmitir o filme inteiro, mas, em vez disso, solicita uma sequência de segmentos curtos do filme, normalmente com alguns segundos de duração (e sempre no limite do GOP). Cada segmento é uma oportunidade para alterar o nível de qualidade para corresponder ao que a rede é capaz de entregar. (Acontece que solicitar pedaços de filme também facilita a implementação de *trick play* , saltando de um lugar para outro no filme.) Em outras palavras, um filme é normalmente armazenado como um conjunto de N × M pedaços (arquivos): N níveis de qualidade para cada um dos M segmentos.

Há um último detalhe. Como o receptor está efetivamente solicitando uma sequência de pedaços de vídeo discretos por nome, a abordagem mais comum para emitir essas solicitações é usar HTTP. Cada pedaço é uma solicitação HTTP GET separada com a URL identificando o pedaço específico que o receptor deseja em seguida. Quando você começa a baixar um filme, seu player de vídeo primeiro baixa um arquivo *de manifesto* que contém nada mais do que as URLs para os pedaços N × M no filme e, em seguida, emite uma sequência de solicitações HTTP usando a URL apropriada para a situação. Essa abordagem geral é chamada de *streaming adaptativo HTTP* , embora tenha sido padronizada de maneiras ligeiramente diferentes por várias organizações, mais

notavelmente o DASH (*Dynamic Adaptive Streaming over HTTP) da MPEG e o HLS (HTTP Live Streaming*) da Apple .

7.2.5 Compressão de áudio (MP3)

O MPEG não apenas define como o vídeo é compactado, mas também define um padrão para compactação de áudio. Esse padrão pode ser usado para compactar a parte de áudio de um filme (nesse caso, o padrão MPEG define como o áudio compactado é intercalado com o vídeo compactado em um único fluxo MPEG) ou pode ser usado para compactar áudio independente (por exemplo, um CD de áudio). O padrão de compactação de áudio MPEG é apenas um dos muitos para compactação de áudio, mas o papel fundamental que desempenhou significa que o MP3 (que significa MPEG Layer III — veja abaixo) se tornou quase sinônimo de compactação de áudio.

Para entender a compressão de áudio, precisamos começar com os dados. Áudio com qualidade de CD, que é a representação digital *de fato* para áudio de alta qualidade, é amostrado a uma taxa de 44,1 kHz (ou seja, uma amostra é coletada aproximadamente a cada 23 μs). Cada amostra tem 16 bits, o que significa que um fluxo de áudio estéreo (2 canais) resulta em uma taxa de bits de

$$2 \times 44{,}1 \times 1000 \times 16 = 1{,}41 \ Mbps$$

Em comparação, a voz com qualidade telefônica tradicional é amostrada a uma taxa de 8 KHz, com amostras de 8 bits, resultando em uma taxa de bits de 64 kbps.

Claramente, alguma compressão será necessária para transmitir áudio com qualidade de CD por uma rede de largura de banda limitada. (Considere o fato de que o streaming de áudio em MP3 se tornou popular em uma época em que conexões de internet doméstica de 1,5 Mbps eram uma novidade.) Para piorar a situação, os overheads de sincronização e correção de erros triplicaram o número de bits

armazenados em um CD, então, se você simplesmente lesse os dados do CD e os enviasse pela rede, precisaria de 4,32 Mbps.

Assim como o vídeo, o áudio apresenta muita redundância, e a compressão se aproveita disso. Os padrões MPEG definem três níveis de compressão, conforme enumerados na Tabela 25. Destes, a Camada III, mais conhecida como MP3, foi por muitos anos a mais utilizada. Nos últimos anos, codecs de maior largura de banda proliferaram, à medida que o streaming de áudio se tornou a forma dominante de consumo de música por muitas pessoas.

Codificação	Taxas de bits	Fator de compressão
Camada I	384 kbps	14
Camada II	192 kbps	18
Camada III	128 kbps	12

Para atingir essas taxas de compressão, o MP3 utiliza técnicas semelhantes às usadas pelo MPEG para comprimir vídeo. Primeiro, ele divide o fluxo de áudio em um número de subbandas de frequência, vagamente análogo à maneira como o MPEG processa os componentes Y, U e V de um fluxo de vídeo separadamente. Segundo, cada subbanda é dividida em uma sequência de blocos, que são semelhantes aos macroblocos do MPEG, exceto que podem variar em comprimento de 64 a 1024 amostras. (O algoritmo de codificação pode variar o tamanho do bloco dependendo de

certos efeitos de distorção que estão além da nossa discussão.) Finalmente, cada bloco é transformado usando um algoritmo DCT modificado, quantizado e codificado por Huffman, assim como para o vídeo MPEG.

O segredo do MP3 está em quantas subbandas ele escolhe usar e quantos bits aloca para cada subbanda, tendo em mente que ele está tentando produzir a mais alta qualidade de áudio possível para a taxa de bits alvo. A maneira exata como essa alocação é feita é regida por modelos psicoacústicos que estão além do escopo deste livro, mas para ilustrar a ideia, considere que faz sentido alocar mais bits para subbandas de baixa frequência ao comprimir uma voz masculina e mais bits para subbandas de alta frequência ao comprimir uma voz feminina. Operacionalmente, o MP3 altera dinamicamente as tabelas de quantização usadas para cada subbanda para atingir o efeito desejado.

Uma vez compactadas, as subbandas são empacotadas em quadros de tamanho fixo, e um cabeçalho é anexado. Este cabeçalho inclui informações de sincronização, bem como as informações de alocação de bits necessárias para que o decodificador determine quantos bits são usados para codificar cada subbanda. Como mencionado acima, esses quadros de áudio podem ser intercalados com quadros de vídeo para formar um fluxo MPEG completo. Uma observação interessante é que, embora possa funcionar descartar quadros B na rede caso ocorra congestionamento, a experiência nos ensina que não é uma boa ideia descartar quadros de áudio, pois os usuários toleram melhor um vídeo ruim do que um áudio ruim.

Perspectiva: Big Data e Analytics

Este capítulo é sobre dados e, como nenhum tópico em Ciência da Computação está recebendo mais atenção do que *big data* (ou, alternativamente, *análise de dados*), uma pergunta natural é qual a relação entre big data e redes de computadores. Embora o termo seja frequentemente usado informalmente pela imprensa popular, uma definição prática é bastante simples: dados de sensores são coletados monitorando

algum sistema físico ou artificial e, em seguida, analisados para obter insights usando os métodos estatísticos de Aprendizado de Máquina. Como a quantidade de dados brutos coletados costuma ser volumosa, o qualificador "big" (grande) é aplicado. Então, há alguma implicação para redes?

À primeira vista, as redes são propositalmente projetadas para serem agnósticas em relação aos dados. Se você os coleta e deseja enviá-los para algum lugar para análise, a rede fará isso por você com prazer. Você pode compactar os dados para reduzir a largura de banda necessária para transmiti-los, mas, fora isso, o big data não é diferente dos dados comuns. No entanto, isso ignora dois fatores importantes.

A primeira é que, embora a rede não se importe com o significado dos dados (ou seja, o que os bits representam), ela se preocupa com o volume de dados. Isso afeta a rede de acesso em particular, que foi projetada para favorecer as velocidades de download em detrimento das velocidades de upload. Essa distorção faz sentido quando o caso de uso dominante é o vídeo que flui para os usuários finais, mas em um mundo onde seu carro, todos os eletrodomésticos em sua casa e os drones que sobrevoam sua cidade estão todos enviando dados de volta para a rede (carregados na nuvem), a situação se inverte. De fato, a quantidade de dados gerada por *Veículos Autônomos* e pela *Internet das Coisas (IoT)* é potencialmente avassaladora.

Embora seja possível imaginar lidar com esse problema usando um dos algoritmos de compressão descritos na Seção 7.2 , as pessoas estão, em vez disso, pensando fora da caixa e buscando novas aplicações que residem na borda da rede. Essas *aplicações nativas de borda* oferecem melhor tempo de resposta em menos de um milissegundo e reduzem drasticamente o volume de dados que, em última análise, precisa ser carregado na nuvem. Você pode pensar nessa redução de dados como uma compressão específica da aplicação, mas é mais preciso dizer que a aplicação de

borda precisa apenas gravar resumos dos dados, não os dados brutos, de volta na nuvem.

Apresentamos a tecnologia de nuvem de acesso à borda necessária para dar suporte a aplicativos nativos de borda no final do Capítulo 2 , mas o que talvez seja mais interessante é observar alguns exemplos de aplicativos nativos de borda. Um exemplo disso é que empresas nos setores automotivo, fabril e de armazéns desejam cada vez mais implantar redes 5G privadas para uma variedade de casos de uso de *automação física* . Isso inclui uma garagem onde um manobrista remoto estaciona seu carro ou um chão de fábrica usando robôs de automação. O tema comum é a conectividade de alta largura de banda e baixa latência do robô para a inteligência localizada nas proximidades em uma nuvem de borda. Isso gera custos mais baixos para robôs (você não precisa colocar computação pesada em cada um) e permite enxames de robôs e coordenação de forma mais escalável.

Outro exemplo ilustrativo é *a Assistência Cognitiva Vestível* . A ideia é generalizar o que o software de navegação faz por nós: ele usa um sensor (GPS), nos dá orientação passo a passo em uma tarefa complexa (locomoção por uma cidade desconhecida), detecta nossos erros prontamente e nos ajuda a nos recuperar. Podemos generalizar essa metáfora? Uma pessoa usando um dispositivo (por exemplo, Google Glass, Microsoft Hololens) poderia ser guiada passo a passo em uma tarefa complexa, talvez pela primeira vez? O sistema agiria efetivamente como "um anjo em seu ombro". Todos os sensores do dispositivo (por exemplo, vídeo, áudio, acelerômetro, giroscópio) são transmitidos sem fio (possivelmente após algum pré-processamento do dispositivo) para uma nuvem de borda próxima que realiza o trabalho pesado. Esta é uma metáfora do ser humano no circuito, com a "aparência e sensação da realidade aumentada", mas implementada por algoritmos de IA (por exemplo, visão computacional, reconhecimento de linguagem natural).

O segundo fator é que, como uma rede é como muitos outros sistemas artificiais, é possível coletar dados sobre seu comportamento (por exemplo, desempenho, falhas, padrões de tráfego), aplicar programas analíticos a esses dados e usar os insights obtidos para aprimorar a rede. Não é de se surpreender que esta seja uma área ativa de pesquisa, com o objetivo de construir um circuito fechado de controle. Deixando de lado a análise em si, que está bem além do escopo deste livro, as questões interessantes são: (1) quais dados úteis podemos coletar e (2) quais aspectos da rede são mais promissores para controle? Vejamos duas respostas promissoras.

Uma delas são as redes celulares 5G, que são inerentemente complexas. Elas incluem múltiplas camadas de funções virtuais, ativos RAN virtuais e físicos, uso de espectro e, como acabamos de discutir, nós de computação de ponta. É amplamente esperado que a análise de rede seja essencial para a construção de uma rede 5G flexível. Isso incluirá o planejamento da rede, que precisará decidir onde escalar funções de rede e serviços de aplicativos específicos com base em algoritmos de aprendizado de máquina que analisam a utilização da rede e os padrões de dados de tráfego.

Um segundo é *a Telemetria de Rede In-band* (INT), uma estrutura para coletar e relatar o estado da rede, diretamente no plano de dados. Isso contrasta com o relatório convencional feito pelo plano de controle da rede, como tipificado pelos sistemas de exemplo descritos na Seção 9.3 . Na arquitetura INT, os pacotes contêm campos de cabeçalho que são interpretados como "instruções de telemetria" pelos dispositivos de rede. Essas instruções informam a um dispositivo com capacidade INT qual estado coletar e gravar no pacote conforme ele transita pela rede. *As fontes de tráfego* INT (por exemplo, aplicativos, pilhas de rede do host final, hipervisores de VM) podem incorporar as instruções em pacotes de dados normais ou em pacotes de sondagem especiais. Da mesma forma, *os coletores de tráfego* INT recuperam (e, opcionalmente, relatam) os resultados coletados dessas instruções, permitindo que os coletores de tráfego monitorem o estado exato do plano de dados que os pacotes "observaram"

enquanto eram encaminhados. O INT ainda está em estágio inicial e aproveita os pipelines programáveis descritos na Seção 3.5 , mas tem o potencial de fornecer insights qualitativamente mais profundos sobre padrões de tráfego e as causas raiz das falhas de rede.

Perspectiva mais ampla

Para saber mais sobre aplicações nativas de ponta promissoras, recomendamos: Open Edge Computing Initiative , 2019.

Para saber mais sobre Telemetria de Rede em Banda, recomendamos: Telemetria de Rede em Banda via Planos de Dados Programáveis , agosto de 2015.