Sistemas Computacionais

Notas de aula

Autor

G. A., Davi; A. P., Lourenço.

Instituição

2022

Sumário

1 Sistemas Operacionais

1.1 Introdução

O Sistema Operacional (SO) é um software que gerencia os recursos de hardware e os tornam simples de serem usados por aplicações. Esse gerenciamento é feito pelo núcleo, ou `Kernel`, do SO, o qual tem permissões especiais de uso. As aplicações criadas pelos usuários se encontram no `User Space`, um local com abstrações para o fácil uso desses recursos. A mudança do espaço pode ser requisitada pela aplicação, com um processo chamado de `trap`, o qual faz referência à uma interceptação, como cair no alçapão de uma casa, tendo assim acesso ao subterrâneo, no caso o núcleo do SO. Esse processo passa pelas chamadas de sistema (System Calls), as quais são uma interface de comunicação entre o espaço de usuário e o `kernel`. As aplicações no espaço de usuário contam com as abstrações GNU libc, uma biblioteca que segue o padrão POSIX, e outros, de normas de compatibilidade entre os sistemas operacionais, tornando, assim, os programas portáteis. Outras abstrações importantes são a de processos, que são programas em execução, e a de `File System` (FS), responsável por gerenciar os arquivos, usando, por exemplo, diretórios para agrupá-los. A inicialização do sistema computacional começa pelo carregamento do BIOS (*Basic Input*

Output System), o qual é armazenado em uma memória flash (uma EEPROM, ou *Electrically Erasable and Programmable Read Only Memory*) na placa mãe, e é encarregado por tarefas básicas de gerenciamento do hardware, como o POST (*Power On Self Test*), processo incumbido por verificar o estado operacional do sistema. O BIOS também é responsável por carregar o SO, da memória secundária para a memória primária, processo que finaliza com o sistema operacional assumindo o controle do hardware. A memória primária, ou RAM (*Random Access Memory*), é uma memória volátil (não mantém os dados após o cessamento da energia) de acesso direto ao processador. Os programas necessitam estar nessa memória para serem executados. Já a memória secundária, também chamada de Drive (ou disco), passa por um sistema I/O (*Input Output*), o qual é encarregado por gerenciar a entrada e saída de dados com outros dispositivos. A Figura 1.1 mostra como as subdivisões de um sistema computacional, destacando as do sistema operacional.

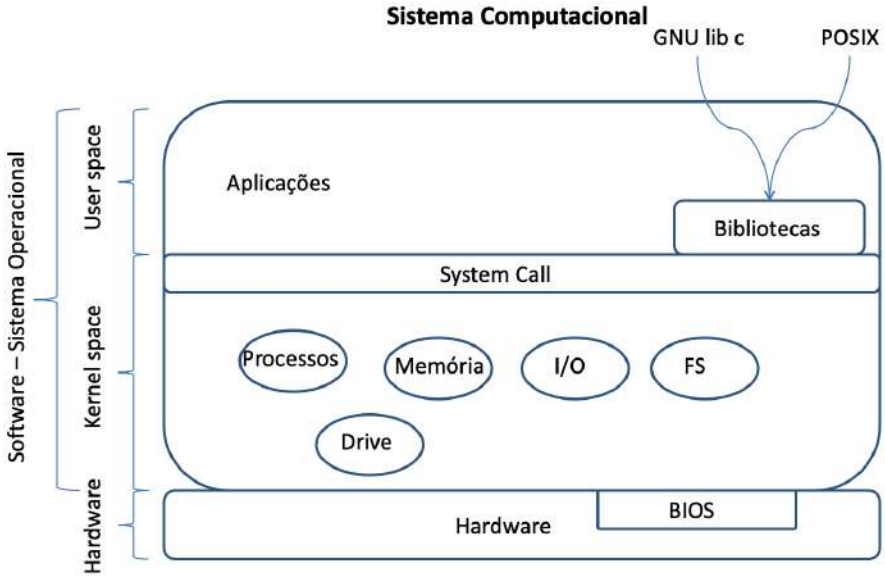

Figura 1.1: Sistema Computacional

1.2 Prática

Com o intuito de mostrar como funciona as chamadas do sistema e como funciona a compilação, vamos criar um programa `Hello world` na linguagem C. Para tal, precisamos entender alguns comandos do Linux.

Comandos Linux

1. `mkdir`: cria um novo diretório
2. `grep`: procura um texto em um arquivo
3. `rm`: remover um arquivo
4. `mv`: move o arquivo
5. `ls`: lista os elementos de um diretório
6. `ls -a`: lista arquivos
7. `ls -l`: lista permissões
8. `ls -F`: edita os nomes, como `*` no final do nome dos arquivos executáveis
9. `ls -F`: edita os nomes, como `*` no final do nome dos arquivos executáveis
10. `ll`: um alias para `ls -alF`
11. `file`: inspecionar um arquivo
12. `cd`: mudar o diretório
13. `ld`: vinculador, responsável por gerar um arquivo executável a partir de um objeto
14. `objdump`: Mostra informações sobre um arquivo objeto
15. `objdump -d`: desmota (disassembly) um arquivo objeto.

```
$ objdump -d file.o
```

16. `Vim`: Vi Improved – Editor de texto. `$vim file`

```
$ vim file
```

17. `Vi`: Editor de texto.

18. `-o`: output – resultado de um comando

19. `as`: compilador assembly

20. `gcc`: compilador GNU para a linguagem C, gera arquivo executável a partir de um arquivo `.c`.

```
$ gcc file.c -o file
```

21. `gcc -E`: gera arquivo pré-processado (`.pre`).

```
$ gcc -E file.c -o file.pre
```

22. `gcc -S`: gera arquivo assembly (`.s`)

```
$ gcc -S file.c -o file.s
```

23. `gcc -c`: converte em linguagem de máquina sem vincular, gerando um arquivo objeto não executável (`.o`)

```
$ gcc -c file.c -o file.o
```

24. `man #`: mostra o manual de uma função de alguma sessão

```
$ man 3 printf
```

25. `man man`: mostra o manual do man

26. `echo`: mostra o texto resultado na saída padrão.

27. `./` : executa um arquivo executável

1.2.1 Etapas de compilação

O processo de compilação de um código está relacionado com transformar um arquivo texto de uma linguagem específica, para um arquivo binário executável. O compilador da linguagem C usado no Linux, `gcc`, realiza esse processo em 4 etapas. A primeira dessas etapas é o pré-processamento, responsável por lidar com as diretivas de pré-processamento, como o `#include` e `#define`, que anexa um arquivo e substitui os macros, respectivamente. Tem como resultado um arquivo de extensão pre (`.pre`). A segunda etapa é a conversão do arquivo pré-processado para o assembly. Ela utiliza o *toolchain*, pacote de ferramentas contendo instruções específicas do processador de destino, como ARM. A saída é um arquivo de extensão s (`.s`). A terceira etapa é a geração de um código objeto em binário, o qual é um arquivo de extensão o (`.o`). Por fim, há a vinculação (*linking*) do código objeto gerado na etapa anterior com outros códigos objetos a fim de substituir as referências à símbolos definidos fora do código objeto original. Tem como saída um arquivo binário executável sem extensão. A Figura 1.2 mostra as etapas de compilação do gcc.

1.3 Compilando um arquivo `.c`

A seguir será mostrado um passo a passo, da criação de um arquivo `.c` até sua compilação.

Dentro do editor de texto Linux, escreva o código `Hello World` disponível a seguir.

```
1    #include <stdio.h>
2    int main() {
3        printf("Hello World!\n);
4        return 0;
5    }
```

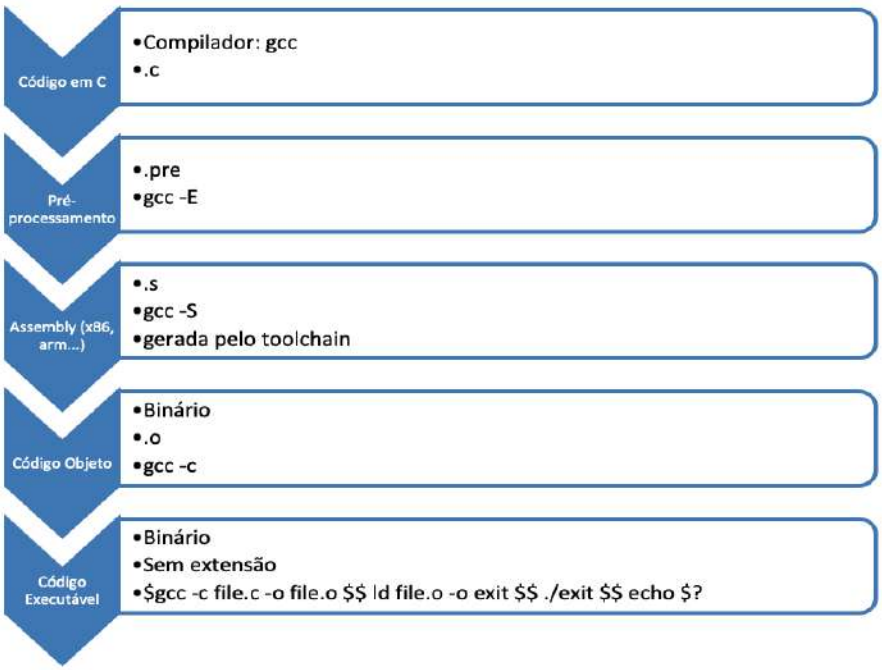

Figura 1.2: Etapas de compilação do gcc

Salve com o nome `hw.c` Abra o terminal, vá até o diretório no qual foi salvo o arquivo `hw.c`, e digite:

```
$ gcc hw.c -o hw
$ ./hw
```

Será assim gerado um arquivo `hw` executável, com sua execução em seguida.
Com o `gcc`, pode-se também gerar os arquivos das etapas explicadas anteriormente, com 1, 2 e 3 gerando os arquivos pré-processados, arquivo assembly e código objeto, respectivamente.

```
1 $ gcc -E hw.c -o hw.pre
2 $ gcc -S hw.c -o hw.s
3 $ gcc -c hw.c -o hw.o
```

É possível também ler o código objeto no formato de assembly com a função de desassembly:

1.4 Executando uma chamada do sistema

Para executar uma chamada do sistema, vamos, primeiro, criar um arquivo assembly (`.s`) chamado de `01-exit64.s` com o conteúdo abaixo:
Após isso, devemos gerar um código objeto, usando um compilador assembly:

```
$ as 01-exit64.s -o exit.o
```

Depois gerar um arquivo executável, pelo processo de vinculação:

```
$ ld exit.o -o exit
```

Para executar e visualizar, basta digitar:

```
$ ./exit
$ echo $?
```

1.4.1 Entendendo System Call

Para entender melhor o resultado gerado anteriormente, podemos acessar o manual do system call com:

```
$ man syscall
```

A primeira tabela do manual será parecida com a tabela a seguir:

Arch / ABI	Instruction	System Call	val	val 2	Error
arm64	svc #0	x8	0	x1	-
i386	int $0x80	eax	ax	edx	-
mips	syscall	v0	0	v1	a3
riscv	ecall	a7	0	a1	-
x86-64	syscall	rax	ax	rdx	-
x32	syscall	a2	2	-	-

Tabela 1.1: System Call por Arquitetura

A Tabela 1.1 mostra que para que a função syscall de um processador com arquitetura x86-64 seja chamada, devemos primeiro ajustar o registrador rax com o valor equivalente à função desejada. No exemplo dado, o valor 60 se refere a função "exit", que encerra a execução de um programa. As diferentes funções e seus respectivos valores podem ser acessados em:
`https://elixir.bootlin.com/linux/latest/source/arch/x86/entry/syscalls/syscall_64.tbl`
A Tabela 1.2 mostra que o primeiro argumento a ser usado na arquitetura x86-64 deve ser passado para o registrador rdi. A tabela completa pode ser encontrada no manual do syscall. O valor passado para esse registrador será retornado pelo syscall.

Arch / ABI	rg1	rg2	rg3	rg4	rg5	rg6	rg7	notes
x86-64	rdi	rsi	rdx	r10	r8	r9	-	

Tabela 1.2: Argumentos das chamadas do sistema

movq A função `movq`, utilizada no programa assembly, move o valor especificado para um dado registrador. No exemplo, movemos o valor 60 para o registrador `rax`, e o valor 10 para o registrador `rdi`. A instrução syscall, que, como mostrado na Tabela 1, usa o valor contido no registrador `rax`, no caso 60, para fazer referência a função a ser chamada, e, como argumento, é passado o valor 10 para o registrador `rdi` que, como pode ser visto na Tabela 2, se refere ao primeiro argumento de uma syscall. Ao ser executada, a instrução syscall bloqueia o programa e passa o controle ao sistema operacional, para que assim possa processar a função requisitada no kernel space, retornando, no final da execução, ao programa. No caso, o syscall terá como retorno o próprio argumento, o valor 10, que é visto usando a função `echo $?`. É interessante notar que, ao usar a função `objdump -d` com o arquivo `exit.o`, é possível ver que os números 60 e 10 foram substituídos por 0x3c e 0xa. O valor 0x é uma referência para hexadecimal, e os valores "3c" e "a" são os equivalentes a 60 e 10 em hexadecimal.

2 Processos

Ao visualizar um ícone de um programa na área de trabalho de um Sistema Operacional (SO), o usuário está olhando para uma referência a uma entidade passiva executável localizada na memória secundária. Ao clicar duas vezes em cima do ícone, o programa `Loader` é acionado, carregando o programa clicado para a memória principal, tornando-o uma entidade ativa chamada de processo, o qual está inserido dentro de um contexto de execução único em user mode, isolado dos demais processos. Assim, um único programa é capaz de gerar múltiplos processos, como o navegador de internet e as diversas abas abertas. De forma geral, um processo divide a memória em quatro seções:

1. Text: código executável.
2. Data: variáveis globais.
3. Heap: memória alocada dinamicamente em tempo de execução.
4. Stack: armazenamento temporário para, por exemplo, variáveis locais.

Como é mostrado na Figura 2.2, em um programa na linguagem C, as variáveis globais são separadas dentro da seção `data` por utilizadas e não utilizadas. As variáveis dentro da função `main` são alocadas na `stack`, sendo removidas,

automaticamente, após o uso. A seção `heap` é acessado pela função `malloc(3)` (*Memory Allocation*, ou alocação de memória), o qual não remove os dados armazenados após o uso, devendo o programador ter o cuidado de liberar a memória ocupada com a função `free(3)`. Por fim, o código executável referente a esse programa está armazenado na seção `text`. No programa em C, os argumentos da função main são armazenados em uma seção específica.

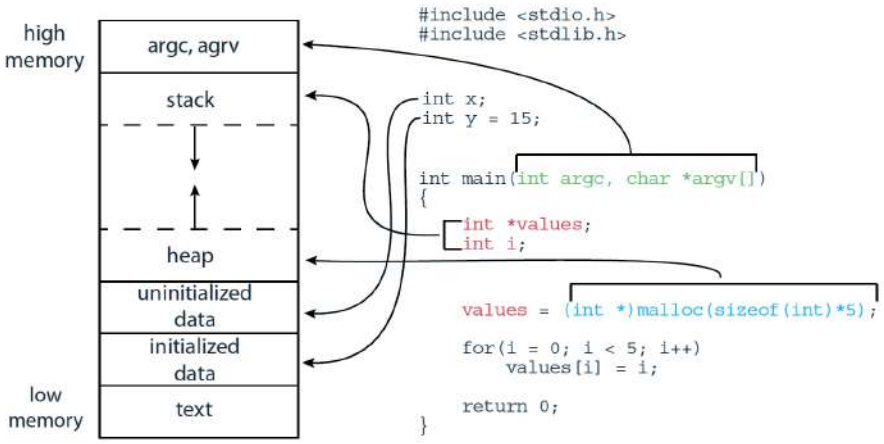

Figura 2.1: Alocação de memória de um programa em C
Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 108.

Como a seção `text` e a `heap` crescem e diminuem dinamicamente durante a execução do programa, existe um espaço, à ser utilizado, entre ambos, de forma que a seção `stack`, sai de um endereço superior da memória para um inferior, e o `heap` da inferior para a superior. Na Figura , o endereço superior está chamado de *high-memory*, e o inferior de *low-memory*. É papel do sistema operacional de assegurar que ambas as seções não se sobreponham.
Cada processo é representado, no sistema operacional, pelo PCB (*Process Control Block*), uma lista encadeada contendo informações como:

1. *Process state* - Podendo ser: `New`, `Running`, `Waiting`, `Ready`, `Terminated`.

2. *Program Counter* - Contador que indica o endereço da próxima instrução.
3. *CPU registers* - Valor dos registradores da CPU.
4. *CPU-scheduling information* - Informação sobre a prioridade do processo.
5. *Memory-management information* - Informações sobre o gerenciamento de memória.
6. *Accounting information* - Aqui está incluído dados como a quantidade de CPU usada.
7. *I/O status information* - Informação sobre arquivos abertos, dispositivos I/O utilizados etc.

2.1 Contexto de Execução

No código a seguir, temos a duas declarações da variável `a`. Isso é possível pois cada declaração está dentro de um contexto de execução diferente, de forma que cada função representa uma posição de memória distinto, e, assim, seus conteúdos ocupam um espaço, ou `Stack Frame`, específico da seção `stack`. Então, quando uma função é chamada, o `Program Counter` é desviado para essa a posição da memória, passando a executar as instruções lá inseridas, até ser redirecionado de volta ao endereço inicial (no qual a função foi chamada), descartando esse `Stack Frame`. O conteúdo contido no `Stack Frame` descartado, não é removido da memória, por causa do custo computacional de fazê-lo, fato gerador do chamado lixo de memória (`Stack Frame` sem contexto), diminuindo, assim, o `stack`. Quando um novo `Stack Frame` é concebido, esse sobrescreve o lixo de memória anteriormente criado, aumentando o `stack`.

```
1    void f()
2    {
3        int a = 4;
4        printf("%d\n", a);
```

```
5       }
6
7       int main(int argc, char *argv[])
8       {
9           int a = 1;
10          f();
11          printf("%d\n", a);
12
13      }
```

Cada tipo de variável ocupa uma quantidade de bits específico, como descriminado abaixo:

1. `char: 8 bits`
2. `int: 16 bits`
3. `long: 32 bits`
4. `float: 32 bits`
5. `double: 64 bits`

2.2 Ponteiros

No programa mostrado na Figura , foi inicializada uma variável do typo `int *`, o qual representa um ponteiro `int`. Os ponteiros são baseados na ideia da indireção, ou seja, na referencia indireta à outra posição da memória de tamanho `int` (16 bits). Por ser uma referência, ele não ocupa o espaço referido, e sim um específico para o seu tipo. No caso de uma arquitetura de 64 bits, o ponteiro ocupa 8 bytes (64 bits).

O trecho de código da Figura ,

```
1       int *values;
2       values = (int *) malloc(sizeof(int)*5);
```

```
3
4       for(i = 0; i < 5; i++){
5           values[i] = i;
6           // *(p + i) = i - Equivalente à: &values + i * sizeof(int)
7           // <= i
8       }
```

mostra que, o valor no qual está sendo referenciado pelo ponteiro values, está no `heap` (jogado lá pela função `malloc(3)`) ocupando um espaço de 5 `int` (ou seja, 5 de 16 bits ou 80 bits). Cada espaço desse é utilizado pela variável `i`, do tipo `int`, fazendo, assim, com que o espaço referenciado por `values` possa receber 5 do tipo `int`, como é utilizado dentro do loop `for`, com a artimética de ocupação `(&values + i*sizeof(int))`, adicionado no código em forma de cometário.

Podemos pensar na variável do ponteiro como um baú, no qual ao por `*`, estamos acessando o conteúdo do baú, `&` para o endereço de memória que contém o valor referenciado,

No código a seguir, podemos ver 3 operações distintas com o ponteiro `p`, as quais `&` revela o endereço de memória do valor "apontado", `*` acessa o conteúdo da memória e, por fim, `p` sem nenhum operador retorna o endereço da variável `p` na `stack`.

```
1       int *values;
2       int *p = malloc(sizeof(int)*4);
3
4       p[0] = 10;
5       printf("&p = %p/n", &p); // Endereço da referência (heap)
6       printf("&p = %d/n", *p); // Conteúdo da referência
7       printf("p = %p/n", p); // Endereço de p (Stack)
```

3 Processos e threads

3.0.1 Thread é uma parte do Processo

Como dito na aula anterior, um processo é uma entidade ativa a qual se refere à um programa em execução. Porém, a CPU não recebe essa entidade na sua integralidade para processamento, e sim uma parte da mesma chamada de *Thread* (a *Thread* pode ser definida como uma unidade de processamento do processo). Nessa está contido os elementos necessários para a mudança de contexto da CPU (*CPU Context Switch*), como os valores dos registradores utilizados, a memória Stack e o PC (*Program Counter*).

Em sistemas *single-threaded* (única *thread* por processo, normalmente encontrado em sistemas mais antigos), cada nova tarefa a ser executada deve-se passar pela criação de um novo processo.

3.0.2 Concurrency vs Parallelism

Uma observação importante é que um programa, em um sistema com um único núcleo de processamento, pode rodar em simultâneo (*concurrency*) com outros programas, devido a alternação do processo a ser executado pela CPU em certos

períodos de tempo, porém não em paralelo (*parallelism*), por não ter múltiplos núcleos.

3.0.3 Concurrency

Os processos são distribuídos para a CPU com a técnica *Round-Robin*, com a qual é gerada uma fila de processos, e a saída de cada fila vai para o processador. Após um certo período de tempo, esse processamento é interrompido, o seu contexto é salvo, e o processo volta para o final da fila (observar que há políticas de prioridades. No exemplo está sendo considerado que todos os processos têm a mesma prioridade). Um novo processo então é chamado para execução, com o seu contexto de CPU sendo carregado. Essa mudança de contexto da CPU (*CPU context-switch*) é o que torna possível a CPU retomar o processamento de um processo específico do momento que o mesmo foi anteriormente interrompido. Esse ciclo continua até a conclusão de todos os processos.

3.0.4 Criação de novos processos

Na linguagem C, a criação de um novo processo passa pela função `clone(2)`, a qual copia, conforme instruções do programador, todos os elementos do processo original para o novo processo, consumindo, assim, recursos como a memória, com a locação múltipla dos mesmos dados. Uma tentativa de otimização pode ser observada na função `fork(2)` (que é uma implementação específica da função `clone(2)`), que utiliza o conceito de `CoW` (*Copy on Write*, ou cópia na escrita, que é uma funcionalidade do sistema de gerenciamento de memória), somente copiando os dados, para o novo endereço de memória, caso haja alteração dos mesmos. Isso é possível com o uso dos ponteiros, discutidos na aula anterior, para a leitura dos dados. É importante notar que a execução do processo criado continuará a partir da última posição do IP (*Instruction Pointer*, ou ponteiro de instrução), fazendo com que as instruções escritas antes do `clone(2)` ou `fork(2)` nunca sejam lidas.

A função `fork()` retorna:

1. -1, caso tenha falhado;

2. 0, para o processo filho, com o PPID (*Parent Process Identifier*, ou identificador do processo pai) equivalendo ao PID (*Process Identifier*, ou identificador do processo) do processo pai

3. PID do processo filho, no processo pai.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        printf("Before fork: %d \n", getpid());
        pid_t returned = fork(void);

        if(returned == 0){
                printf("Child here: %d \n", getpid());
        } else {
                printf("Parent here: %d, child pid: %d \n", getpid(), returned);
        }

        return 0;
}
```

3.0.5 Por que devemos criar novos processos?

Há dois motivos para a criação de novos processos (em sistemas mais antigos):

1. Inicialização de um novo programa, com o `execve(2)`.

2. Múltiplas tarefas de um mesmo programa.

Apesar da otimização citada na seção anterior (com o `CoW`), o procedimento de criação de um novo processo é custoso de recursos computacionais, sendo o *multi-threading* (múltiplas threads no mesmo processo) uma solução para a criação de múltiplas tarefas de um mesmo programa (caso 2).

3.0.6 execve(2)

A função `execve(2)` tem como objetivo alterar a imagem de execução, que está contida no processo, para a do novo programa, isso é, o *Data*, *Heap*, *Stack* e *Text* serão carregados com o conteúdo do novo programa.
A execução do programa `hello.c`, dentro do `execv.c`, ambos escritos abaixo, retornará o mesmo `pid` (*Process Identification*) do que no primeiro programa, pois o processo é o mesmo, mas a imagem é diferente.
Arquivo `execv.c`:

```
1   #include <stdio.h>
2   #include <sys/types.h>
3   #include <unistd.h>
4
5   int main()
6   {
7           char *argv[] = {"", NULL}; //array of char*
8           char *envp[] = {"", NULL}; //array of char*
9
10          printf("before execve. pid(%d)\n", getpid());
11
12          //Troca a imagem de execução.
13          int ret = execve("hello", argv, envp);
14          //Caso positivo, as linhas a seguir não existirão mais;
15
16      printf("supposed never executed\n");
17          if(ret == -1) {
```

```
18                      perror("execve failed");
19                      return 1;
20              }
21
22              return 0;
23      }
```

Arquivo `hello.c`:

```
1       #include <stdio.h>
2       #include <sys/types.h>
3       #include <unistd.h>
4
5       int main()
6       {
7               printf("hello world from pid(%d)\n", getpid());
8               return 0;
9       }
```

*Obs: o programa irá buscar um arquivo em código de máquina, e não em C. Assim:

```
$ make hello.c
$ make execv && ./execv
```

3.0.7 Multi-threading

É importante notar que os sistemas operacionais evoluíram, com os mais antigos limitando-se à *single-threading* (uma *thread* por processo), e os mais novos rompendo essa limitação, passando à *multi-threading* (múltiplas *threads* por processo), como mostrado na Figura 3.1.
O *multi-threading* trás consigo diversas vantagens, como:

1. Responsividade: Um programa continua rodando mesmo quando uma parte sua é bloqueada, trava ou demora para concluir.

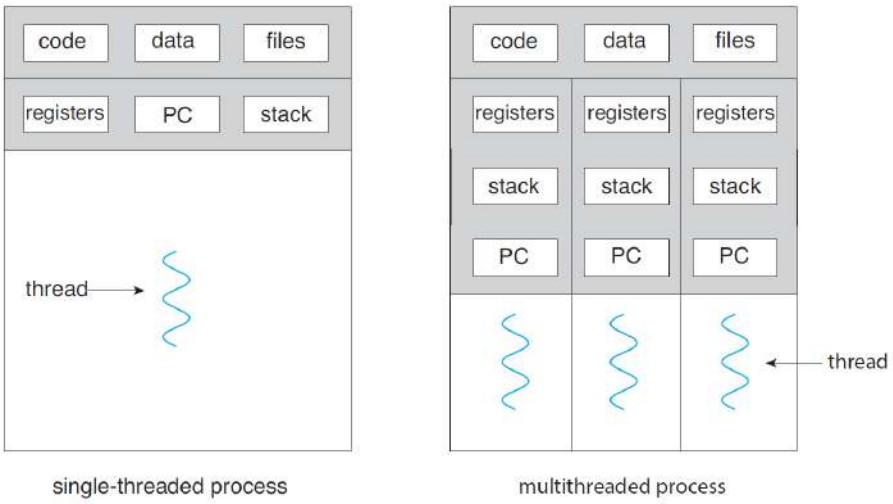

Figura 3.1: Single and Multi-Threaded Example
Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 160.

2. Compartilhamento de Recursos: As *Threads* compartilham o mesmo código e dados por padrão.

3. Economia: é mais econômico criar uma *Thread* do que um processo. É mais rápido trocar o contexto da CPU (*CPU context-switch*) entre *Threads* do que entre processos, desde que estejam em *userspace*.

4. Escalabilidade: *Threads* podem usufruir do paralelismo, rodando em múltiplas CPU's em simultâneo de forma assíncrona.

É com o *multi-threading* que um programa consegue tirar vantagem de vários núcleos de processamento, de forma a utilizar tanto o *Concurrency* como o *Parallelism* (multiplas *threads* com um único núcleo ocorre o *Concurrency* mas não o *Parallelism*).

3.1 Função como argumento de outra função.

Utilizando Ponteiros, podemos passar uma função como argumento de outra função, basta coletar o endereço de memória da mesma (o nome da função coincide com o endereço dela) e passar para um ponteiro. Depois passar o ponteiro no argumento da função. A seguir temos um código de exemplo de como utilizar ponteiros para funções (bastando passar a variável `fp` como argumento de uma função).

```
1   int funca(int); //declarando uma função sem definir
2   int funcb(int);
3
4   int main()
5   {
6           int (*fp)(int);
7           //Declarando um ponteiro de função com um argumento int
8
9           fp = funca;// fp = funcb;
10          printf("fp(%d) = %d\n", 3, fp(3));
11  }
12
13  int funca(int n)
14  {
15          return 2 * n
16  }
17
18  int funcb(int n)
19  {
20          return 100 * n
21  }
```

3.2 Complementar

Pesquisar Sobre

Para saber mais, procurar sobre as funções do bash $:

1. ps aux
2. ps aux | wc -l
3. ps -ef
4. top (u)
5. htop
6. lscpu

Função C: 1. atoi() -> alphabet to integer 7.

4 Retomando aos processos

Nas últimas, foram debatidos diversos aspectos dos processos dos SO (Sistemas Operacionais). Esses aspectos orbitam os 3 pontos enfatizado a seguir:

1. Identificar os componentes de um processo.
2. Criação e Término de processos
3. Comunicação entre processos: *Shared Memory* x *Message Passing*.

4.0.1 Componentes de um processo.

Enquanto que um programa é uma entidade passiva localizada na memória secundária, um processo é instância ativa de um programa, e está localizado na memória principal em quatro sessões básicas (Como mostrado na Figura **??**): *Text*, código executável; *Data*, variáveis globais; *Heap*, memória dinamicamente alocada; *Stack*, armazenamento de dados temporários.
Do ponto de vista do Sistema Operacional, cada processo é representado pelo *Process Control Block* (PCB), mostrado na Figura 4.2, o qual contém vários pedaços de informação necessários para iniciar ou reiniciar um processo específico, como *Process State, Program counter, CPU registers* e *CPU-scheduling information*.

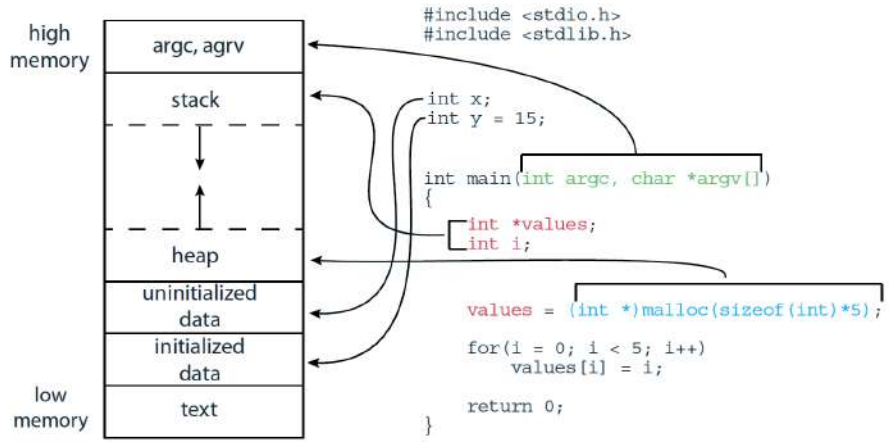

Figura 4.1: Processo na memória principal

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 106.

De forma geral, os processos podem assumir 5 estados (Como mostrado na Figura 4.3): criado, pronto, rodando, esperando e terminado.

No Linux, todos os PCB's encontram-se em uma lista duplamente encadeada (com o próximo e o anterior), com o sistema operacional armazenando um ponteiro para a estrutura que está atualmente em execução pela CPU. Mostrado na Figura 4.4.

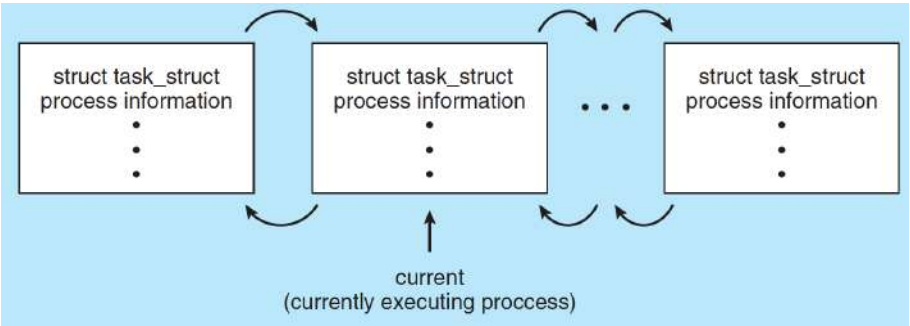

Figura 4.4: Representação da fila encadeada

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 111.

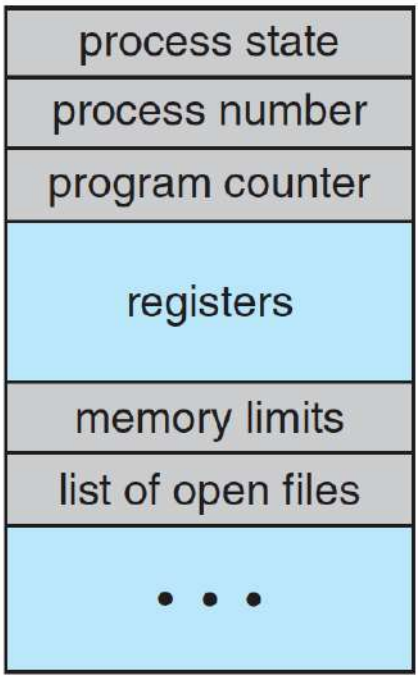

Figura 4.2: PCB: Representação do processo no SO
Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 109

Conforme o estado do processo muda, o PCB também muda de fila, alternando entre a fila encadeada de “pronto” (pronto para ser executado) e espera (espera de ocorrência de evento), mostradas na Figura 4.5.

O responsável por depositar e recolher os PCB’s dessas filas é o *process Scheduling*, o qual foi representado no diagrama da Figura 4.6.

O *CPU Scheduling*, por sua vez, tem como papel recolher um processo da fila *ready* para um núcleo de processamento. Para ser alocado na CPU, é necessário salvar o estado do processo anteriormente alocado, bem como os elementos necessários para continuar processando-o posteriormente, e carregar o novo contexto de execução. No decorrer dessa tarefa, chamada de *Context Switch*, a CPU fica ociosa, como mostra a Figura 4.7.

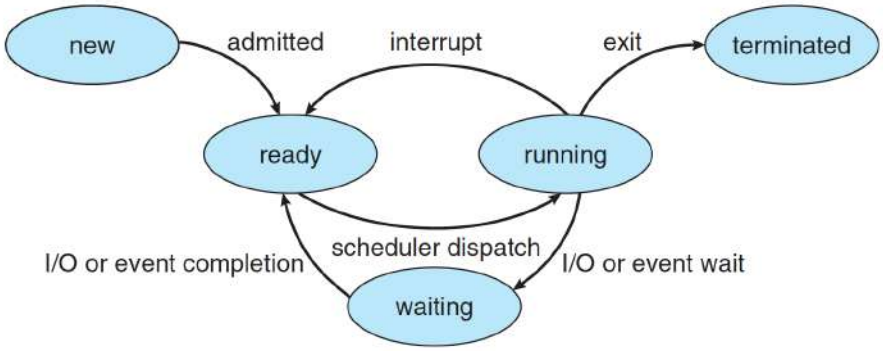

Figure 3.2 Diagram of process state.

Figura 4.3: Estados de um processo

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 109.

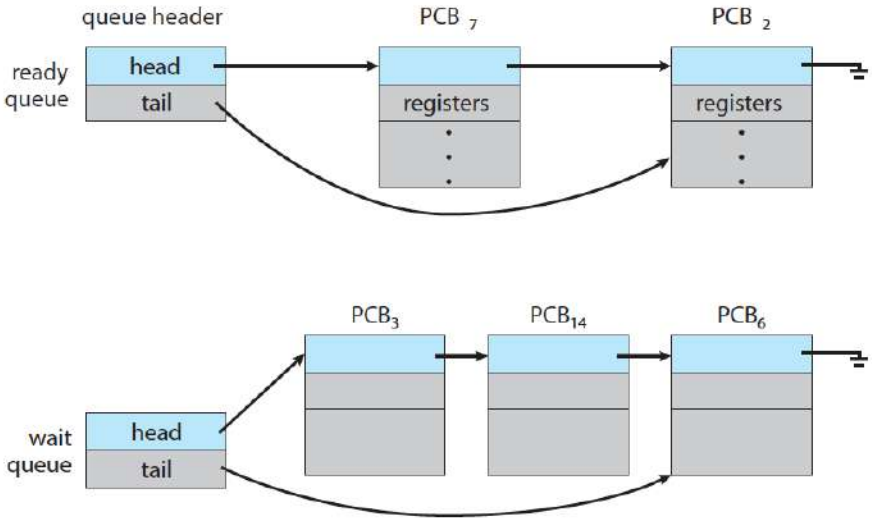

Figura 4.5: Filas de espera e pronto

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 112.

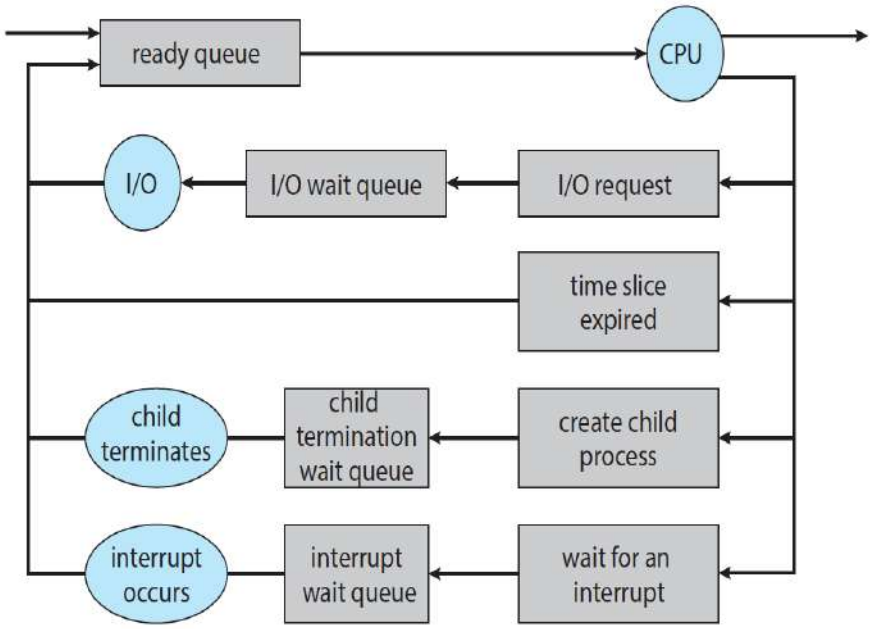

Figura 4.6: Diagrama de agendamento de processos

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 113.

4.0.2 Criação e Término de um processo.

Cada processo tem uma identificação única chamada de `pid` (*process identifier*). A partir dele pode ser criado um processo filho. Conscutivas criações formam uma árvore de processos, mostrada na Figura 4.8.

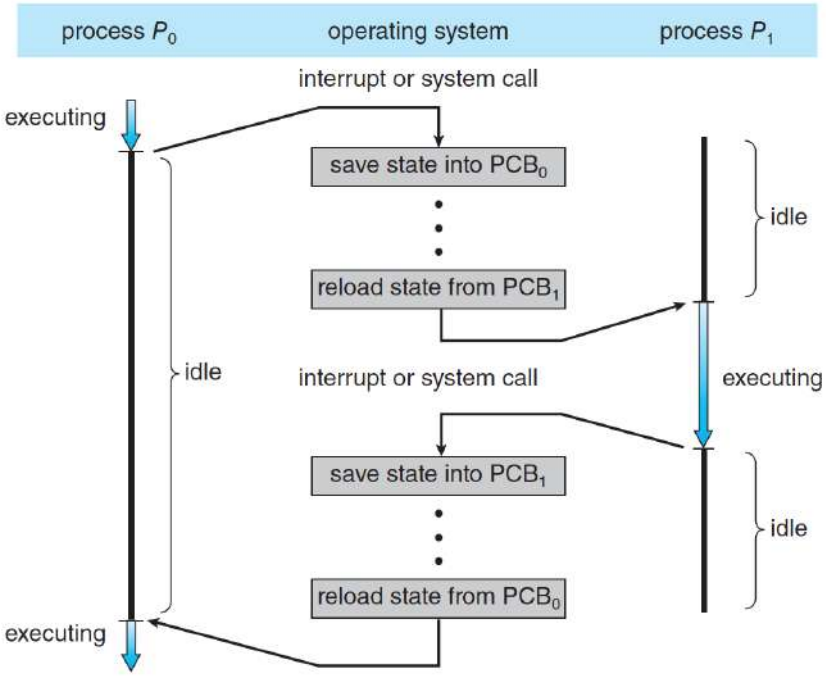

Figura 4.7: Diagrama da troca de contexto

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 113.

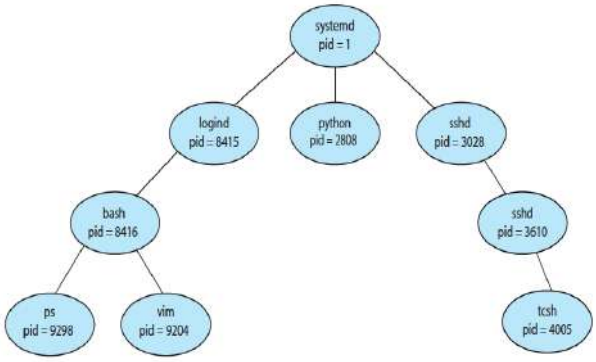

Figura 4.8: Árvore de processos típico de sistema Linux

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 116.

O ciclo de criação e término de um processo, mostrado na Figura 4.9, passa pelas funções de chamada de sistema na linguagem `C`: `fork()`, para criar um novo processo; `wait()`, esperar o processo filho acabar; `execve()`, troca a imagem de execução (troca o programa); `exit()`, encerrar o processo.

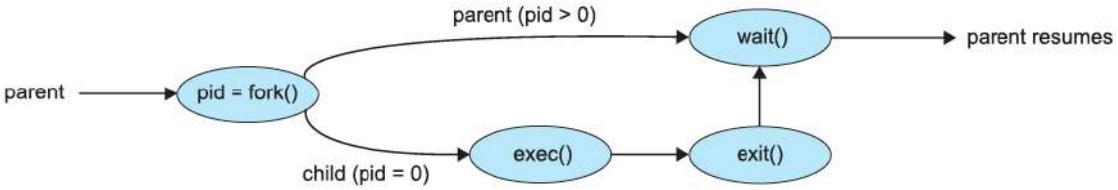

Figura 4.9: Criação e término de um processo
Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 119.

Alguns sistemas não permitem que um processo "viva" sem que o "pai" exista. Então, se o "pai" for encerrado, todos os "filhos" serão encerrados. Da mesma forma que os filhos dos filhos. Fenômeno chamado de *cascading termination*

Ao chamar a função `exit()`, um processo filho tem seus recursos deslocados pelo sistema operacional. Porém ele continua na tabela de processos até o processo pai executar a função `wait()`, pois nessa tabela está contido o estado de saída do mesmo. Durante essa transição, de seus recursos terem sido deslocados mas ainda estar na tabela de processos, o processo é chamado de `Zumbie`, o que, normalmente, ocorre para todos os processos (ao serem terminados) por um curto período de tempo. Se o pai não chamar a função `wait()` e terminar sua execução, os processos filhos passarão a ser chamados de processos `órfãos`. O Sistema Operacional Linux utiliza de outros processos para herdar os `órfãos` e libera-los da tabela de processos chamando a função `wait()`.

4.1 Comunicação entre processos: *Shared Memory* x *Message Passing.*

Um processo pode ter sido criado com o objetivo de executar um algoritmo que independe de outros processos (esse processo é chamado de independente). Ou, pode ter sido criado para cooperar com outros processos (esse processo é chamado de cooperativo). Para que a cooperação ocorra, é necessário criar um canal de comunicação (um *Interprocess Communication - IPC*), de tal maneira que os dados possam ser transmitidos de uma para outra. Há dois métodos para tal: o *Shared Memory* (compartilhamento de memória); e o *Message Passing* (sistema de passagem de mensagem).

Há várias razões para ter um processo cooperativo. 1. Troca de informações: Compartilhamento de informações entre os interessados. 2. Aceleração computacional: Dividir uma tarefa em várias pequenas partes e, assim, processá-los ao mesmo tempo. 3. Modularidade: Com as funções do sistema dividido em diferentes processos ou threads.

4.1.1 Shared Memory

Shared Memory é o método que torna uma faixa de endereços de memória acessível à diferentes processos. Por questões de segurança, o sistema operacional (de forma padrão) não permite que os processos tenham acesso aos espaços de memória dos outros. Para que esse método ocorra, é fundamental que todos os processos participantes desse método concordem em compartilhar a sua faixa de memória. Após executado, o sistema operacional não mais intervem, deixando para os processos a responsabilidade de garantir que não ocorra o problema chamado *Race Conditions* (ou condição de disputa).

O *Race Conditions* decorre de um problema de *concurrency* (simultaneidade) entre os diferentes processos, como a escrita em um mesmo endereço de memória transcorrendo por mais de um processo ao mesmo tempo.

Uma solução para garantir a comunicação entre os processos é utilizando a arquitetura *producer-consumer*, no qual o processo *producer* produz e armazena o conteúdo em um buffer, que será consumido pelo *consumer* posteriormente. Esse *buffer* pode ser *unbounded buffer* (não tem limites de tamanho) ou *bounded buffer* (com o tamanho limitado).

4.1.2 Message Passing

A idéia por de trás do método *message passing* é a de passar a responsabilidade da distribuição e validade da comunicação inter-processual para o sistema operacional, promovendo uma abstração de uso facilitado (em comparação ao *shared memory*), pois descarta a necessidade da criação explícita de um código de gerenciamento de acesso e manupulação dos dados compartilhados. Esse mecanismo de comunicação cria um elo de ligação (*link*) entre dois processos, que demanda ao menos duas operações: *send(message)*; e *receive(message)*.
A implementação desse método passa por 3 pontos:

1. Comunicação direta e indireta.
2. Síncrona ou assíncrona.
3. *Buffer* automático ou explícito.

Comunicação Direta

Na comunicação direta, o processo deve nomear explicitamente o nome do receptor ou emissor: 1. *send(P, message)*: Enviar menssagem para o processo P 2. *receive(Q, message)*: Receber mensagem do processo Q

Comunicação indireta

Na comunicação indireta, as mensagens passam através de uma abstração de caixa de correios (*mailbox* ou *ports*) compartilhada, na qual as mensagens podem ser postadas e removidas. Cada *mailbox* tem uma identificação única (no POSIX, cada

mailbox é identificado por um inteiro). Nessa implementação, o processo deve nomear de qual e para qual caixa de correios deve enviar ou receber a mensagem:

1. *send(A, message)*: Enviar uma mensagem para o *mailbox* A.
2. *receive(A, message)*: Receber uma mensagem do *mailbox* A.

Um possível *Race Condition* ocorre quando o processo P1 envia uma mensagem, e o P2 e P3 estão chamando a função `receive()`. Para evitar que isso aconteça, existem 3 possibilidades:

1. Permitir que somente 2 processos estejam associados à uma caixa de correios.
2. Permitir que somente 1 processo por vez execute a função `receive()`
3. Permitir que o sistema operacional escolha qual processo deve receber (P1 ou P2, mas não ambos).

Sincronia

Para cada um dos dois primitivos (`send()` e `receive()`), o sistema de transmissão de mensagens pode assumir 2 estados, o síncrono e o assíncrono:

1. Síncrono (*Blocking* ou bloqueante): 1.1. O emissor (P1) é travado até o receptor (P2) receber a mensagem. 1.2. O receptor (P1) é travado até uma mensagem ficar disponível.
2. Assíncrono (*Nonblocking* ou não-bloqueante) 2.1. O emissor (P1) continua a executar suas instruções após a emissão da mensagem. 2.2. O receptor (P2) pode receber uma mensagem inválida (NULL), assim continuar sua execução mesmo sem ter recebido uma mensagem válida.

4.1.3 Buffer

Todas as mensagens enviadas devem passar por um sistema de fila que, basicamente, tem 3 formas distintas (que apenas variam sua capacidade):

1. Capacidade 0 (*Zero capacity* ou sem *buffer*): O emissor deve ser síncrono com o receptor, caso contrário a mensagem será perdida.

2. Capacidade limitada (*Bounded capacity*): O emissor pode ser assíncrono caso haja capacidade de armazenamento de mensagens no *buffer*. E, caso a fila esteja cheia, deve ser bloqueado até haver espaço disponível.

3. Capacidade ilimitada (*Unbounded capacity*): O emissor nunca é bloqueado, pois a fila não tem limites de recepção de mensagens.

4.2 Complementar

Essa sessão faz alusão à outros conteúdos também importantes relacionados com a aula, mas que dará ao leitor o papel de buscar mais informações sobre os mesmos conforme sua necessidade e interesse.

4.2.1 Threads - criação em C

Nas aulas anteriores foram discutidos as vantagens das threads em comparação aos processos. Abaixo está um código comentado em C para criação de threads.

```c
#include <stdio.h>
#include <pthread.h> //POSIX thread library, or pthread.


int sum;
void *runner(void *param);

int main(int argc, char *argv[])
{
        pthread_t tid;
        pthread_attr_t attr;

```

```
13              pthread_attr_init(&attr);
14              //Atributo necessário para a criação da thread
15
16              /*
17               * pthread_create
18               * tid vai retornar o 'id' da thread.
19               * runner é a função que será executada
20               * argv[1] é o argumento no qual será passado para a função runn
21               *
22               */
23              pthread_create(&tid, &attr, runner, argv[1]);
24              pthread_join(tid, NULL); //Espera uma thread Terminar
25
26              return 0;
27      }
28
29      void *runner(void *param){
30
31              int upper = atoi(param);
32              int sum = 0;
33
34              for(int i = 0; i < upper; i++){
35                      sum += 1;
36              }
37
38              pthread_exit(0);
39      }
```

4.2.2 Pesquisar Sobre

1. Teoria de filas

2. *pipes* (mecanismo *IPC - InterProcess Communication*) - Página 139 (Capítulo 3.7) de Silberschatz, A. Operating System Concepts, 10th.

5 Retomando às Threads.

Imagine, caro leitor, que você seja dono de uma loja e que você não tenha funcionários, como na Figura 5.1. Todas as tarefas, entre administrar o estoque, dar baixa no caixa e atender os clientes, deve ser feito por você. Qual estratégia você deve tomar para tornar isso possível ?

A proposta natural é a de seguir uma sequência de passos: inicialmente deve-se chamar o primeiro cliente da fila; atendê-lo; depois administrar o estoque; vender o produto; dar baixa no caixa; repetir o ciclo.

A vantagem é que todo o processo de venda é concluído antes de iniciar o próximo ciclo. A desvantagem é que os novos clientes terão que esperar bastante tempo para serem atendidos, e a fila de espera pode desanimar possíveis compradores, como na Figura 5.2. É preciso perceber também que o gargalo desse processo está no atendimento ao cliente, afinal, a escolha do cliente (ou *input* do usuário do processo) não tem um momento determinado para ocorrer. Outro problema é que um travamento (por conta de um problema imprevisto) em uma dessas tarefas impedirá (ou atrasará) que as próximas sejam trabalhadas.

Para solucionar esse gargalo, poderíamos pensar em alocar, para cada tarefa, uma parcela de tempo, digamos 10 minutos, de forma que, após esgotado esse tempo, você terá que alterar a tarefa a ser executada, independente se a mesma foi

Figura 5.1: Loja de único atendente
Autor: rawkkim, de: unsplash.com

concluída ou não, passando para a próxima não interdependente (*non-blocking*). Assim, o leitor iniciaria o processo chamando o primeiro da fila. Após 10 minutos, caso o atendimento não seja concluído (por exemplo, o cliente ainda não tenha escolhido o produto que quer), essa tarefa atualmente em trabalho será deixada de lado (entrando no estado de espera), e a próxima tarefa não interdependente (que no caso seria chamar o próximo da fila) deverá ser executada. A vantagem dessa abordagem é que o tempo dentro da fila de espera deve cair substancialmente, pois aproveita-se o tempo (no qual, anteriormente, você ficava ocioso) de escolha (de um produto) de um cliente, atendendo o próximo da fila. Essa abordagem também trás desvantagens. Primeiro, você terá que administrar (e muito bem) a prioridade e a alternância entre tarefas, aumentando assim a complexidade do ciclo. Os clientes que fazem escolhas rápidas podem não ficar contentes em ter que esperar que outras tarefas relacionadas com outros compradores sejam concluídas (ou que entrem no estado de espera) previamente, ocorrência que

Figura 5.2: Fila de espera
Autor: Levi Jones, de: unsplash.com

diminui o tempo de resposta (responsividade) do processo. Igualmente, um número suficiente de compradores pode tornar a responsividade tão baixa que os clientes, simplesmente, desistirão do atendimento, derrubando a qualidade percebida e, a loja, deixando de realizar a venda. A imposição de padrões de qualidade (como um tempo máximo de espera por cliente), pode resolver o problema anterior, mas criar outros, como a loja negar a servir mais que uma quantidade específica de pessoas por período de tempo (analogamente, ataque DDoS). Pode-se pensar também em abrir uma segunda (ou terceira, quarta, etc) loja ao lado, com um colaborador cada, de forma que mais clientes possam ser atendidos. Porém, novos recursos (como caixa, material lampadas, etc) terão que ser adquiridos, demandando tempo e dinheiro para tal. Então, a solução mais simples (e com diversas vantagens) é a contratação de empregados para que múltiplas tarefas possam ser desempenhadas em paralelo, como mostrado na Figura 5.3.

Figura 5.3: Três trabalhos em paralelo
Autor: Charlie Firth de: Unsplash

A história acima é uma analogia ao que ocorre nos sistemas computacionais modernos, de forma que:

• O processo (em um sistema operacional) é equivalente à loja. • Funcionários são como *threads*. • Os recursos da loja (caixa, material lampadas, etc) são como os recursos computacionais (como a memória). • O ciclo de venda é o serviço oferecido pelo sistema computacional. • E assim em diante.

Essa comparação é interessante pois a mesma pergunta vem à tona: devemos criar um novo processo (abrir uma nova loja) ou, simplesmente, criar novas *threads* (contratar colaboradores) ?

O capítulo atual foi dividido em:

• Identificar os componentes de uma *thread*. • Benefícios e desafios. • *Threads* e o Linux. (descrever como as *threads* são representadas no Linux e projetar uma aplicação). • Complementar

5.1 Identificar os componentes de uma *thread*.

A *Thread* (unidade de processamento do processo) é composto pela *thread id* (seu identificador único), PC (*program counter*), valores dos registradores, memória *stack*. Ela compartilha, com outras *theads* do mesmo processo, recursos como as sessões de memória *text*, *data* e os arquivos abertos, como mostrado na Figura 5.10.

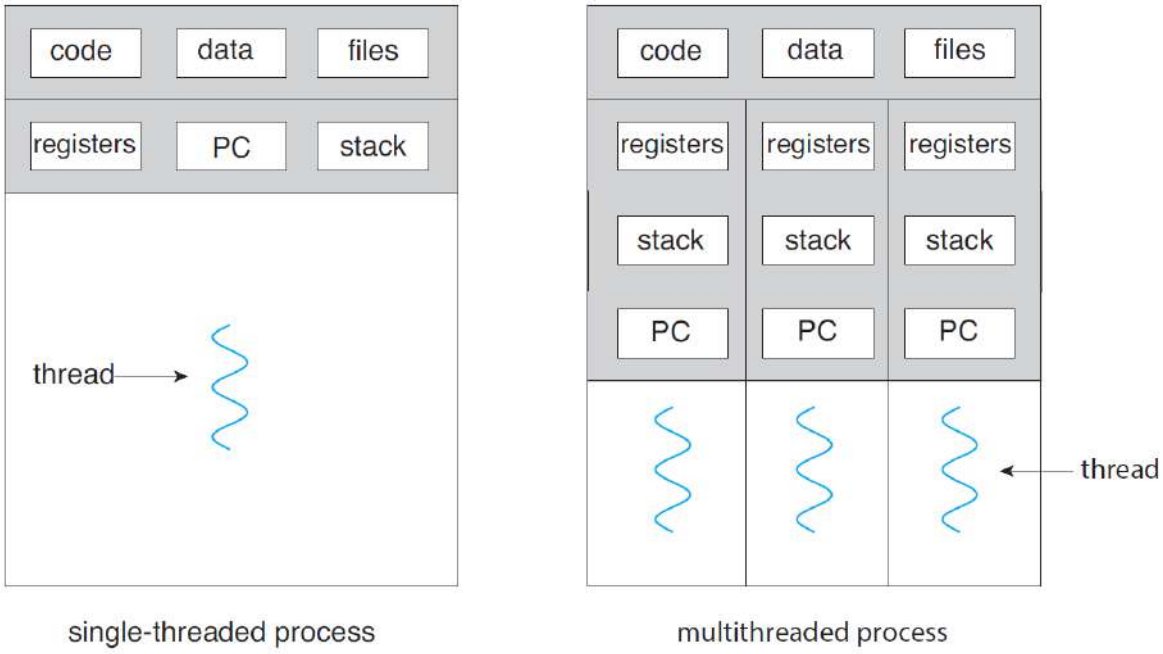

Figura 5.4: Processos com uma ou múltiplas threads

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 160.

5.2 Benefícios e Desafios.

Benefícios.

Em sistemas com múltiplos núcleos de processamento, as *threads* podem trazer várias vantagens, como:

1.Responsividade: Um programa continua rodando mesmo quando uma parte sua é bloqueada, trava ou demora para concluir. 2.Compartilhamento de Recursos: As Threads compartilham o mesmo código e dados por padrão. 3.Economia: é mais econômico criar uma Thread do que um processo. É mais rápido trocar o contexto da CPU (CPU context-switch) entre Threads do que entre processos, desde que estejam em userspace. 4.Escalabilidade: Threads podem usufruir do paralelismo, rodando em múltiplas CPU's em simultâneo de forma assíncrona.

Desafios.

1. Identificar tarefas: examinar a aplicação e separá-la em diferentes tarefas que possam ser executadas em paralelo.

2. Balancear: a carga de trabalho deve ser distribuída entre as tarefas (idealmente, as tarefas teriam a mesma carga).

3. Divisão de dados: os dados devem sem divididos para rodar em núcleos de processamento separados (para evirar a *race condition*).

4. Dependência de dados: os programadores devem garantir que as tarefas estejam sincronizadas quando uma *thread* depende dos dados de outra.

5. Teste: é mais difícil testar e replicar casos de erro em sistemas com múltiplas *threads* em comparação com as de *thread* única.

5.3 Modelos de multithread

Há dois tipos diferentes de *threads*, aquelas que estão no nível de usuário, sendo gerenciadas sem ajuda do *kernel*, e aquelas que estão no nível do *kernel*, que são gerenciadas diretamente pelo sistema operacional. Assim, os modelos de *multithread* definem o relacionamento desses dois tipos de *threads*.

1. *Many-to-one* (Figura 5.5): nesse modelo, várias *threads* de nível do usuário são mapeadas para uma única *thread* de *kernel*.

2. *One-to-one* (Figura 5.6): cada *thread* de usuário tem, mapeada, uma *thread* de *kernel*.

3. *Many-to-many* (Figura 5.7): várias *threads* de usuário são multiplexadas entre as diferentes *threads* de kernel.

4. *Two-level* (Figura 5.8): esse modelo une o *one-to-one* com o *many-to-many*.

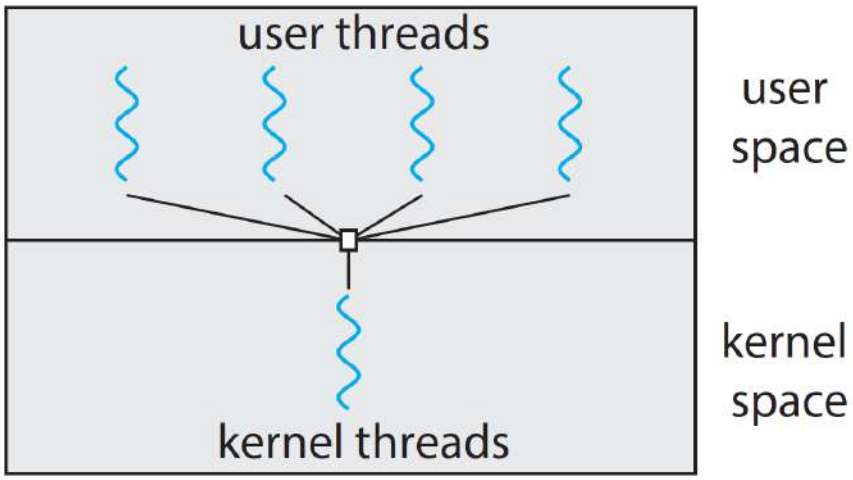

Figura 5.5: Modelo Many-to-One
Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 166.

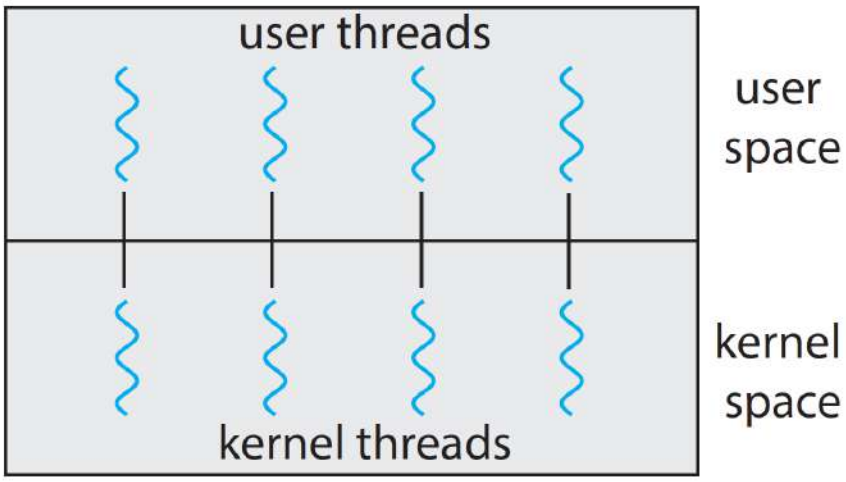

Figura 5.6: Modelo One-to-One

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 167.

5.4 Threads e o Linux

5.4.1 Biblioteca `pthread`

O Linux utiliza a biblioteca POSIX *threads*, nomeada, em `C`, de `pthread`.

O exemplo a seguir foi retirado de: Silberschatz, A. Operating System Concepts, 10th, página 170.*

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int sum; /* this data is shared by the thread(s) */
6  void *runner(void *param); /* threads call this function */
7
8  int main(int argc, char *argv[])
```

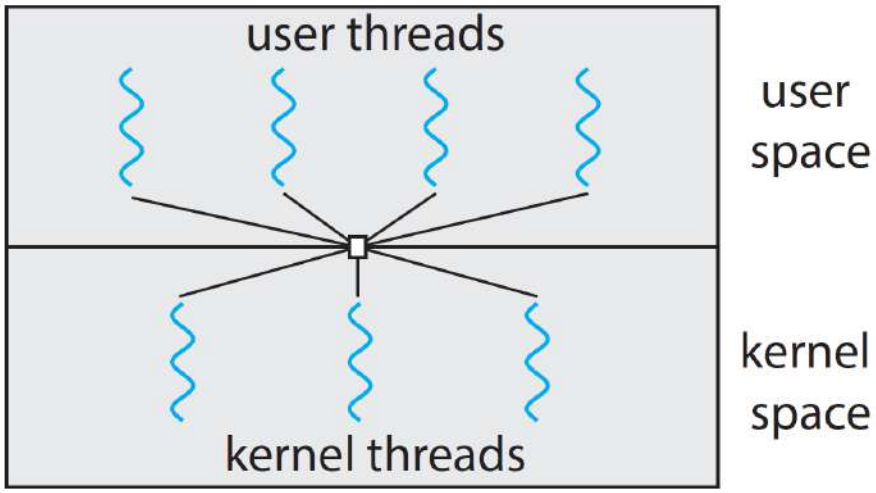

Figura 5.7: Modelo Many-to-Many

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 167.

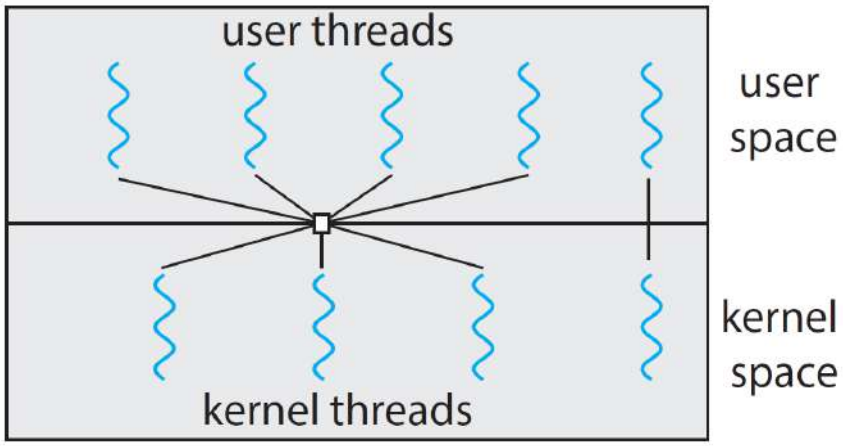

Figura 5.8: Modelo two-level

Imagem retirada de: Silberschatz, A. Operating System Concepts, 10th, página 168.

```
9       {
10          pthread_t tid; /* the thread identifier */
11          pthread_attr_t attr; /* set of thread attributes */
12          /* set the default attributes of the thread */
13          pthread_attr_init(&attr);
14          /* create the thread */
15          pthread_create(&tid, &attr, runner, argv[1]);
16          /* wait for the thread to exit */
17          pthread_join(tid,NULL);
18          printf("sum = %dn",sum);
19      }
20
21      /* The thread will execute in this function */
22      void *runner(void *param)
23      {
24          int i, upper = atoi(param);
25          sum = 0;
26          for (i = 1; i <= upper; i++)
27          sum += i;
28          pthread exit(0);
29      }
```

5.4.2 Semáforos

A execução em paralelo de várias *threads* (de dados compartilhados) pode gerar uma *race condition* (conceito debatido em capítulos anteriores) por poder alterar, com dados desatualizados, o mesmo valor. Considere:

```
1       int a = 5;
2       a = a - 1;
```

A *thread* "A" e "B", executando em paralelo, podem gerar o resultado final de 4, e não de 3, como esperado, Pois:

1. *Thread* "A" coleta o valor de `a` (5).
2. *Thread* "B" coleta o valor de `a` (5).
3. *Thread* "B" calcula `a - 1` (4).
4. *Thread* "A" calcula `a - 1` (4).
5. *Thread* "A" ajusta o valor de `a` (4).
6. *Thread* "B" ajusta o valor de `a` (4).

Esse caso fica visível no exemplo a seguir:

Exemplo retirada de: Griffiths, David; Griffiths, Dawn; Head First C, página 510.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <errno.h>
6  #include <pthread.h>
7
8  int beers = 2000000;
9  void* drink_lots(void *a)
10 {
11 int i;
12 for (i = 0; i < 100000; i++) {
13 beers = beers - 1;
14 }
15 return NULL;
16 }
17 int main()
18 {
19 pthread_t threads[20];
20 int t;
```

```
21      printf("%i bottles of beer on the wall\n%i bottles of beer\n", beers, be
22      for (t = 0; t < 20; t++) {
23      ( , NULL, , NULL);
24      }
25      void* result;
26      for (t = 0; t < 20; t++) {
27      pthread_create(threads[t], &result);
28      }
29      printf("There are now %i bottles of beer on the wall\n", beers);
30      return 0;
31      }
32      pthread_join
33      &threads[t]
34      drink_lots
```

no qual, o trecho abaixo (região crítica) necessita ser protegido pelo acesso múltiplo, como explicado anteriormente.

```
1       for (i = 0; i < 100000; i++) {
2           beers = beers - 1;
3       }
```

Sem essa proteção, os resultados serão:

Resultados retirados de: Griffiths, David; Griffiths, Dawn; Head First C, página 511.

```
> ./beer
2000000 bottles of beer on the wall
2000000 bottles of beer
There are now 0 bottles of beer on the wall
> ./beer
2000000 bottles of beer on the wall
2000000 bottles of beer
There are now 883988 bottles of beer on the wall
```

```
> ./beer 2000000 bottles of beer on the wall
2000000 bottles of beer
There are now 945170 bottles of beer on the wall ```
```

Perceba que, sem proteção, os resultados não são consistentes.

Para que essas duas *threads* trabalhem de forma síncrona, é necessário que haja um semáforo (mesmo conceito de controle de tráfego de carros), para o controle de acesso, como mostrado na Figura 5.9. Os semáforos que previnem que as *threads* não se choquem são chamadas de `mutex` (*mutually exclusive*, só permite uma única *thread* na região crítica) e também de `locks`.

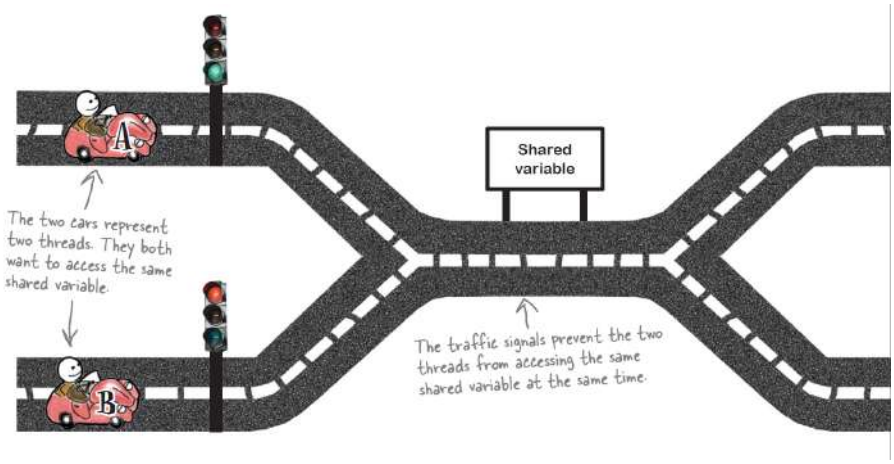

Figura 5.9: Semáforo para controle de acesso.

Imagem retirada de: Griffiths, David; Griffiths, Dawn; Head First C, página 513.

Existem, então, duas soluções para o trecho de código anterior, cada qual gerará resultados de saída, desempenho e sincronicidade diferentes.

Solução A.

```
1    pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
2
3    void* drink_lots(void *a)
4    {
```

```
5           int i;
6           pthread_mutex_lock(&beers_lock);
7
8           for (i = 0; i < 100000; i++) {
9               beers = beers - 1;
10          }
11
12          pthread_mutex_unlock(&beers_lock);
13          printf("beers = %i\n", beers);
14          return NULL;
15      }
```

Solução B.

```
1       pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
2
3       void* drink_lots(void *a)
4       {
5           int i;
6
7           for (i = 0; i < 100000; i++) {
8               pthread_mutex_lock(&beers_lock);
9               beers = beers - 1;
10              pthread_mutex_unlock(&beers_lock);
11          }
12
13          printf("beers = %i\n", beers);
14          return NULL;
15      }
```

Resultados retirados de: Griffiths, David; Griffiths, Dawn; Head First C, página 517.

Resultado Versão A.

```
> ./beer
2000000 bottles of beer on the wall
2000000 bottles of beer
beers = 1900000
beers = 1800000
beers = 1700000
beers = 1600000
beers = 1500000
beers = 1400000
beers = 1300000
beers = 1200000
beers = 1100000
beers = 1000000
beers = 900000
beers = 800000
beers = 700000
beers = 600000
beers = 500000
beers = 400000
beers = 300000
beers = 200000
beers = 100000
beers = 0
There are now 0 bottles of beer on the wall
>
```

Resultado Versão B.

```
> ./beer_fixed_strategy_2
2000000 bottles of beer on the wall
2000000 bottles of beer
beers = 63082
```

```
    beers = 123
    beers = 104
    beers = 102
    beers = 96
    beers = 75
    beers = 67
    beers = 66
    beers = 65
    beers = 62
    beers = 58
    beers = 56
    beers = 51
    beers = 41
    beers = 36
    beers = 30
    beers = 28
    beers = 15
    beers = 14
    beers = 0
    There are now 0 bottles of beer on the wall
>
```

5.5 Complementar

5.5.1 Técnicas de paralelismo.

Fundamentalmente, existem dois tipos diferentes de paralelismo, o paralelismo de tarefas (*task parallelism*), que consiste em dividir múltiplas tarefas compartilhando os dados necessários, e o paralelismo de dados (*data parallelism*), o qual há uma distribuição dos dados sobre as tarefas.

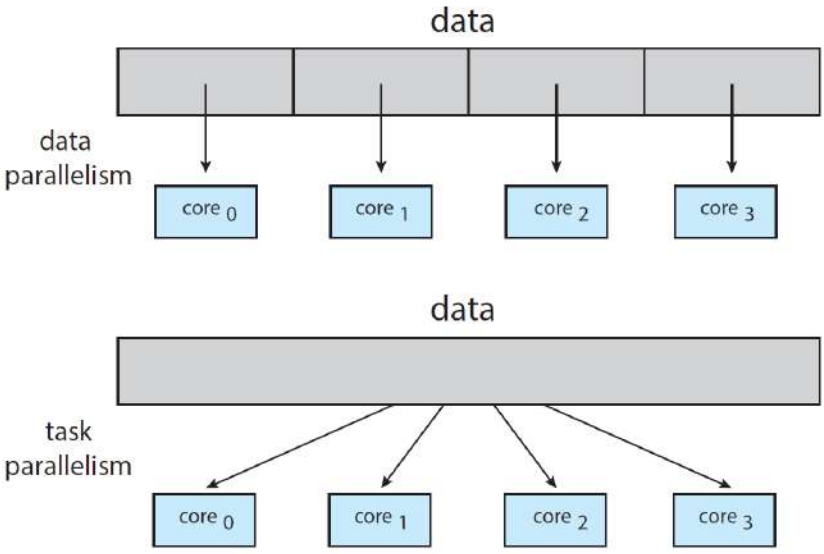

Figura 5.10: Processos com uma ou múltiplas threads

Silberschatz, A. Operating System Concepts, 10th, página 170.

6 Introdução à redes de computadores

A Internet é uma rede mundial de computadores que interconecta, por meio de *communication links*, bilhões de dispositivos *hosts* (anfitriões), também referidos como *end systems* (sistemas finais) por estarem na borda de internet, como os clientes (computadores pessoais) e servidores. Seu funcionamento é análogo ao modal de transporte rodoviário, pois assim como um produto, que ao sair de uma manufatura, deve ser empacotado, carregado no caminhão, transportado através das rodovias, descarregado no destino, e desempacotado, para em fim estar disponível para uso, os dados também passam pelo mesmo processo, pois ao sair de um *host*, devem ser empacotados, carregados no protocolo de transmissão (como *Transmission Control Protocol* e *Internet Protocol*, que definem o formato, utilizando o *struct* na linguagem `C`, e a ordem das mensagens), transportados por um meio físico (como cabos ou espectro eletromagnético), descarregados do protocolo no destino, e desempacotados, para finalmente serem usados por alguma aplicação.

É importante citar que os protocolos de Internet são desenvolvidos pela *Internet Engineering Task Force* (IETF) e os seus documentos são chamados de *requests for comments* (RFCs).

Essa rede pode também ser definida como uma plataforma que oferece serviços de comunicação entre aplicações, tornando-se, dessa forma, análogo à um sistema de correios que oferece serviços aos seus clientes como entrega normal ou expressa.

O acesso à internet é fornecido por ISPs (*Internet Service Providers), que transmitem os dados de forma guiada (*guided media*), através, por exemplo, de cabos de cobre de par trançado, redes de telefone (com o DSL,* Digital Subscriber Line*), cabos de televisão ou fibra optica (com o conceito* fiber to the home*, ou FTTH), ou de forma não guiada (*unguided media**), no qual ondas eletromagnéticas propagam-se pela atmosfera e espaço, com o uso das torres de rádio e dos satélites (como os geoestacionários, que introduzem um atraso de 280 milissegundos na comunicação, e os de baixa órbita).

7 Internet: a rede das redes

Atualmente, é cada vez mais comum encontramos dispositivos, como impressoras e celulares, que são capazes de interconectar-se e trabalhar em conjunto, formando uma rede de comunicação. Nas residências e empresas, essa rede de comunicação é chamada de LAN (*Local Area Network* ou rede de acesso local), pois tem um alcance limitado.

Essas redes podem ter seu alcance ampliado, a partir de uma interconexão, para uma cidade inteira, algo que forma a MAN (*Metropolitan Area Network*). Por fim, é possível amplificar o alcance das MAN's, vinculando-as para, assim, gerar a WAN (*Wide Area Network*), uma rede que engloba regiões e países. A internet é advinda da interconexão de múltiplas WAN's.

O usuário final tem acesso à internet através das ISP's (*Internet Service Provider*) municipais. Essas tem a comunicação estabelecida por ISP's regionais e nacionais (que, geralmente, não fornecem seus serviços aos clientes finais), ou *tier 1 ISP's*. Por fim, essas últimas interconectam-se diretamente ou através de IXP's (*Internet Exchange Point*) e CPN's (*content-provider networks* ou redes de provedores de conteúdo), como o Google, que vincula-se externamente, com os IXP's e ISP's, e internamente, com uma rede privada inacessível ao público. A Figura 7.1 mostra, de uma forma gráfica, a rede de redes.

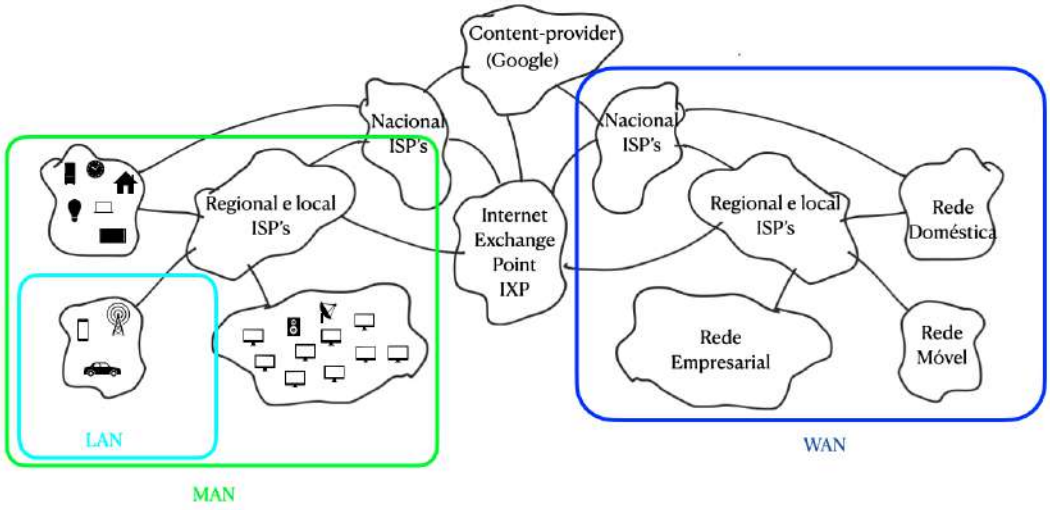

Figura 7.1: Rede de redes

7.1 Dispositivos

De forma simplificada, a rede LAN tem a mesma estrutura das demais, contendo o modem, responsável por conectar diferentes redes, o roteador, o qual gerencia a rota de tráfego dos dados, e o switch, que interconecta diversos dispositivos na mesma rede. É importante dizer que essas funcionalidades podem estar contidas em um ou mais dispositivos.

7.1.1 Roteador

Os roteadores são computadores de propósito específico, sendo otimizados para operar as camadas de rede, enlace e física (a primeira em destaque). Executam programas para configurações, com sua página HTML enviando comandos ao shell do Linux (sistema operacional normalmente encontrado nesses dispositivos). São responsáveis por mover os pacotes de entrada para a sua saída apropriada. Para tal, é utilizado uma tabela de encaminhamento, no qual a determinação do *link* que a mensagem deve ser transmitida é feita a partir do *IP address* (endereço do computador no procolo de internet), que está localizado no *header*

adicionado pela camada *network*. Nos sistemas mais antigos, o algoritmo de roteamento estava contido no roteador, porém, com o aumento da complexidade das redes, esse algoritmo foi desacoplado do roteador, passando para um servidor que atualiza a tabela de encaminhamento (que pode ser entendida como uma configuração da rede), sistema chamado de *software define networks* (trazendo vantagens como a velocidade de operação do roteador). Esse sistema de configuração de rede ocorre no *core* da rede, e não nas bordas, pois, por exemplo, em redes domésticas (que se encontram na borda), isso não é necessário pelo fato de somente haver uma única entrada e saída (assim não precisa de configuração).

7.2 Transmissão

A transmissão dos dados entre os dispositivos é feito através de 5 camadas, iniciando-se a partir da camada de aplicação, que ocorre no *user space*, local de origem da mensagem (enviado em pedaços de dados intitulado de *packets*, ou pacotes), passando por transporte, rede (com ambas estando no *kernel space*), enlace (com uma parte do software no *kernel space* e parte em *hardware*) e física (encontrada somente em *hardware*), no qual cada uma encapsula os dados das camadas anteriores e adiciona o seu *header*, tendo como saída, respectivamente, os chamados *segment*, *datagram*, *frame* e a série de sinais físicos que representam os *bits*. As duas últimas camadas, enlace e física, são implementadas pela NIC (*Network Interface Card*, ou placa de rede).

O receptor desses dados fará o processo inverso, extraindo o *header* de sua respectiva camada de forma a desencapsular o pacote, até que a mensagem seja entregue para a aplicação receptora, como mostrado na Figura 7.2.

É interessante perceber que essa arquitetura de protocolos empilhados transforma cada camada em uma provedora de serviços à camada superior, algo que torna o sistema modular, fácil de ser atualizado e fácil de ser debatido e explicado. Porém, esses sistema de camadas pode conter duplicações de funcionalidade e de informação.

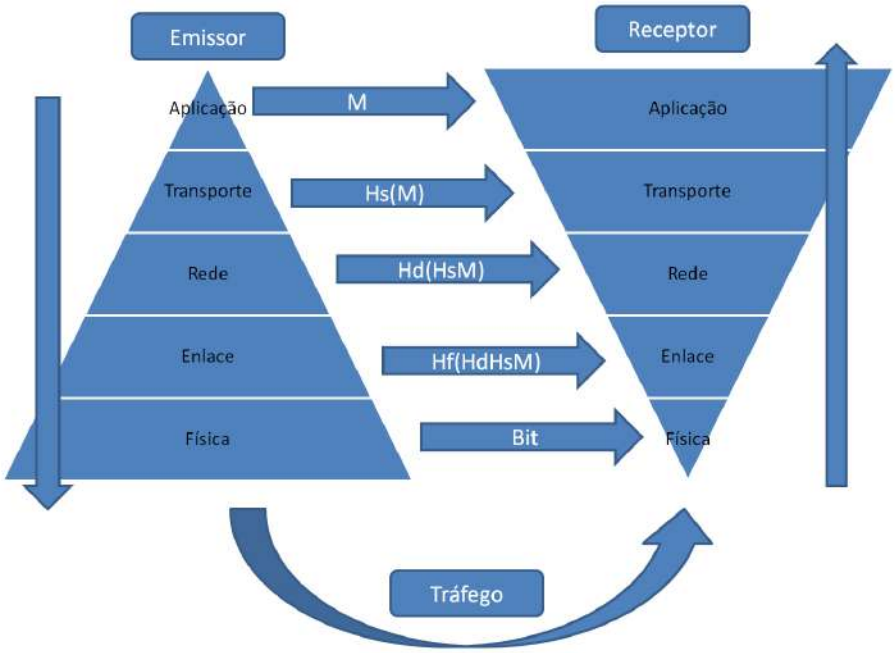

Figura 7.2: Emissor e receptor dos dados

Normalmente a transmissão não ocorre de um modo direto entre o emissor e o receptor, e sim aliado também aos dispositivos já mencionados anteriormente, como o *switch*. Esses aparelhos utilizam de algumas camadas para a sua operação, como a de rede, enlaço e física para o roteador, procedendo com o desencapsulamento dos dados recebidas e o encapsulamento com o seu protocolo, para assim repassá-las adiante.

7.2.1 Packet-switching: store-and-forward

O repasse dos dados é feita utilizando a técnica *store and fowarding*, no qual cada pacote "pula" de um dispositivo para o outro, sendo primeiro armazenado (store) e em seguida repassado (fowarding). Assim a transmissão entre o computador A e o B contendo um roteador como intermediário ocorrerá da seguinte forma:

1. A mensagem é gerada pela aplicação do emissor

2. A mesma é separada em pacotes, e esses vão passar por todas as 5 camadas, recebendo um *header* de cada uma.

3. O *frame* (dados gerados pela camada de enlace), será enviado bit a bit para o intermediário.

4. O intermediário salva (*store*) os bits recebidos até completar o *frame*.

5. O *frame* é repassado (*fowarding*) para o computador B.

6. Em paralelo ao envio, o roteador recebe os dados do *frame* seguinte.

7. O computador B desencapsula o *frame* recebido e armazena o pacote.

8. Esse processo se repete até que todos os pacotes sejam recebidos e utilizados para remontar a mensagem.

Ao ser transmitido, a mensagem demora L/R (*transmission delay*) para sair do computador B até o roteador, no qual L é o tamanho do pacote em bits e R é a velocidade de transmissão em bits/segundo. Considerando que o *transmission delay* é igual entre os computadores A e B e o roteador, o atraso total na transmissão será de $2L/R$. *Similarmente,* 3L/R para 2 pacotes (pois a operação de recebimento e envio do dispositivo intermediário ocorre em simultâneo) e 4*L/R para 3 pacotes. Assim, o atraso total na transmissão pode ser dado por:

```
transmission delay = (n + 1) * L/R
```

Sendo N o número total de pacotes que deve ser transmitido. A Figura 7.3 mostra, graficamente, esse processo.
Considerando L = 10 kbits e R = 100 Mbps, então L/P = 0.1 milissegundos.
Em casos reais, o cálculo do atraso total deve levar em conta, além do *transmission delay* (que está relacionado com o tempo necessário para o dispositivo enviar os dados), outros tipos de atraso como no processamento, atraso de enfileiramento e de propagação (que está relacionado com o atraso na propagação do bit através do meio de transmissão, impactado, portanto, pela distância. É calculado pela divisão

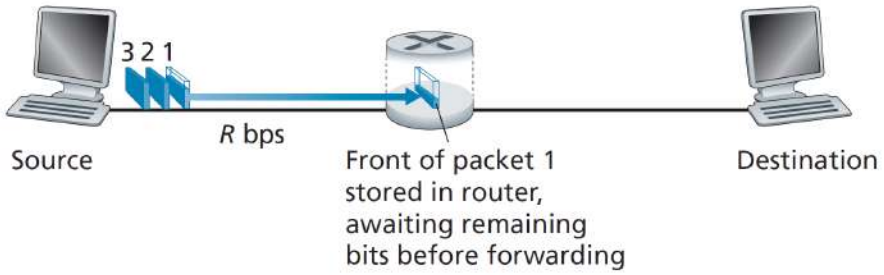

Figura 7.3: Store and Fowarding
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson. Página 24.

entre a distância e a velocidade de propagação). Deve-se destacar que diferentes dispositivos irão fornecer distintas taxas de transmissão e meios de propagação, algo que pode gear um gargalo no sistema (afetando seu desempenho geral).

É importante perceber que, por consequência do limite de memória RAM, pode ocorrer uma insuficiência de espaço para o armazenamento e enfileiramento dos dados para transmissão. Além disso, o padrão *store-fowarding* trás consigo todos os problemas inerentes da arquitetura produtor-consumidor (por causa de sua equivalência). Assim, pacotes podem ser ignorados caso a memória RAM seja utilizada em sua totalidade (algo que envolve outros conceitos como qualidade de serviço e prioridade de transmissão).

Outro desvantagem dessa arquitetura tem haver com o seu impacto no atraso de enfileiramento, pois cada pacote recém chegado deve esperar todos os que entraram anteriormente serem retirados da fila, e como a chegada de pacotes ocorre de forma aleatória (ou imprevisível), não há como determinar exatamente o tempo de enfileiramento. Porém, podemos estimar a intensidade de tráfego (*traffic*

intesity) analisando a média, em bits por segundos, dos pacotes que chegam (La), pela taxa de transmissão (R). Desse modo, pode ocorrer 3 casos, no qual:

1. La/R > 1: fila e atraso crescentes (e, consequentemente, pode haver um estouro na memória RAM, como explicado anteriormente. Dispositivo com capacidade inadequada).
2. La/R = 1: fila e atraso constantes.
3. La/R < 1: fila e atrasos decrescentes ou mínimos.

7.2.2 Analogia com uma caravana

A diferença entre o atraso da transmissão e propagação pode ser melhor entendido fazendo-se uma analogia com uma caravana. Imagine uma caravana (pacote) de 10 carros (10 bits) que percorrem (propagam) em uma velocidade de 100 km/h, com cada carro gastando 12 segundos (transmissão = 1 carro / 12 segundos) com o serviço do pedágio (roteador). O próximo pedágio está a uma distância de 100 km. Assim, o tempo total de percorrer do primeiro pedágio até a chegada ao segundo vai ser a soma do tempo da caravana ser processada pelo primeiro pedágio (10 carros / 1 carro / 12 segundos = 2 minutos de atraso na transmissão *transmission delay*) mais o tempo necessário para percorrer a pista (100km / 100 km/h = 60 minutos), algo que resulta em 62 minutos.

7.2.3 Circuit-swiching

A alternativa ao *packet-switching* é o *circuit-switching* (mostrado na Figura 7.4), no qual faz uma conexão direta (ou circuito) entre os computadores comunicantes (*end-to-end connection*), tendo assim, como vantagem, a garantia de uma taxa de transmissão constante (impossibilita que outros computadores interfiram nessa estabilidade). O uso do circuito é fundamentado na multiplexação, que, por sua vez, pode ser baseada no tempo, com o TDM (*time-division multiplexing*), método em que um *frame* (período fixo de tempo) é fracionando em *slots* de tempo, com cada *slot* sendo dedicado a uma conexão específica (assim, em um *frame*

ocorre diversas comunicações diferentes), ou na frequência, utilizando o FDM (*frequency-division multiplexing*), no qual as frenquências de comunicação de um *link* são divididas entre as conexões.

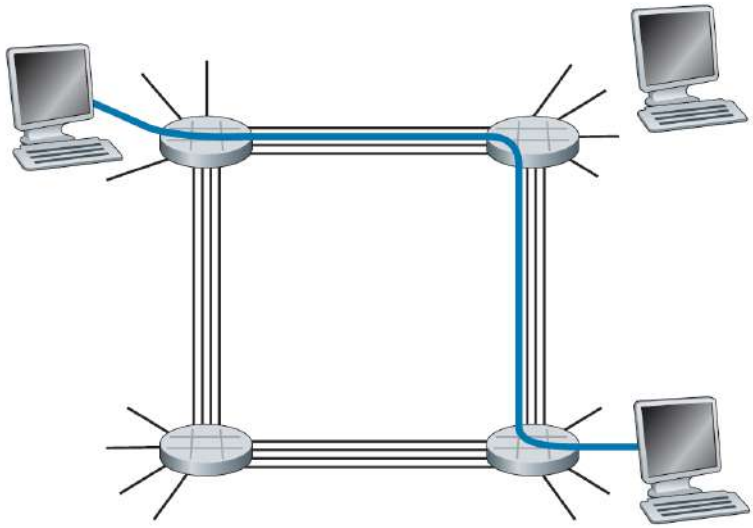

Figura 7.4: Circuit Switching
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson. Página 28.

A desvantagem desse sistema é a ocorrência de ociosidade em períodos no qual os computadores não estão se comunicando (*silent periods*), assim recursos de rede estão sendo alocados e não utilizados, portanto desperdiçados. Outra inferioridade (comparado ao *packet-switching*) vem da complexidade de reservar uma capacidade de transmissão ponta a ponta (*end-to-end transmission*) e de coordenar a sinalização relacionada com a multiplexação.

8 Camada de Aplicação: a razão de existir das redes

A camada de aplicação (primeira camada) é considerada como a mais importante, pois ela é quem gera a utilidade da rede. Assim, a infraestrutura, bem como os protocolos, foram criados para servi-la, e, portanto, sem a aplicação, não existiria as demais camadas. Essa utilidade vem dos benefícios produzidos pela interação de dois processos executados em computadores diferentes e, possivelmente, sistemas operacionais diferentes. Para tal, há duas principais arquiteturas, *client-server* e *peer-to-peer* (P2P).

8.1 HTTP

A arquitetura *client-server* é empregue na aplicação *Web*, o qual usufrui do protocolo *Hypertext Transfer Protocol* (HTTP). Essa arquitetura funciona no

sistema *always on*, no qual o servidor, ou processo que espera para ser contactado no início da interação, deve sempre ficar ativo, e o seu endereço na rede fixado. Os clientes não interagem diretamente entre si. O protocolo de transporte usado é o TCP.

A escalabilidade dessa arquitetura (a capacidade de suportar o aumento do número de clientes) está limitada pela capacidade da infraestrutura já instalada. Assim, o aumento no número de *requests* (derivado do aumento no número de *clients*) deve ser precedido por um aumento na competência do servidor de lidar com os mesmos. Por essa razão, os servidores são comumente hospedados em *data centers*, algo que facilita a escalabilidade, por alocar, dinamicamente, centenas de dispositivos para o processo *server* (o servidor se torna uma entidade processada por múltiplos dispositivos).

O HTTP define a estrutura das mensagens e a forma que as mesmas são trocadas. Essa troca é baseada no padrão *response-request* (a mensagem do *client* é o *request*, e a reposta do *server* o *response*). O *client* (uma navegador como o *Chrome*, por exemplo), definido por ser o processo que inicia a comunicação, necessita do endereço do *server*, chamado de URL, que é composta por dois componentes, o *hostname* (*http://www.google.com/*) e o *pathname* (*/search?q=alguma+coisa*):

```
URL: http://www.google.com/search?q=alguma+coisa
```

A mensagem HTTP *request* segue o seguinte padrão:

Request Line: `GET /somedir/page.html HTTP/1.1 \r\n` (= Método + URL + HTTP *version* + *carriage return and line feed*)

Header lines (line 1): `HOST: www.google.com.br \r\n` (*host* no qual hospeda o objeto requisitado)

Header lines (line 2): `Connection: Mozilla/5.0 \r\n` (navegador *Firefox*)

Header lines (line 3): `Accept-language: fr \r\n` (prefere a versão em língua francesa)

Header lines (line x): `... \r\n` (outras configurações)

Entity body (line y): `\r\n` (pode ficar vazio, como no método *GET*, ou preenchido, como no método *POST*)

A mensagem HTTP *response* segue o seguinte padrão:
Status Line: `HTTP/1.1 200 OK \r\n` (Tudo ocorreu bem)
Header lines (line 1): `Connection: close \r\n` (*server* irá encerrar a conexão)
Header lines (line 2): `Date: Tue, 18 Aug 2015 15:44:04 GMT \r\n` (data em que o *response* foi enviado)
Header lines (line 3): `Server: Apache/2.2.3 \r\n` (nome do servidor)
Header lines (line 4): `Last-Modified: Tue, 18 Aud 2015 15:11:03 GMT` (data da última modificação)
Header lines (line 5): `Content-Length: 6821 \r\n` (número de bytes do objeto enviado - O mesmo está em *Entity body*)
Header lines (line 6): `Content-Type: text/html \r\n` (indica o formato do objeto, no caso uma página html)
Entity body (line y): `... \r\n` (dados do objeto)

8.2 Torrent

A segunda arquitetura citada anteriormente, a P2P, baseia-se na inter-colaboração intermitente dos dispositivos na rede, ou *peers*, de forma que esses podem ser *client* e *server* simultaneamente (É importante perceber que, em uma sessão de comunicação com um par de dispositivos, o *client* é aquele que inicia o contato e o *server* aquele que espera para ser contactado, assim, apesar de cada *peer* assumir ambas as formas simultaneamente nessa rede, elas tem um único papel para cada sessão). Essa arquitetura é auto-escalável (*self-escalability*), pois cada *peer*, apesar de adicionar novos *requests*, adicionam também capacidade de serviço na rede.
A popularidade desse protocolo vem do uso da tecnologia *Torrent*. Desenvolvida pela empresa *BitTorrent*, essa tecnologia descentraliza o fornecimento de arquivos, de forma que os *peers*, conforme fazem o *download*, também proporcionam os dados já baixados de volta para a rede. Assim, a capacidade de transmissão não está limitada pelo *seed* (primeiro fornecedor de um conteúdo), crescendo, e, consequentemente, tornando arquivos populares mais acessíveis, conforme

aumenta a participação dos usuários (sendo, consequentemente, barato, por não necessitar a contratação de uma infraestrutura para o fornecimento de arquivos). Essa arquitetura enfrenta diversos desafios, como: segurança, performance e confiabilidade.

8.3 Sockets

Os *sockets* são uma estrutura de dados em *kernel space* (encontrados dentro do sistema operacional, ou seja, seu acesso é através de uma chamada de sistema) que fornece (analogamente a uma porta) uma interface (API, *Application Programming Interface*) para a interação da camada de aplicação com a camada de transporte. Ele se manifesta através de um FD (*file descriptor*, um arquivo), fornecendo um *pipeline* para o processo (em *user-space*). Para ser executado, o *socket* necessita do endereço do *host* (*internet protocol address* ou *IP address*) e do identificador que especifica o processo receptor no *host* (chamado de *Port Number*). Por padrão, o *web server* utiliza a porta 80, e o *mail server* (com o protocolo SMTP) é identificado pela porta 25.

8.4 Serviços de transporte

Como citado anteriormente, a camada de transporte fornece uma série de serviços para a camada superior. Esses serviços podem ser analisados sob a ótica de: confiabilidade (*reliable data transfer*), taxa de transferência (*throughput*), atraso (*delay* ou *timing*) e segurança (*security*). Nesse sentido, a *internet* prover dois protocolos de transporte, o UDP e o TCP, os quais disponibiliza serviços com características diferentes, de maneira que:
TCP:

1. *Handshaking procedure*: com uma preparação anterior ao envio da mensagem (confiabilidade).

2. *Full duplex connection*: com os processos podendo receber e enviar dados.

3. *Reliable data transfer service*: sendo os dados enviados sem erros e na ordem correta (confiabilidade).

4. *Transport Layer Secutiry: uma melhora no TCP que fornece criptografia (*encryption*), integridade de dados (*data integrity*) e* end-point authentication* (segurança).

5. Garantia de entrega da mensagem.

UDP (um *lightweight transport protocol*):

1. *No handshaking procedure*: Não se sabe se o receptor está preparado para receber a mensagem.

2. *Unreliable data transfer*: podendo ser recebidos dados corrompidos e fora de ordem.

3. Não há garantia de entrega da mensagem.

4. Rápida.

É importante perceber que os serviços de *timing* e *throughput*, apesar de serem necessários para algumas aplicações, como *internet telephony* (Skype), não são fornecidos pelos protocolos de *internet* atuais.

8.5 Complementar

Pesquisar Sobre

1. *Socket programming*: página 152 do livro *Computer Networking a top-down approach 8th ed.*

2. *Dual Stack*: IPv4 e IPv6

3. NAT

4. WireShark: www.wireshark.org

5. ip addr shwo
6. ip config /on
7. telnet 127.0.0.1 3000
8. SS - apt | grep 3000
9. SS - apt | grep telnet
10. nc - V - U 127.0.0.1 1200
11. ncat
12. CDN (*Content Distribution Networks*)
13. DASH (*Dynamic Adaptive Streaming over HTTP*).
14. DNS (*domain name system*): The Internet's Directory Service
15. telnet serverName 25
16. SMTP (*Simple Mail Transfer Protocol*)
17. *Cookies* (*User interaction*)

9 DNS: Domain Name System

Quando um usuário acessa um *site* na Internet, o mesmo digita um nome, por exemplo:

`www.google.com.br`

Sabendo que o sistema de endereçamento utilizado é baseado no TCP/IP, como o navegador é capaz de converter o nome digitado no endereço de IP (*Internet Protocol*) do servidor requisitado ?

Essa tradução é feita pelo *Domain Name System* (DNS).

O DNS é um protocolo executado na camada de aplicação, que roda em UDP (usando a porta 53), no qual fornece o serviço de tradução de *hostnames* (nome dos servidores) em IP *address*. Seu funcionamento é baseado em um sistema de banco de dados distribuídos implementado em uma hierarquia de servidores.

Essa arquitetura adotada, em contraste com o *design* centralizado, foi escolhida por evitar:

1. Único ponto de falha
2. Volume de tráfego
3. Manutenção
4. Distância do usuário (tempo de resposta)

5. Baixa escalabilidade

9.1 Hierarquia e funcionamento

A hierarquia de servidores de DNS é composta por:

0. *Local*: Não pertence a hierarquia, mas é de suma importância. É proporcionado pelo *Internet Service Provider* (ISP)

1. *Root*: cópia de 13 servidores, gerenciados por 12 diferentes organizações, e coordenado pela *Internet Assigned Numbers Authority* (IANA)

2. *top-level domain* (TLD): fornece o IP *address* para os *authoratives servers.* É mantido por diferentes empresas, como o Verisign Global Registry (*.com*) e Educause (*.edu*)

3. *Authoritative*: toda organização com acesso público deve prover servidores DNS (que pode ser mantido por eles ou por terceiros), com o *primary* sendo o servidor principal e o *secondary* o de *backup.*

Voltando ao exemplo do `www.google.com.br`, quando um usuário digita esse *hostname* e aperta enter, o navegador (*browser*) gera uma uma estrutura de dados, chamada de DNS *query message* (ou mensagem de consulta DNS), com uma série de informações, como o nome `www.google.com.br` (*hostname*), que serão enviadas para o *local* DNS *server.*
A partir do *local* DNS *server,* pode ocorrer duas formas de interação: recursiva e iterativa.

Iterativa

No modelo iterativo, o *local server* envia um *request* para cada um dos servidores da hierarquia, como mostrado na Figura 9.1:

1. Usuário: *request* para *local server*

2. *local server*: *request* para o *Root server*
3. *local server*: *response* do *Root server*
4. *local server*: *request* para o TLD *server*
5. *local server*: *response* do TLD *server*
6. *local server*: *request* para o *Authorative server*
7. *local server*: *response* do *Authorative server*
8. Usuário: *response* do *local server*

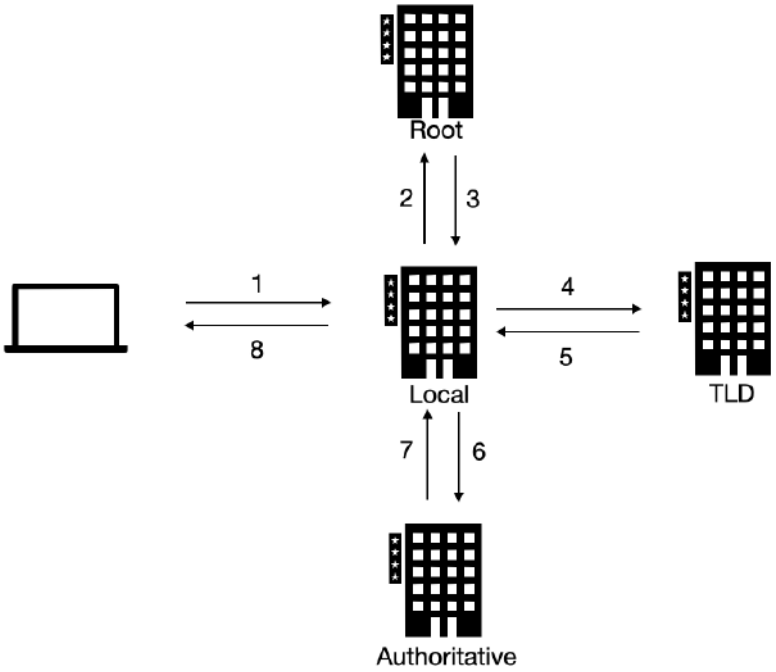

Figura 9.1: Modelo Iterativo

Recursivo

No modelo recursivo, mostrado na Figura 9.2, a sequência de *requests* ocorrem em cadeia entre os servidores:

1. Usuário: *request* para o *local server*
2. *Local server*: *request* para *root server*
3. *Root server*: *request* para TLD *server*
4. TLD *server*: *request* para *Authorative server*
5. TLD *server*: *response* do *Authorative server*
6. *Root server*: *response* do TLD *server*
7. *Local server*: *response* do *root server*
8. Usuário: *response* do *local server*

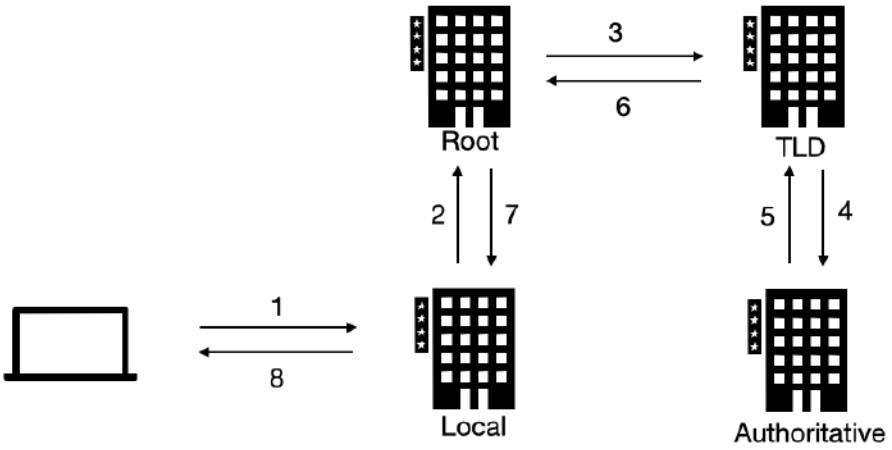

Figura 9.2: Modelo Recursivo

9.2 DNS caching

Nos dois modelos apresentados anteriormente, foi necessário 8 DNS *query messages* para a execução do serviço de tradução de *hostname* para IP *address* (esse número pode ser maior, caso houver *Authorative servers* intermediários). Porém, de forma a minimizar os impactos de tantas requisições, o *local server* utiliza um

sistema de armazenamento temporário (pois o mapeamento entre *hostname* e IP *address* não é permanente, alterando-se com o tempo) da tradução, fazendo com o número de DNS *query messages* caia para apenas 2, o *request* e o *response* entre o usuário e o *local server*.

9.3 Vulnerabilidades

O DNS pode ser visto como um multiplicador de requisições, transformando 1, originada do usuário, em 8, que ocorrem entre os servidores do DNS, algo que o torna especialmente vulnerável à ataques do tipo *Distributed Denial of Service* (DDoS). Para evitar tal fato, vem sendo desenvolvido o DNSSEC (DNS *Security Extensions*), uma versão de segurança do DNS.

9.4 Resource Records

O DNS é um banco de dados distribuídos que armazena uma tupla de 4 elementos, *Name*, *Value*, *Type* e *time to live* (TTL), o qual é utilizado para prover, entre outros, o mapeamento do *hostname* para IP *address*. O dado armazenado em cada um dos 2 primeiros elementos citados da tupla variam conforme o *type*. O último elemento especifica quando o *resource record* deve ser removido do *cache*.

A seguir será mostrado os dados contidos em *Name* e *Value* para cada caso de *Type* (os exemplos mostrados não contém o TTL).

`tupla = (name, value, type, ttl)`

Type A

Name = *hostname*, *Value* = IP *address*

Tupla: `(relay1.bar.foo.com, 145.37.93.126, A)`

Type NS

Name = *domain*, *Value* = *Authorative hostname server*

Tupla: `(foo.com, dns.foo.com, NS)`

Type CNAME

Name = apelido (*alias*) para *hostname* , *Value* = *hostname*

Tupla: `(foo.com, relay1.bar.foo.com, CNAME)`

Type MX

Name = apelido (*alias*) do *mail hostname, Value* = *mail hostname*

Tupla: `(foo.com, mail.bar.foo.com, MX)`

DNS *Messages*

O formato da mensagem DNS é mostrado na Figura 9.3, segue a seguinte estrutura:

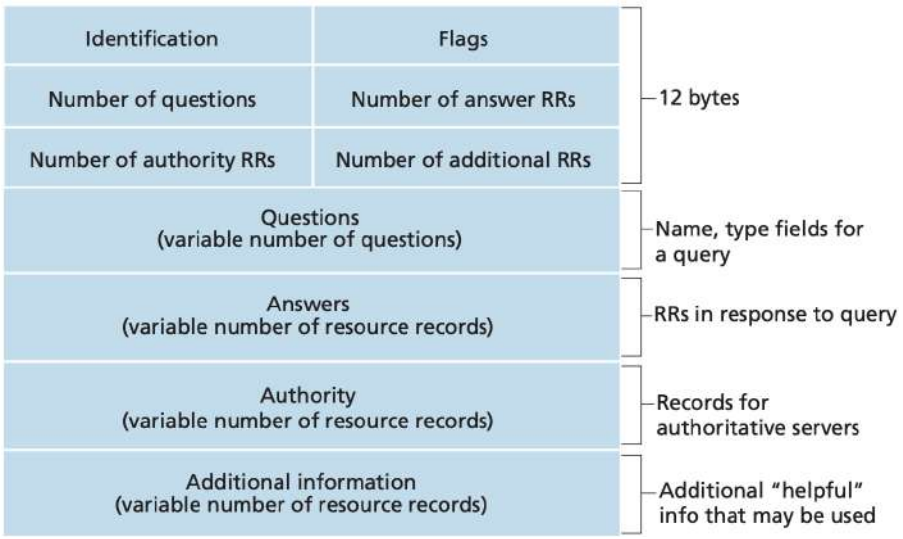

Figura 9.3: Formato do DNS message

1. Identificador: número de 16 bits que identifica a *query*

2. Flags: cada flag é um indicador de 1 bit, no qual:

   2.1. *query*/*reply*: indica se é uma *query* (0) ou um *reply* (1)

   2.2. *authoritative flag*: indica se o *response* vem de um *authorative server*

   2.3. *recursion-desire*: indica o desejo do modelo recursão para as requisições (como mostrado anteriormente)

2.4. *recursion-available field*: ajustado na resposta, indica se o DNS *server* suporta recursão

3. *Number fields*: cada campo indica o número de ocorrências de cada sessão de dados
4. *Questions*: contém informações como: *host address* e *name*, para *type* A.
5. *Answers*: populado pelo DNS *server*, contém os *resource records* para o *hostname* requisitado.
6. *Authority*: *records* de *authority servers*
7. *Additional information*: contém outras informações interessantes.

Adicionando RR no banco de dados DNS A verificação (de unicidade) e adição de um novo RR no DNS é feito através de entidades comerciais chamadas de *registrar*, as quais são acreditadas pela *Internet Assigned Names and Numbers* (ICANN). Para a adição, é necessário prover o *domain name*, por exemplo `google.com.br`, e o endereço de IP do DNS primário e secundário.

9.5 Complementar

9.5.1 Pesquisar Sobre

1. https://alexanderell.is/posts/rpc-from-scratch/
2. https://datatracker.ietf.org/doc/html/rfc/
3. DNS over HTTPS (DoH): https://datatracker.ietf.org/doc/html/rfc8484
4. DNS over TLS (DoT): https://datatracker.ietf.org/doc/html/rfc8310

10 Transport-Layer

O objetivo da camada de transporte (*transport layer*), executada nos dispositivos localizados nas pontas da comunicação, é estender as funcionalidades da *network layer* com a preparação e envio dos dados transmitidos pelos diferentes *sockets*, técnica chamada de `multiplexação`, e o direcionamento do pacote recebido para o *socket* competente, procedimento chamado de `demultiplexação`. Dessa maneira, do ponto de vista da aplicação, a *transport layer* provê uma comunicação lógica, como se os processos estivessem interconectados diretamente [algo similar ocorre na *network layer*]. Porém os diferentes protocolos existentes nessa camada podem fornecer serviços adicionais, como confiabilidade da transferência dos dados e controle de congestionamento, ambos presentes no protocolo TCP (*Transmission Control Protocol*, ou Protocolo de Controle de Transmissão) e não encontrados no UDP (*User Datagram Protocol*).

A `multiplexação` e `demultiplexação`, ambas demonstradas graficamente na Figura 10.1, são possíveis devido à estrutura de dados *4-tuple*, no qual está contido os endereços de IP da origem e do destino, e os identificadores únicos de 16 *bits*, chamado de porta (*port*), dos *sockets* de origem e destino.

Diferentemente do UDP, no qual a demultiplexação ocorre com o encaminhamento dos dados diretamente para a porta de destino, descrita

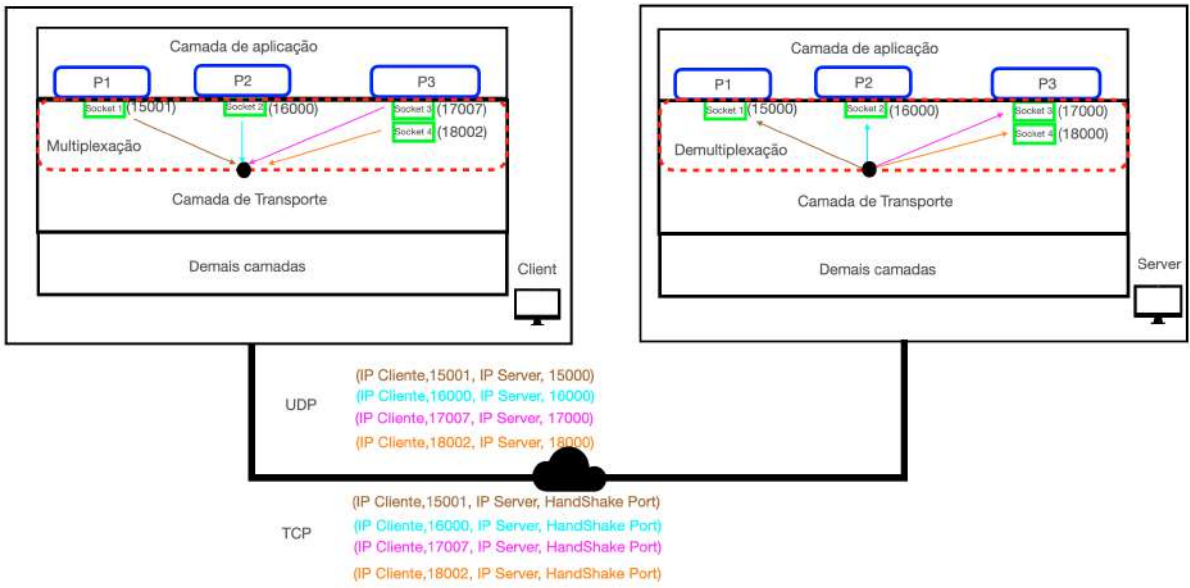

Figura 10.1: Multiplexação e Demultiplexação

no *header* do *segment*, (ou seja, o *socket* UDP é completamente identificável com *2-tuple*, ou estrutura de dados que contém o IP e *port* de destino), o TCP necessita da *4-tuple* completa para a identificação do respectivo socket. Essa abordagem simplifica a arquitetura *client-server*, pois o servidor pode utilizar de um única porta (como uma *well-known port number*, número de porta que varia de 0 até 1023) para a execução do *handshaking*, como a 80 no caso do HTTP, enquanto que entrega dinamismo à criação (ocorrendo no *request*) e finalização dos *sockets*, algo que sucede o encerramento do canal de comunicação. Na Figura 10.1 pode ser observado como os dados contidos na *4-tuple* podem variar conforme o protocolo escolhido.

É importante perceber que um processo pode ler e escrever no `file descriptor` (*socket*) criado para cada conexão iniciada. E múltiplas conexões podem ser manipuladas por um único processo, vinculando-se o *socket* correspondente a uma thread responsável pelo processamento das

requisições do cliente. Desse modo, um processo pode possuir múltiplos *sockets* e interagir individualmente com eles pelo `file descriptor` correspondente na thread específica para a conexão em questão.

Dessa maneira, pode-se salientar uma forma de operação dos servidores, no qual cada *request* recebido gera um novo conjunto *thread* e *socket* e o fim da conexão encerra esse conjunto. Portanto, o período de vida do *thread* e *socket* pode ser longo, durando toda a comunicação no modo persistente, ou curto, com a conexão encerrando-se logo após o envio do *response* no modo não persistente. Requisições frequentes no modo não persistente podem impactar no desempenho do sistema.

Como criar uma thread ou processo para cada requisição é computacionalmente custoso, os *servers* atuais implementam um sistema produtor-consumidor, como mostrado na Figura 10.2, composto por uma *thread* produtora e um *pool of threads* consumidoras. A *thread* produtora, vinculada à um *socket*, recebe as requisições oriundas dos *clients* e deposita-os em uma fila chamada *task queue* (*buffer*). Essas requisições serão direcionadas para *threads* consumidoras disponíveis, oriundas do *pool of threads*, as quais manterão-se ocupadas processando as respectivas requisições coletadas (ou seja, não estarão disponíveis e, portanto, não poderão adquirir novas requisições).

[Para saber mais, acesse: https://httpd.apache.org/docs/2.4/mod/worker.html e https://www.nginx.com/blog/thread-pools-boost-performance-9x/]

10.1 Reliable Data Transfer Protocol

Para entender melhor sobre como funciona uma transferência de dados confiável (*Reliable Data Transfer Protocol*, RDT), será adotada uma máquina computacional abstrada com um número finito de estados, no qual assume somente um único estado por vez, chamada de Máquina de Estados Finitos (*Finite-State Machine*, FSM).

A cada nível de RDT será adicionado uma nova camada de complexidade, até chegar no modelo mais próximo do real.

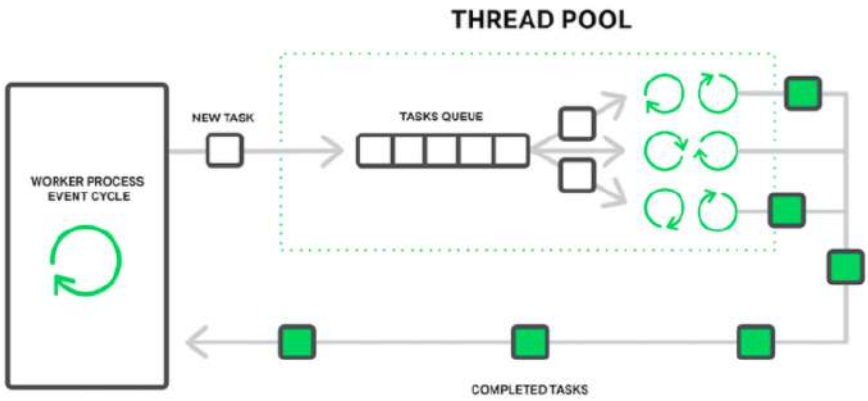

Figura 10.2: Pool of Threads
Imagem retirada de: https://www.nginx.com/blog/thread-pools-boost-performance-9x/ em 05/12/2021.

10.1.1 RDT 1.0

No primeiro caso, consideramos que as camadas mais baixas são confiáveis. Ou seja não há perda ou alteração de dados e nem alteração na ordem de envio. Assim:
Emissor:

1. rdt_send(data)
2. packet=make_pkt(data)
3. udt_send(packet)

Receptor:

4. rdt_rcv(packet)
5. extract(packet,data)
6. deliver_data(data)

10.1.2 RDT 2.0

Em RDT 2.0, vamos considerar que, durante a transmissão, algum bit pode ter sido corrompido. Assim, é necessário utilizar o protocolo ARQ (*Automatic Repeat reQuest*), o qual é baseado em três pontos: detecção de erro, permitindo o receptor reconhecer a ocorrência de erro; o *feedback* do receptor, com o parecer positivo (equivalente à "entendi!") chamado de ACK ou negativo (equivalente à "pode repetir?"), denominado de NAK (em princípio, esse retorno basta ser de 1 bit de tamanho, sendo 0 para negativo e 1 para positivo); e retrasmissão, com o emissor reenviando os pacotes caso tenha recebido uma negativa (NAK) do receptor.
Emissor:
Envia os dados

1. rdt_send(data)
2. sndpkt = make_pkt(data,checksum)
3. udt_send(sndpkt)

Espera por uma resposta
(reemissão em caso de NAK) 7. rdt_rcv(rcvpkt) && isNAK(rcvpkt)
8. udt_send(sndpkt)
(Encerra estado em caso de ACK)
9. rdt_rcv(rcvpkt) && isACK(rcvpkt)
Receptor: (Caso dados corrompidos)

4. rdt_rcv(rcvpkt) && corrupt(rcvpkt)

(Resposta)
5. sndpkt=make_pkt(NAK)
6. udt_send(sndpkt)
(Caso dados não-corrompidos)

4. rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

5. extract(rcvpkt,data)

6. deliver_data(data)

(Resposta)
7. sndpkt = make_pkt(ACK)
8. udt_send(sndpkt)

É importante perceber alguns detalhes desse protocolo. Primeiro, esse protocolo é conhecido por *stop-and-wait* pois o emissor não pode receber nenhum comando do seu operador enquanto estiver esperando uma resposta do receptor (novos comandos só correm quando a máquina estiver em um estado apropriado). Segundo, não foi considerada a possibilidade do ACK e NAK estarem corrompidos. Para o segundo caso, há duas variações do RDT 2.0 no qual é implementado um protocolo análogo ao incremento feito de RDT 1.0 para 2.0, chamados de RDT 2.1 e 2.2.

10.1.3 RDT 3.0

Por fim, vamos considerar que poderá haver pacotes, ou suas respostas ACK, perdidos durante a transmissão. A solução é o emissor esperar tempo suficiente no qual garanta que os dados enviados ou respondidos foram perdidos. Assim, caso o tempo seja ultrapassado, o emissor deve reenviar os pacotes.

Como pode-se observar, não há como determinar um tempo "garantidor", pois um atraso particularmente alto experimentado pelos pacotes pode ser suficiente para ultrapassar o tempo estimado.

A determinação do tempo sofre de uma dicotomia, pois há vantagens e desvantagens tanto com o aumento como com a diminuição do mesmo. Quanto menor for o tempo, maior a chance de ocorrer falsas perdas (ocasionando duplicações na emissão) por consequência de atrasos na rede. Porém, incrementos nesse tempo podem impactar na velocidade da resposta do emissor, pois o mesmo poderá ficar longos períodos de inatividade aguardando uma resposta (perdida durante a transmissão). Assim, a melhora na efetividade do

protocolo passa pela determinação de um tempo de espera ótimo (provavelmente desenvolvido para o atraso mais frequente).

A Figura 10.3 mostra um exemplo do funcionamento do protocolo RDT 3.0 (também conhecido por *alternating-bit protocol*, por causa da alternância no número da sequência dos pacotes).

10.1.4 Performance

O fundamento dos protocolos mencionados é enviar 1 pacote e esperar por sua resposta. Uma forma de melhorar a performance é utilizar um padrão de enviar múltiplos pacotes antes de entrar no estado de espera da resposta de cada um, método chamado de *pipeline*, como mostrado na Figura 10.4.

Erros na abordagem do *pipeline* são abordados nos protocolos *go-back-n* e *selective repeat*, ambos tratados na próxima aula.

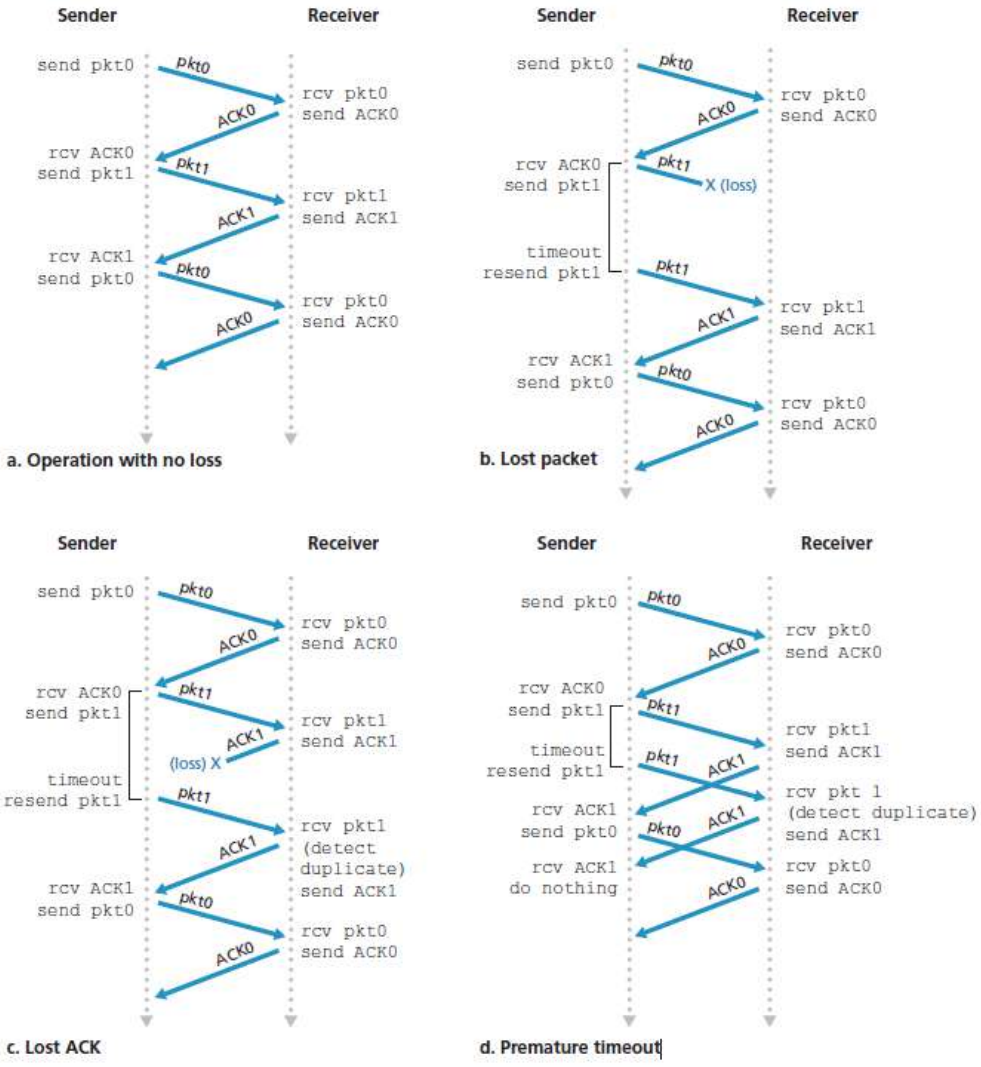

Figura 10.3: Protocolo RDT 3.0

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 212.

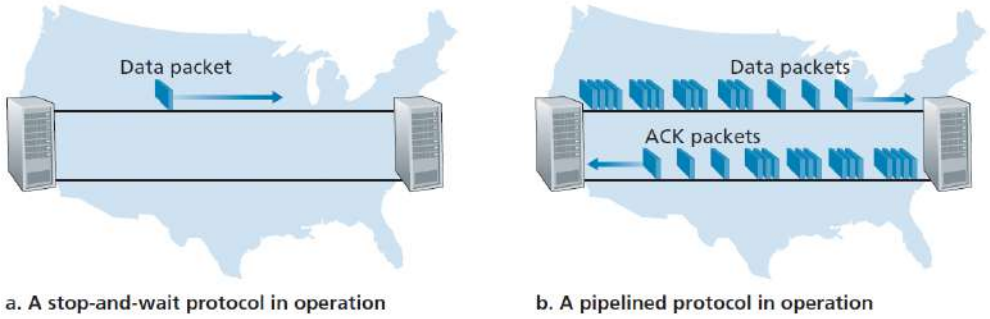

Figura 10.4: Stop-and-wait vs pipelined

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 213.

11 Continuação: Transport Layer

Como citado na aula anterior, o protocolo *stop and wait*, no qual o envio do próximo segmento (também é chamado de pacote por uma questão histórica) ocorre somente após o recebimento da resposta do segmento anterior, é uma forma ineficiente para a transferência de dados confiáveis (*reliable data transfer*, rdt) pois, após a transmissão de um segmento, os recursos disponíveis para a conexão ficarão ociosos até a chegada da resposta do segmento enviado. Dessa maneira, objetivando o aumento da eficiência do mesmo, fora propostos duas soluções: *Go-Back-N* (GBN); e *selective repeat*

11.0.1 GO-BACK-N

A ideia do protocolo GBN é baseado em um subconjunto de N elementos da fila de transmissão (um dos motivos para a imposição de um tamanho limite é o controle de fluxo). Os elementos dessa fila são compostos por espaços que podem ser preenchidos por segmentos oriundo das camadas superiores. Esse subconjunto, chamado de janela, contém os espaços preenchidos por segmentos enviados mas

sem confirmação (*acknowledged*) e espaços ainda não preenchidos. Ao receber uma resposta, o espaço relacionado ao respectivo segmentos sai da janela, um novo elemento da fila de transmissão é adicionado, gerando o efeito de deslizar da janela para a direita na fila de transmissão. Devido a esse efeito o GBN é chamado de *sliding-window protocol*.

Caso todos os espaços disponibilizados pela janela estejam preenchidos, novos dados não poderão ser aceitos, retornando às camadas superiores, sendo esse retorno uma indicação de indisponibilidade.

Na metade superior da Figura 11.1 podem ser reconhecidos os parâmetros: `base`, que identifica o valor inicial incluído; `nextseqnum`, referente ao próximo elemento a ser enviado; e `send window size`, tamanho da janela (valor do N supracitado). A metade inferior mostra o registro dos eventos ocorridos durante o protocolo.

Figura 11.1: Go-Back-N

Disponível em: https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html

Como o envio dos segmentos é feito em ordem, é esperado que os respectivos ACK's sejam recebidos em ordem (*cumulative acknowledgment*). Caso o *server* receba um segmento corrompido ou fora de ordem, o mesmo é descartado e um ACK referente ao último segmento íntegro ordenado é disparado. O ACK duplicado recebido é descartado. Da perspectiva do *client*, a não recepção ACK correspondente ao segmento enviado, pode resultar em dois casos. Primeiro, se a recepção do ACK 'x + 1' ocorrer, porém a do 'x' não, o GBN considerará que o segmento 'x' foi recebido corretamente pelo *server* e o seu ACK fora perdido durante a transmissão, marcando, portanto, o segmento 'x' como enviado corretamente. O segundo caso é a respeito da não recepção de respostas dentro de um período predeterminado (*timeout*), algo que resulta na retransmissão dos segmentos relativos.
Exemplo 1:

1. *cliente* dispara 5 segmentos (1, 2, 3, 4, 5)
2. *server* recebe o segmento número 2 corrompido
3. *server* dispara os seguintes ACK's: 1, 1, 1, 1, 1
4. *cliente* recebe 5 ACK's referentes ao segmento 1, indicando a necessidade de retransmissão dos segmentos 2, 3, 4 e 5.

Exemplo 2:

1. *client* dispara 5 segmentos (1, 2, 3, 4, 5)
2. *server* recebe todos os segmentos corretamente
3. *server* dispara os seguintes ACK's: 1, 2, 3, 4, 5
4. *client* somente o ACK número 5, tendo, o resto, sido perdido durante a transmissão.
5. *client* qualifica todos os 5 segmentos como tendo sido recebidos corretamente pelo *server*

11.0.2 Selective Repeat (SR)

É importante perceber, como mostrado no Exemplo 1, que um único elemento corrompido pode causar a retransmissão de uma série de segmentos, tornando, assim, o GBN ineficiente para esses casos. O aumento dessa ineficiência é diretamente proporcional ao número de erros provocados pelo canal de transmissão.

O protocolo *Selective Repeat* (SR), como o próprio nome já induz, tem o objetivo de diminuir o número de retransmissões desnecessárias. Para tal, utiliza um subconjunto de N elementos da fila de transmissão, chamada de janela, com cada elemento sendo um espaço que pode ser preenchido por um segmento e marcado como não usável, usável, enviado e confirmado, algo análogo ao GBN. Porém, diferencia-se pelo seu comportamento, como mostrado na Figura 11.2.

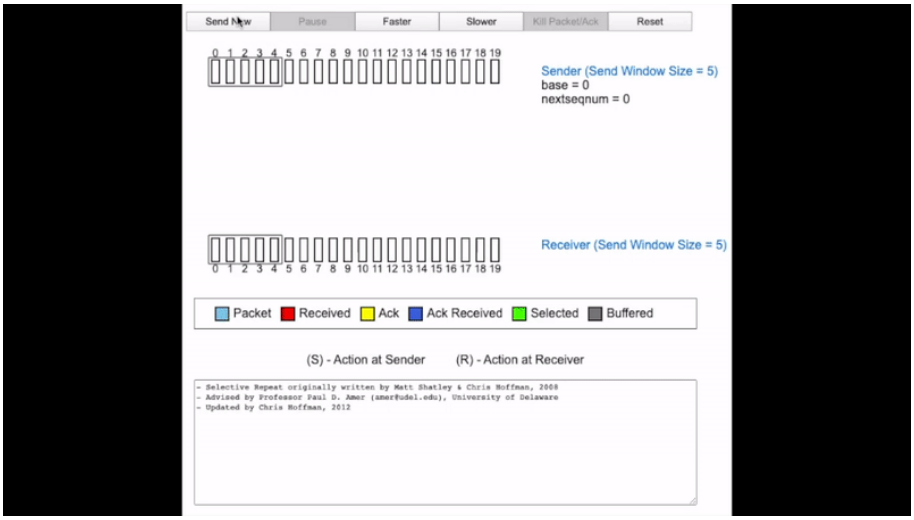

Figura 11.2: Selective Repeat

Disponível em: https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactivean
repeat-protocol/index.html

1. um elemento só é marcado como confirmado quando o mesmo receber seu respectivo ACK.

2. a janela desloca-se somente após o recebimento do ACK relativo ao elemento na posição `base`, localizado no início da janela.

3. o *server* enviará um ACK para cada segmento mesmo se ele estiver fora de ordem (mas dentro das condições citadas a seguir).

4. o *client* terá uma janela própria (do mesmo tamanho da janela do *client*), organizando-a, também, com 4 marcadores: esperado; fora de ordem; aceitado; não usável.

5. um segmento recebido fora de ordem não será descartado e sim armazenado em uma memória temporária (*buffer*) (novamente, dentro das condições citadas a seguir).

Esse comportamento gera uma possível dessincronização da posição das janelas do *cliente* e do *server*, pois o *server* pode receber adequadamente um segmento mas o seu ACK relativo ter sido corrompido ou perdido durante a transmissão. O pior caso de dessincronização ocorre quando todos os ACK's enviados tenham sido perdidos e, consequentemente, o *server* estará adiantado em `N` elementos. Assim, caso o *server* receba um segmento com o número da sequência entre entre os intervalos:

1. `[‘base‘, ‘base‘ + ‘N‘ − 1]: armazenar em memória temporária e enviar o ACK.`

2. `[‘base‘−N, ‘base‘ − 1]: reenviar o ACK.`

3. Fora dos anteriores: ignorar.

A dessincronização pode causar a recepção de um segmento já recebido, gerando um dilema para o receptor: o segmento recebido é novo ou é uma retransmissão ? Essa possível dessincronização pode causar um dilema para o receptor dos dados, pois, como mostrado na A Figura 11.3 mostra o dilema ocasionado pela possível

dessincronização: os dados recebidos são derivados de uma retransmissão, como mostrado em (a), ou de um novo segmento (b)?
Colocações importantes:

1. O número de sequência é um valor finito determinado pelo número de bits disponibilizados para tal.

2. O *buffer* do receptor deve poder armazenar 2 janelas, ou seja, o seu tamanho deve ser, no mínimo, o dobro do `window size` (`N`).

3. O `N` é determinado durante o *handshaking*, limitado pelo tamanho do *buffer* do receptor (como citado em 2).

11.0.3 TCP

A camada de aplicação (*application layer*), para o envio de um arquivo, conta com os serviços de transmissão de dados confiáveis fornecidos pelo *Transmission Control Protocol* (TCP), protocolo da camada de transporte. O TCP, ao receber o arquivo oriundo da *application layer*, divide-o em pedaços de comprimentos iguais chamados de *chunk's of data* (com exceção do último, normalmente menor), de tamanho igual ao MSS (*maximum segment size*), normalmente de 1460 *bytes*. O encapsulamento de um *chunk of data*, etapa que une o mesmo com o *header* do TCP, resulta em um conjunto de dados chamado de segmento.
O parâmetro MSS é determinado pelo MTU (*Maximum Trasmission Unit*), tamanho máximo do *frame* da camada de enlace (*link-layer*), que tem como objetivo garantir que o segmento TCP somado com o *header* do TCP/IP (tipicamente 40 bytes), caberá no *frame* do *link-layer*.
Após sua formação, o segmento é inserido no *send buffer*, memória temporária destinada ao envio dos dados para as camadas inferiores. Essa memória é acessada de tempos em tempos para o envio dos segmentos ali presentes.
Porém, diferente do UDP, o TCP é um protocolo orientado à conexão (*connection-oriented*), pois o primeiro contato entre dois dispositivos ocorre com base no procedimento *3-way handshake*, o qual visa assegurar à confiabilidade na

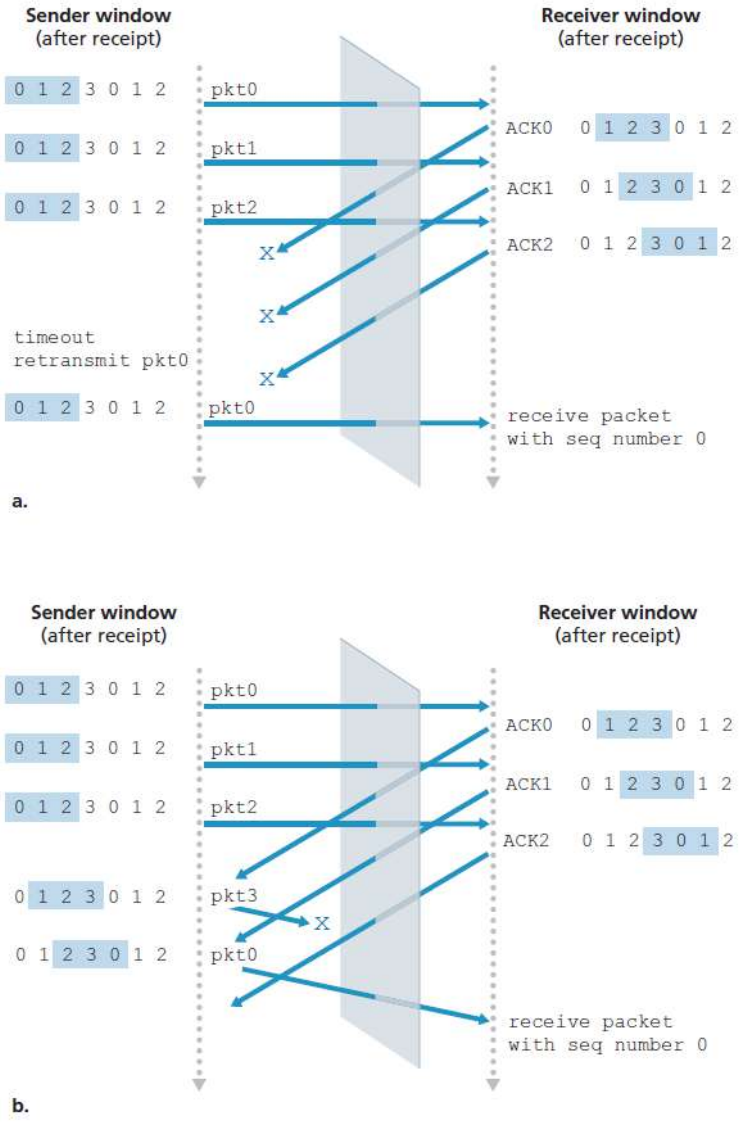

Figura 11.3: Dilema do Selective Repeat

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 225.

transmissão dos dados a partir da definição dos parâmetros da conexão. O *3-way handshake* é caracterizado pela transmissão de 3 segmentos especiais. O primeiro é emitido pelo *client* visando o início da conexão. O segundo é uma resposta do *server*, indicando que o segmento foi recebido corretamente. Por fim, o *client* confirma que também recebeu o segmento oriundo do *server*. Os dois primeiros segmentos especiais não contém dados, mas o terceiro pode conter.
A seguir está listado as características principais do TCP (mencionado em aulas anteriores):

1. Ponto-a-ponto
2. Transmissão de confiança, com fluxo de dados ordenado
3. *Full Duplex*: fluxo de dados bi-direcional na mesma conexão
4. *cumulative* ACK's
5. *Pipelining*: controle de fluxo e congestionamento
6. *Connection-oriented*
7. Fluxo controlado: o emissor não irá sobrecarregar o receptor

Estrutura do segmento TCP

Como citado anteriormente, o segmento do TCP é composto por seu *header* e pelo pedaço do dado enviado pela camada de aplicação. O *header* é a sessão do segmento responsável pelos parâmetros de conexão. São eles:

1. números das portas de origem e destino, utilizados na multiplexação e demultiplexação, respectivamente.
2. *checksum field*, importante na validação da integralidade dos dados recebidos
3. 32-bit *sequence number field*
4. 32-bit *acknowledgment number field*

5. 16-bit *receiver window*: usado para *flow control*, indica o número de bytes que o receptor está disposto a aceitar

6. 4-bit *header length field*: especifica o tamanho do *header* do TCP (como, normalmente, o *options field* não é populado, o tamanho típico do *header* é de 20 *bytes*)

7. *options field*: é útil para a negociação do MSS entre outros.

8. *flag field*: campo que contém 6 bits 8.1. bit ACK: indica a validade do campo ACK 8.2. bits RST, SYN e FIN: são usados para configuração das conexões 8.3. bits CWR e ECE: notificações de congestionamento 8.4. bit PSH: indica que o receptor deve repassar, de imediato, os dados recebidos para as camadas superiores 8.5. bit URG: marcado pela camada de aplicação do emissor, indica que há dados urgentes (Na prática, os bits PSH e URG não são usados)

A Figura 11.4 mostra toda a estrutura do segmento TCP.

Sequência e ACK

O TCP vê os dados como um conjunto ordenado e não estruturado de fluxo (*stream*) de bytes, de forma que o *sequence number* é uma referência à ordem dos bytes (mais especificamente, a ordem do primeiro *byte* dos dados do segmento) e não da série de segmentos enviados. Assim, para um arquivo de 500.000 *bytes* (500 kB) e um MSS de 1.000 bytes (1 kB) serão construídos 500 segmentos, com o primeiro assumindo o *sequence number* de 0, o segundo 1000, o terceiro 2000, e assim em diante.

Já *acknowledgment number* (ACK *number*) é relativo ao *sequence number* do próximo *byte*. Seguindo o exemplo anterior, está contido, no primeiro segmento, 1000 bytes, e o seu *sequence number* é de 0 (portanto existem nesse segmento os *bytes* 0 até 999). Assim, após a chegada no *byte* 999, o receptor enviará a confirmação da recepção desse segmento com o *acknowledgment number* de 1000 (byte seguinte ao último recebido). Dessa maneira, como o receptor só confirma (*acknowledges*) o primeiro byte ausente, no caso, o byte 1000, o protocolo TCP é dito como provedor de *cumulative acknowledgments*.

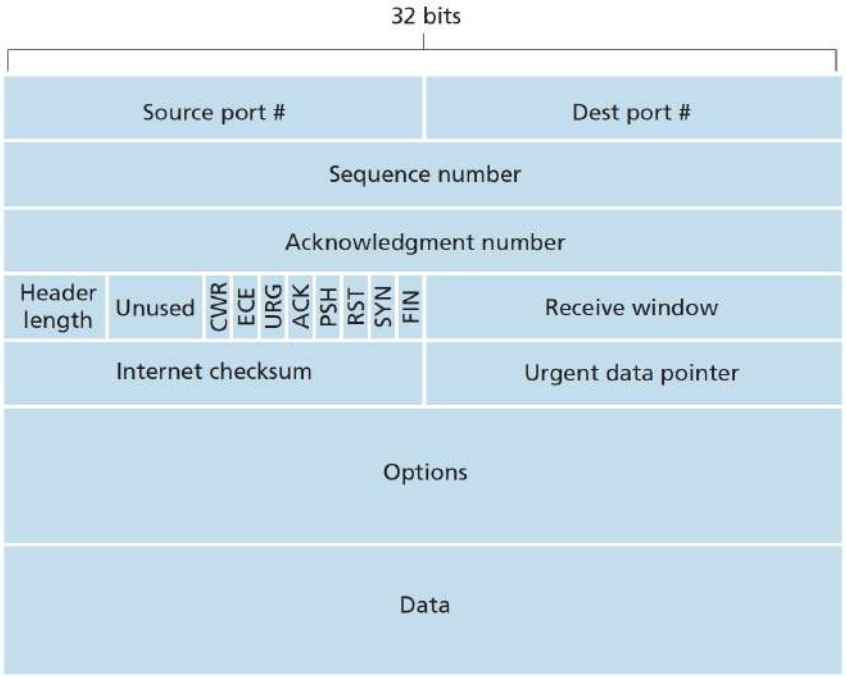

Figura 11.4: Estrutura do segmento TCP
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 231.

A Figura 11.5 mostra um exemplo de como a variação do *sequence number* e do *acknowledgment number* ocorrem com o MSS de 1 *byte* no Telnet. O *Host A* envia seu *byte* 42 (*sequence number*) requisitando (ACK) o byte de *sequence number* 79 do *Host B*, e o *Host B* responde com o seu *byte* 79 e requisita o *byte* 43 do *Host A*.

Segmentos fora de ordem

Como citado anteriormente, o tratamento dos segmentos fora de ordem, para um sistema confiável de transferência de dados, pode seguir uma dessas suas estratégias, *GO-BACK-N* ou *Selective Repeat* (SR). Por uma questão de eficiência, o protocolo SR é a abordagem praticada.

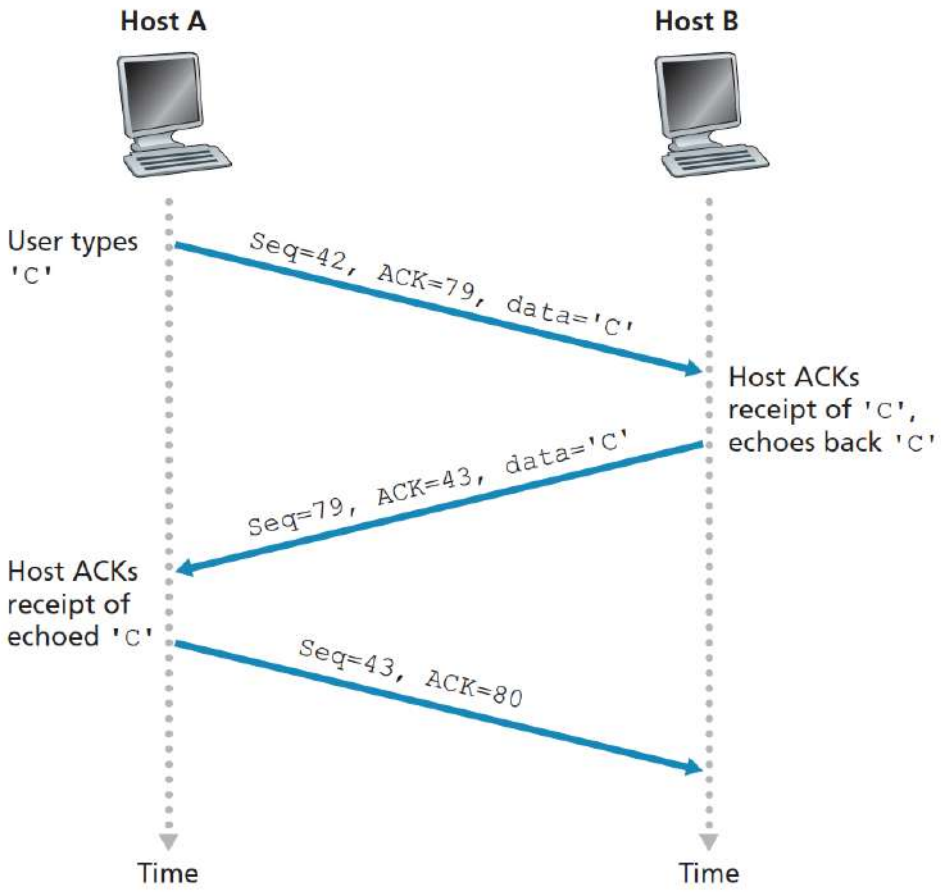

Figura 11.5: Sequence number e ACK

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 234.

RTT e Timeout

Depois do envio de um segmento, ao fim de qual intervalo de tempo o TCP deve considerar que os dados foram perdidos (`timeout interval`) ? Esse tempo sofre de uma dicotomia, pois tanto períodos pequenos como grandes tornam a

comunicação ineficiente por causar retransmissões desnecessárias e aumentar o atraso na retransmissão de segmentos perdidos, respectivamente.

Podemos considerar que, no mínimo, o tempo esperado deve superar o *Round-Trip Time* (RTT), período entre o envio de um dado e a chegada de sua resposta. Como pode-se imaginar, por consequência da não previsibilidade de uso [da rede] e da ocorrência de erros, as condições presentes na rede são variáveis (como um possível congestionamento), algo que impacta diretamente no (RTT), tornando-o, também, variável.

Assim, a determinação do `timeout interval` passa por um cálculo estatístico, definido pelo `SampleRTT`, uma amostra desse período medida de tempos em tempos, `EstimatedRTT`, uma estimativa do valor do RTT que utiliza a técnica da média móvel exponencialmente ponderada (EWMA, *Exponential Weighted Moving Average*), e `DevRTT`, uma estimativa de quanto o `SampleRTT` desvia do `EstimatedRTT`. Os cálculos podem ser vistos a seguir.

Valores recomendados [RFC 6298]: = 0.125 (1/8) = 0.25 (1/4)

```
EstimatedRTT = (1 - ) * EstimatedRTT +  * SampleRTT

DevRTT = (1 - ) * DevRTT +  * | SampleRTT - EstimatedRTT |
```

Para questões de eficiência, é interessante manter o valor do *timeout interval* algo como o valor estimado do RTT (`EstimatedRTT`) mais uma margem que se adéque à flutuação de valor do `SampleRTT` (`DevRTT`). Dessa maneira, chegamos do cálculo a seguir (sendo 1 segundo o valor inicial):

```
TimeoutInterval = EstimatedRTT + 4 * DevRTT
```

11.1 Complementar

11.1.1 Pesquisar Sobre

1. Flow Control (Computer Networking a top-down approach. 8th ed. Pearson, página 246)

2. TCP Connection Management (Computer Networking a top-down approach. 8th ed. Pearson, página 249)

3. Congestion Control (Computer Networking a top-down approach. 8th ed. Pearson, página 255)

4. TCP Congestion Control (Computer Networking a top-down approach. 8th ed. Pearson, página 263) (TCP Cubic, Fairness)

5. Evolution of Transport-Layer Functionality (Computer Networking a top-down approach. 8th ed. Pearson, página 279) (QUIC (*Quick UDP Internet Connection*))

12 IP, o protocolo da Network Layer

O objetivo da *network layer* (camada de rede) é transferir dados de um *host* emissor para o *host* receptor (como não há garantias, esse serviço é conhecido como *best-effort service*). Para tal, é necessário determinar a rota (*route* ou *path*) global que os dados devem pecorrer para alcançar o seu destino, chamado de *routing*, bem como especificar o caminho local com o direcionamento dos dados recém chegados no roteador para a sua saída apropriada, chamado de *fowarding*.

Dessa forma, a *network layer* pode ser decomposta em duas partes, *control plane* e *data plane*, nas quais estão contidos o *routing* e o *fowarding*, como mostrado na Figura 12.1.

É importante perceber que apesar dessas duas funcionalidades serem requisitos para a *network layer*, é possível encontrá-las em dispositivos separados, algo possibilitado pelo SDN (Software-Defined Networks), que aloca o *control plane* em um servidor, algo que torna os roteadores especialistas em *fowarding*.

A *fowarding table*, responsável por determinar, a partir do *header* dos dados recebidos pelo roteador, a saída apropriada, é o elemento chave para a interação

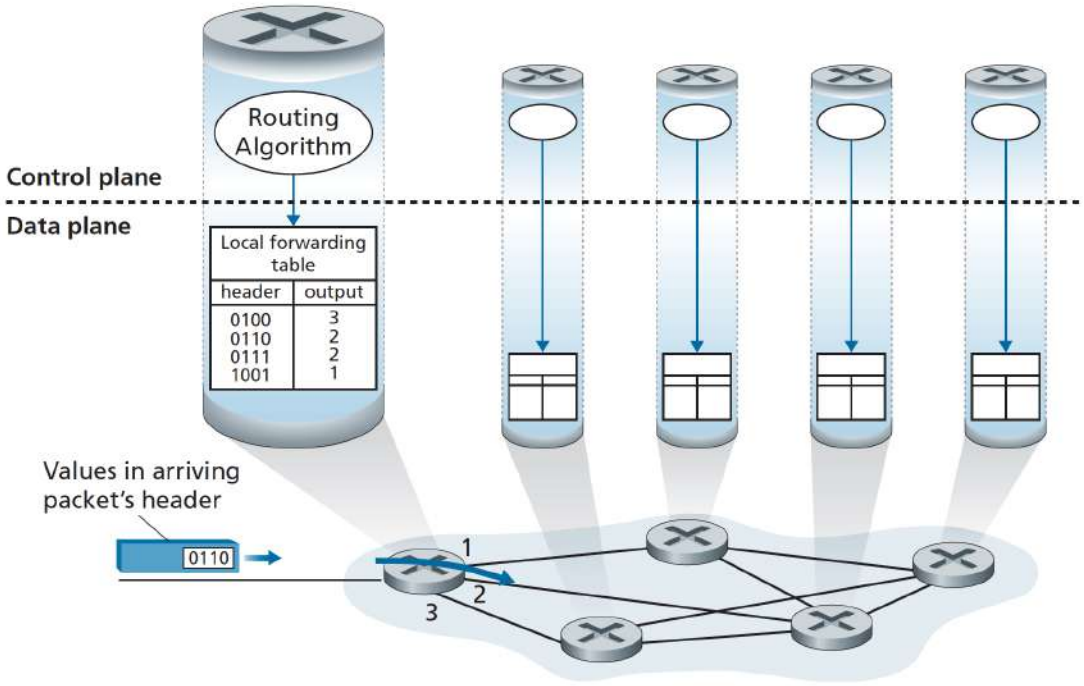

Figura 12.1: Control plane e data plane
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 307.

entre essas partes (*control plane* e *data plane*), pois seus registros são gerados pelo *control plane* e utilizados pelo *data plane*.

12.1 Protocolo IP

O IP (*Internet Protocol*) é o protocolo utilizado na camada de rede. Atualmente está largamente em uso as versões 4 e 6, as quais serão discutidas a seguir. A estrutura de dados resultante da *network layer* é o *datagram*, no qual o seu arranjo varia conforme a versão utilizada.
Citando o livro-texto desse curso:

"Nevertheless, the datagram plays a central role in the internet - every networking student and professional needs to see it, absorb it, and master it." (Computer Networking a top-down approach. 8th ed. Pearson, página 331)
Entender o *datagram* é de suma importância para a aprendizagem de redes de computadores.

12.2 IPv4 Datagram

O *datagram*, como mostrado na Figura 12.2, segue o seguinte formato:

1. Version Number: especifica a versão utilizada (no caso, 4)
2. Header length: a versão 4 do IP tem um tamanho de *header* variável, causado pelo campo *options*, o que torna necessário a manutenção dessa informação dentro do *datagram* (o campo *options* não é normalmente utilizado, portanto o tamanho típico do *header* IPv4 *datagram* é de 20 *bytes*)
3. Type of Service (TOS): exemplos são *real time, non-real-time*.
4. Datagram length: o tamanho do *datagram* é calculado pela soma dos tamanhos do header dos dados carregados (*payload*). Composto por 16 bits e medido em *bytes*, o tamanho teórico máximo é de 65535 bytes (raramente ultrapassa os 1500 bytes).
5. Identifier, flags e fragmentation offset: utilizados para fragmentar um *datagram* muito grande em pedaços pequenos que serão enviados independentemente e remontados no destino.
6. time-to-live (ttl): decrementa em 1 toda vez que o *datagram* é processado por um roteador, sendo descartado ao atingir 0.
7. Protocol: indica o protocolo da camada de transporte. contém o valor de 6 (00110) para TCP e 17 (10001) para UDP.
8. Header Checksum: Objetivando detectar erros, esse campo é computado tratando cada 2 bytes do **header** como um número e somando-os utilizando

a aritmética complementar de 1. Esse cáculo é somente feito para o **header**, evitando possíveis redundâncias com as camadas inferiores.

9. Source e destination IP addresses.

10. Options: permite uma extensão do *header*. Não foi incluido no *datagram* do IPv6 por dificultar a determinação do início do *payload* (necessitando do campo *header length*) e a variação do tempo requerido para processamento (pois alguns *datagrams* podem requisitar ou não o processamento do campo *options*).

11. Data (payload): contém o segmento da camada de transporte.

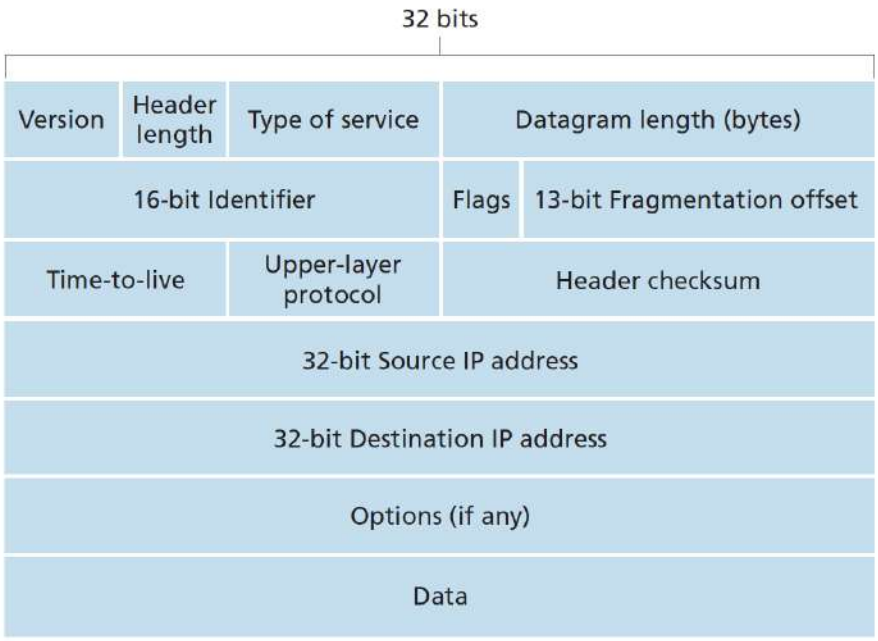

Figura 12.2: IPv4 Datagra

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 331.

12.3 Endereçamento

A fronteira entre o *host* e a conexão física é chamada de interface. Em um roteador é possível identificar múltiplas interfaces. Como cada interface está associado um endereço de IP, um roteador, portanto, pode estar associado a múltiplos endereços de IP. O endereço de IP é formado por 4 bytes (32 bits), escritos na notação *dotted-decimal*, no qual cada byte é escrito em decimal e separado por um ponto.
O endereçamento segue a estratégia conhecida como *Classless Interdomain Routing* (CIDR), no qual o endereço de IP (formado por 32 bits) é separado em prefixo (os `x` primeiros bits) e identificador de host (os restantes `32 - x` bits). O número de bits (`x`) que formam o prefixo é determinado pela máscara de rede (*mask subnet*), como mostrado a seguir:

```
Endereço de IP: a.b.c.d
Endereço de IP com máscara de rede: a.b.c.d/x
Notação: 193.32.216.9/24 (11000001 00100000 11011000 00001001)
Prefixo (Sub-rede): 193.32.216 (11000001 00100000 11011000)
Identificador de Host: 9 (00001001)
```

A máscara de sub-rede (*network mask*) distingue o endereço referente à sub-rede ao do *host*. A sub-rede pode ser entendida como uma ilha de rede isolada, com as interfaces compondo as bordas dessa rede, como mostrado na Figura 12.3.

Obter um endereço de IP A obtenção de um endereço de IP ocorre de forma automática com o protocolo DHCP (*Dynamic Host Configuration Protocol*), chamado também de protocolo *plug-and-play* ou *zeroconf* (*zero configuration*), o qual gera um endereço de IP temporário diferente toda vez que o *host* conecta-se nessa rede. O DHCP é um protocolo baseado na arquitetura *client-server*, e o seu processo é feito em quatro passos (mostrado na Figura 12.4):

1. Server Discovery: o *client* dispara *datagram* contendo um *discovery message* para o destino 255.255.255.255 (esse endereço de IP indica ao roteador que

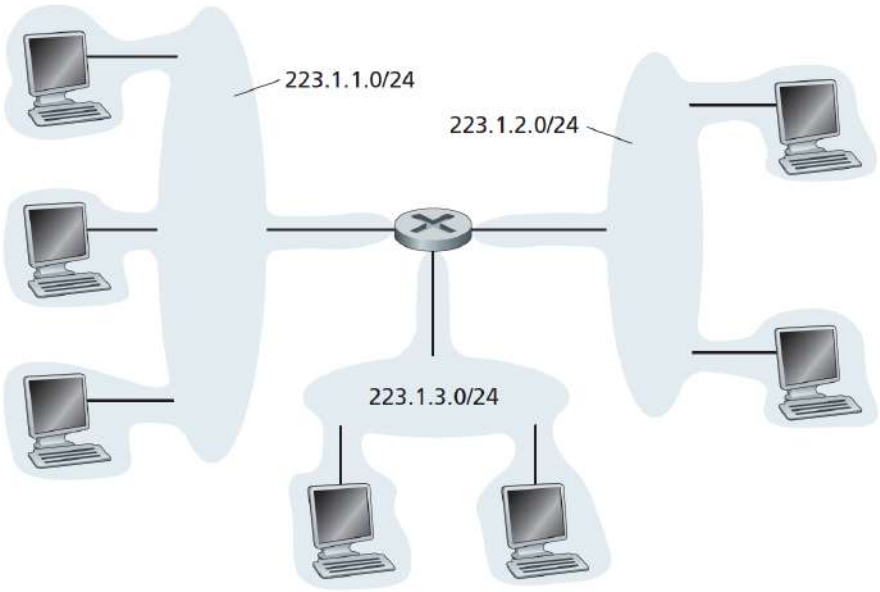

Figura 12.3: Sub-rede
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 336.

a mensagem deva ser entregue para todos as interfaces da sub-rede), com a origem em 0.0.0.0.

2. Server Offer: O Servidor DHCP, após receber a discovery message*, responde com a* offer message* para 255.255.255.255 (ou seja, disparando para todos os dispositivos da subrede). A *offer message* contém: uma proposta de *IP address*; *network mask*; e o *IP address lease time*, referente à validade do IP. É importante perceber que na mesma rede pode haver múltiplos servidores DHCP e, portanto, múltiplas *offer message* podem ser disparadas durante essa etapa.

3. Request: o *client* escolhe uma *server offer* e ecoa os seus parâmetros com a *request message*.

4. ACK: o servidor selecionado confirma a seleção do endereço enviando uma *ACK message*.

12.4 NAT

1. Como um servidor DHCP local geraria um IP único diferente de um outro servidor geograficamente distante ?
2. Os diferentes servidores DHCP devem estar sincronizados ou faixas específicas de IP devem ser pré-determinadas ?
3. Eu já encontrei redes domésticas com a mesma faixa de endereço de IP, como isso é possível ?

Essas e outras perguntas vem à tona quando é imaginado como funcionaria uma interação global de sub-redes. A solução passa pelo uso do protocolo NAT (*Network Address Translation*), mostrado na Figura 12.5.
Um roteador com o protocolo NAT ativo é visto como um dispositivo único (com o IP único) para o resto do mundo, escondendo, assim, os detalhes das configurações de uma rede doméstica para as redes externas.
É interessante notar que o roteador obtém o endereço de IP via o servidor DHCP oriundo do ISP (*Internet Service Provider*). E por sua vez, oferece um servidor DHCP para a sua sub-rede.
O segredo para o funcionamento do NAT está no *NAT translation table*.
Quando uma requisição é disparada por um *host* para um *server* fora da rede, o roteador converte o endereço de IP de origem para o seu, e gera uma nova porta. Dessa forma, a *NAT translation table* é populada, no lado WAN (rede do ISP), com endereço de IP do roteador e porta gerada, e no lado LAN (rede doméstica), endereço de IP do *host* e porta da *Thread*.
Por exemplo:

1. *Host* dispara um *datagram* com origem 10.0.0.1 e porta 3345 para o *server* 128.119.40.186 porta 80.

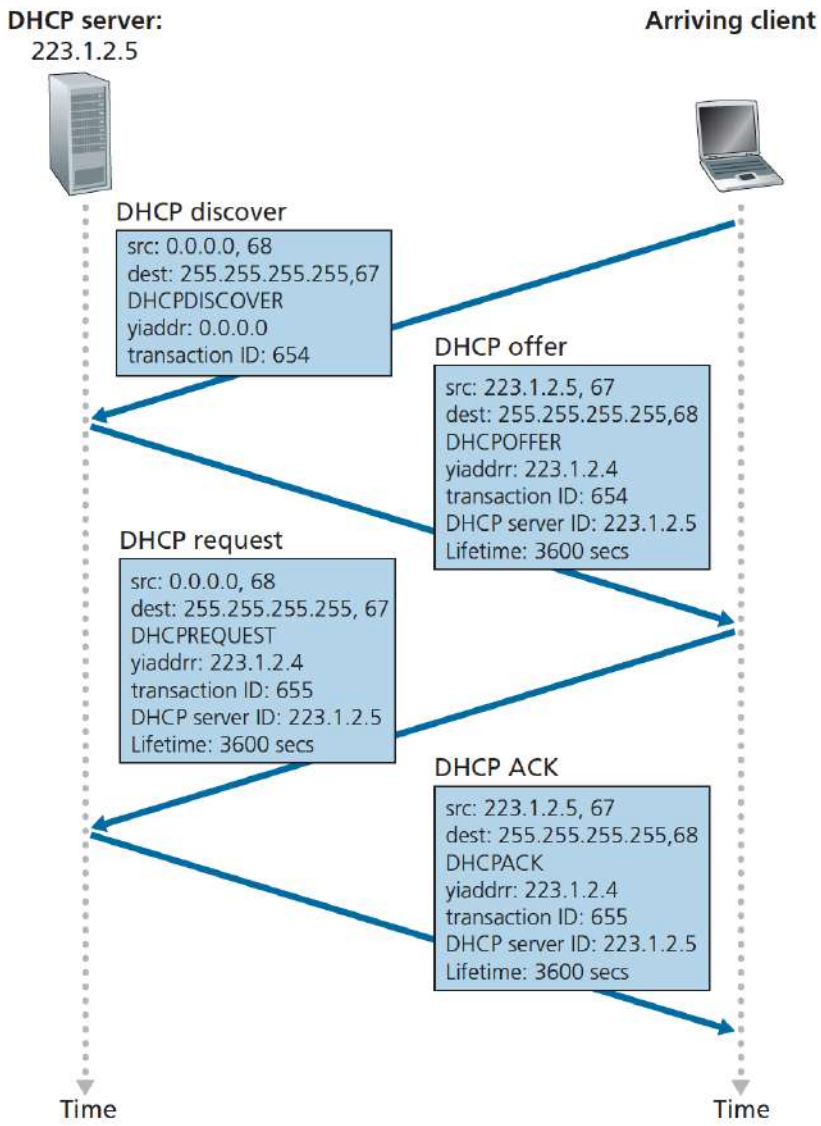

Figura 12.4: Processo DHCP

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 343.

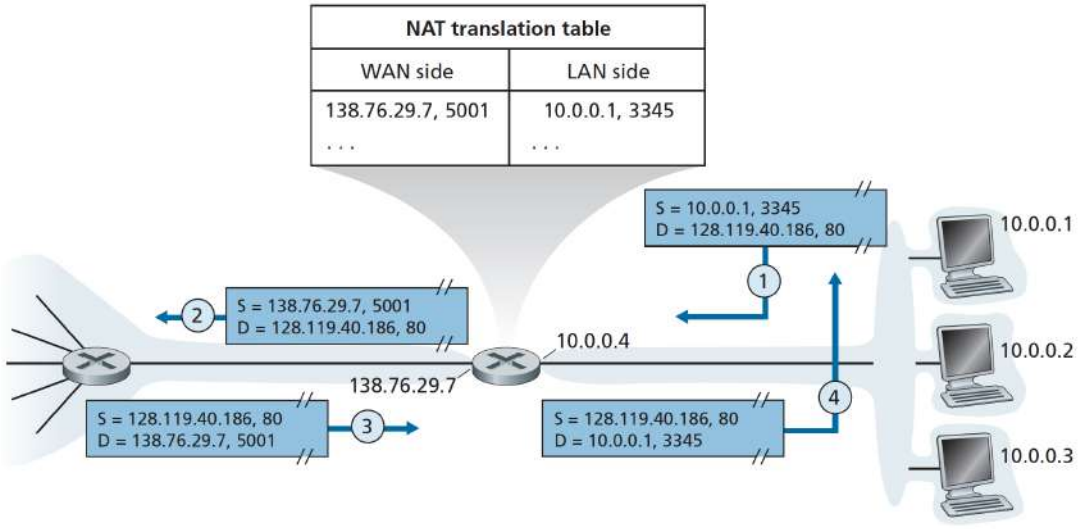

Figura 12.5: NAT

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 345.

2. O roteador gera uma nova porta e substitui os parâmetros de origem para essa porta gerada e para o seu endereço de IP (que por sua vez foi gerado pelo ISP). E registra essa conversão no *NAT translation table*.

3. O *server* recebe o *datagram* com os parâmetros de origem do roteador, e o responde.

4. O roteador converte os parâmetros de destino utilizando a tabela NAT, e, por fim, direciona o *datagram* recebido ao *host*.

Esse registro é mantido até o fim da conexão. Como o tamanho do campo porta é de 16 bits, o protocolo NAT suporta mais de 60 mil conexões com somente um único *IP address*.

Um dos problemas causados por esses protocolos (DHCP e NAT) é referente aos *Home Servers*, pois como um servidor espera por uma requisição de um

client, como esse *client* pode saber qual é o atual endereço de IP do servidor ? Como funcionaria a arquitetura P2P ? Soluções para esse problema incluem *NAT transversal tools* [RFC 5389, RFC 5128], algo que não será debatido nesse texto.

12.5 IPv6 datagram

Há uma série de mudanças introduzidas com o IPv6, mostrado na Figura 12.6:

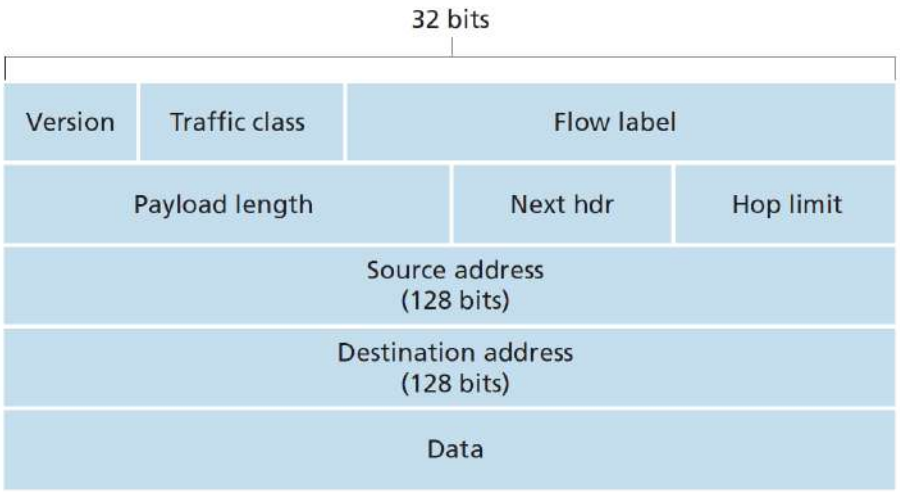

Figura 12.6: IPV6 Datagram
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 349.

1. Capacidade de endereçamento expandido: de 32 bits para 128 bits
2. *header* de tamanho fixo: o *header* foi fixado em 40 *bytes*, permitindo um processamento mais rápido pelo roteador
3. fluxo e seu rótulo: capacidade de rotular os *datagrams* de um fluxo específico, o qual cada rótulo sinaliza uma requisição específica, como *real-time service* ou *non-default quality*

Os campos do *datagram* do IPv6 são:

1. Version: versão do IP (no caso, 6).
2. Traffic class: equivalente ao TOS do IPv4, é usado para dar prioridade à algum *datagram*
3. *flow label*: campo de 20 *bits* usado para rotular um fluxo de *datagrams*
4. *Payload Length*: campo de 16 *bits* que indica o tamanho do *payload*
5. *Next Header*: identifica o protocolo para qual o *datagram* (ou o *payload*) será entregue
6. *Hop limit*: equivalente ao TTL do IPv4, diminui em 1 toda vez que o *datagram* é transmitido por um roteador e descartado quando chega em 0.
7. Endereço de origem e destino: no formato de 128 *bits*
8. *Data*: também chamado de *payload*, é o conteúdo encapsulado pelo protocolo.

Foram removidos:

1. Fragmentação e remontagem: IPv6 não permite a fragmentação e a remontagem do datagram (essa operação era feita com o objetivo de reduzir o tamanho do *datagram* grande demais para ser transmitido). Assim caso um *datagram* seja muito grande, o roteador descarta esse *datagram* e retorna uma mensagem de erro, de forma que o emissor deverá reenviar o *datagram*. A remoção dessa funcionalidade gera, como resultado, uma redução no tempo de processamento do roteador.
2. *Header checksum*: Considerado suficientemente redundante e enfatizando a velocidade de processamento, esse campo não se mostrou necessário para os desenvolvedores do IPv6.

3. Options: a remoção do *options* possibilitou a fixação do tamanho do *header* em 40 *byts*, algo que além de causar um encolhimento no tempo de processamento, também deixa explícito o início do *payload* (afinal, o *payload* sempre estará 40 bytes após o início do *datagram*).

12.6 Transição de IPv4 para IPv6

Há um problema inerente na atualização dos sistemas distribuídos: a adoção de uma nova tecnologia por todos os elementos da rede Um sistema com o IPv6 pode ser projeto para ser retro-compatível com o IPv4, mas um sistema com IPv4 não é capaz de lidar com o IPv6.
Como, então, atualizar todos os incontáveis dispositivos já integrados na rede ?

1. Transição abrupta: substituir todos os elementos de uma vez só, marcando um dia x para inutilizar os dispositivos compatíveis com IPv4.

2. Transição suave: substituir os elementos de forma gradual.

A abordagem da transição suave foi o caminho escolhido. Para tal, fora adotado a prática do *tunneling* (algo que torna os dispositivos IPv6 compatível com o IPv4). O *tunnel* encapsula o *datagram* do IPv6 integralmente, tornando-o o *payload* do IPv4, resultando em um *datagram* com o *header* do IPv4 (como mostrado na Figura 12.7).

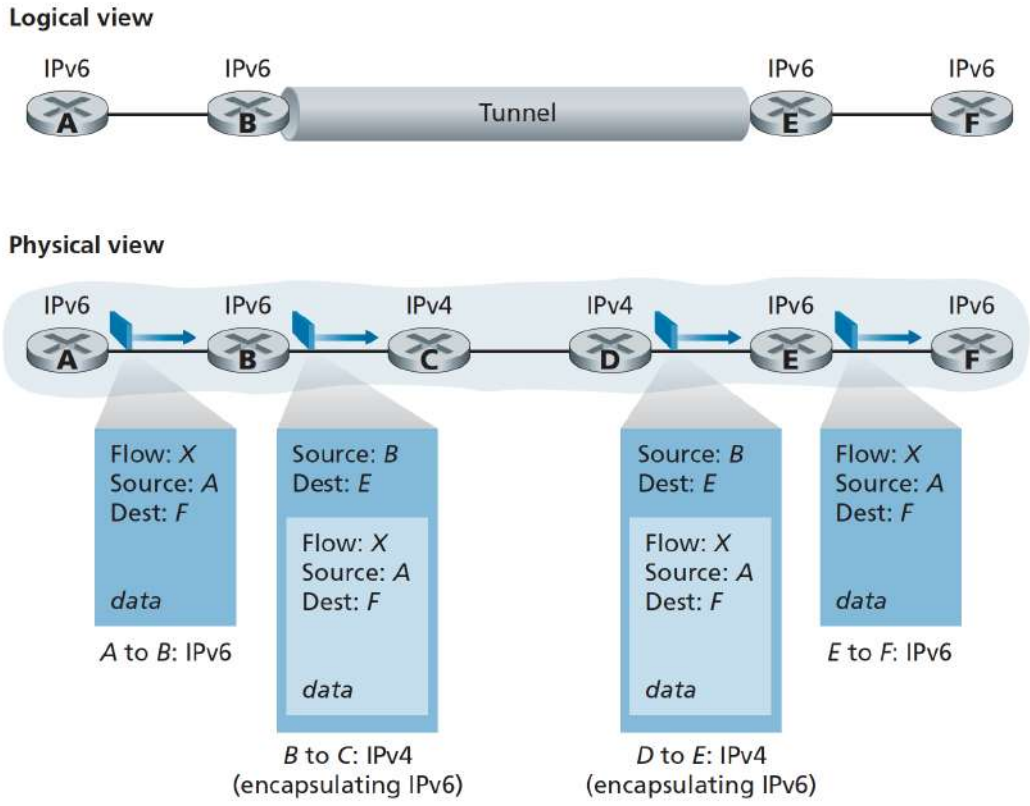

Figura 12.7: Tunneling

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 352.

13 Link Layer

Após a *Network Layer* determinar qual o caminho de comunicação (chamado de *link* ou enlace) o *datagram* deve percorrer, como o *WiFi* ou o *Ethernet*, entra em cena o *Link Layer* (camada de enlace), responsável por encapsular o *datagram* e transmitir o resultado (o *frame*) através do *link*. Os dispositivos que executam a camada de enlace são chamados de nós (*nodes*).

Fundamentalmente, os *links* podem ser classificados em dois canais de comunicação. O primeiro refere-se aquele no qual os nós compartilham do mesmo caminho de transmissão (*single broadcast link*), como os *wireless LANs*. O segundo tipo é de ponto-a-ponto, no qual somente dois nós são conectados em cada *link*.

O *link layer* está presente tanto em *software* (com, por exemplo, a montagem das informações de endereçamento e o controle de interrupções) como em *hardware* (com, por exemplo, *link access* e *framing*) e é implementado em um *chip* chamado de *Network Interface Controller* (NIC).

13.1 Serviços

A camada de enlace provê os seguintes serviços:

1. *Framing*: constituição do *frame* a partir do encapsulamento do *datagram*.

2. *Link access*: o controle de acesso é algo fundamental para a realização da transmissão, sendo esse realizado pelo *Medium Access Protocol* (MAC protocol). Nos *links* ponto-a-ponto, o protocolo somente verifica se o *link* está disponível. Em *single broadcast link*, ocorre o problema de multiplos acessos simultâneos, sendo da responsabilidade do protocolo MAC especificar as regras para a transmissão dos *frames*

3. *Reliable delivery*: protocolo que objetiva garantir a transmissão de cada *datagram*.

4. *Error detection and corretion*: devido à possíveis erros introduzidos pela atenuação do sinal ou ruídos eletromagnéticos, vários protocolos fornecem mecanismos de detecção e correção de erros.

13.2 Endereçamento

O *Link Layer* utiliza um sistema de endereçamento similar ao *Network Layer* com o *IP Address*.
Na fabricação de um NIC, lhe é designado um endereço único de 6 bytes chamados de *MAC Address* (*LAN Address*, ou *Physical Address*), que se complementa ao *IP Address* de forma similar à relação entre o CPF de um brasileiro com seu endereço residencial, pois o endereço residencial é alterado conforme ocorre uma mudança (assim como o *IP Address* é alterado após o dispositivo mudar de rede), mas o seu CPF não é modificado (assim como o *MAC Address*). Portanto, é designado (pelo fabricante) ao NIC um *MAC Address*, e (pela rede) um *IP Address*.

13.2.1 ARP

O *Link Layer* necessita, para o envio de seus *frames*, o endereço MAC do dispositivo receptor. Mas como obter esse endereço ? Esse endereço pode ser obtido através do *Address Resolution Protocol* (ARP), no qual armazena em uma tabela (ARP *table*)

as equivalências entre *IP Address* e *MAC Address*. A ARP *table* está localizada nos NICs de cada *host* e *router* da sub-rede.
O ARP é similar ao DNS, com a diferença de estar limitado à sua sub-rede (diferente do DNS que está disponível para toda a Internet).

Funcionamento do ARP

Suponha que o *host* 222.222.222.220 quer enviar um *datagram* para 222.222.222.222, mas não tenha em sua ARP *table* o registro contendo a equivalência entre o IP e o MAC *Address*. Para tal, o emissor deve utilizar o protocolo ARP:

1. O emissor deve inquerir (*query*) todos os *hosts* e *routers* da sub-rede para determinar a equivalência. Esse inquérito ocorre com a montagem de um ARP *packet* (uma estrutura especial que contém o endereço IP e MAC do emissor e do receptor) destinado ao *MAC Broadcast Address* (um endereço específico, FF-FF-FF-FF-FF-FF, no qual sinaliza que a transmissão deve ser feito para todos os *hosts* da sub-rede)

2. O NIC encapsula o ARP *packet* em um *link layer frame* e transmite para a sub-rede (equivalente a uma pessoa gritar em uma sala quem tem um CPF XXX.XXX.XXX-XX)

3. O *host* no qual o seu módulo ARP contém o registro inquerido envia de volta (diretamente ao inquisidor) um *response* ARP *packet* com o mapeamento desejado.

4. O *host* inquisidor, por fim, pode atualizar sua ARP *table* e enviar os seus dados para o endereço MAC correspondente à inquisição.

É importante perceber que: 1. O *query* ARP *message* é enviado para todos em *broadcast*, enquanto que sua resposta não (ela é enviada em um *frame* padrão). 2. ARP é *plug-and-play*, montando sua tabela automaticamente. 3. O ARP é um protocolo localizado entre o *Network Layer* e o *Link Layer*, por conter

parâmetros oriundos de ambas as camadas (como o *IP Address* e o MAC *Address*, respectivamente) 4. O ARP opera quando um *host* quer enviar um *datagram* para outro *host* na mesma sub-rede.

Como um emissor pode enviar um *datagram* para fora de sua rede se o MAC *Address* de destino é necessário e o protocolo ARP não fornece o endereço de dispositivos fora da sub-rede ?

A resposta: Não é necessário que o emissor saiba do MAC *Address* do *host* de destino ! Basta o emissor popular o campo MAC *Address* de destino com o MAC *Address* do roteador da sua rede! (Que pode ser obtido pelo ARP)

O roteador da rede do emissor receberá o *datagram* e avaliará para qual NIC esse *datagram* deve ser direcionado (isso é feito consultando o *fowarding table*). Uma vez no NIC (da sub-rede 2), o módulo ARP é acionado para obter o MAC *Address* do destinatário. Por fim, o *datagram* é encapsulado pelo *Link Layer* e transmitido para o receptor da mensagem.

13.3 Ethernet

O protocolo *Ethernet* foi inventado nos anos 1970 por Bob Metcalfe e David Boggs. Inicialmente utilizava um barramento coaxial (uma *broadcast* LAN) para interconectar os nós. Os padrões utilizados eram os 10BASE-2 e 10BASE-5 (10 refere-se à velocidade, em Mbps, BASE ao *baseband* Ethernet, significando que a mídia física só carrega o tráfico de dados Ethernet, e a parte final refere-se a mídia física em si, como o cabo coaxial), os quais especificavam 10Mbps Ethernet em cabos coaxiais limitados a 500 metros. Distâncias maiores poderiam ser obtidas com o uso dos *repeaters*, dispositivos da camada física que repetem o sinal de entrada na sua saída.

Em 1990, os barramentos coaxiais foram substituídos pelo cabo de cobre de par trançado conectado em um Hub, um dispositivo da camada física (*1-layer*) que retransmite os bits de entrada para todos os nós conectados a ele, em uma arquitetura estrela (com todos os *hosts* conectados ao dispositivo central, o Hub), mantendo-se, portanto, uma *broadcast* LAN. Um problema presente nos *Hubs*

ocorre quando múltiplos *frames* são transmitidos simultaneamente, algo que gera uma colisão, e os nós que criaram os *frames* devem retransmiti-los.
A solução para as colisões veio nos anos 2000, quando o *Hub* fora substituído pelo *Switch*, dispositivo de segunda camada (*2-layer*, camada de enlace) que armazenam e transmitem (*store-and-foward*) os dados utilizando o *MAC Address* para direcioná-los aos seus respectivos *links*, tornando, assim, dispensável o uso do protocolo MAC (protocolo para controle de colisão).
O *Ethernet* tornou-se o protocolo dominante em redes LAN (*Local Area Network*) por:

1. Ser o primeiro a implantar largamente o *high-speed* LAN
2. Era mais simples e barato do que seus concorrentes
3. *Data rates* compatíveis ou maiores do que os seus concorrentes
4. Por consequência de sua popularidade, dispositivos *Ethernet* são mais baratos.

É importante perceber que, apesar da intensa mudança de padrão sofrida desde os anos 1970, o Ethernet ainda utiliza a mesma estrutura de *frame*, algo que será debatido posteriormente.

13.3.1 Ethernet Frame

O *frame* do Ethernet, como mostrado na Figura 13.1, é composto por:

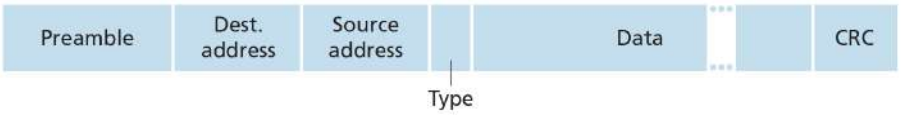

Figura 13.1: Estrutura do frame Ethernet
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 486.

1. *Data Field* (*payload*): é o local no qual é carregado o *datagram* (resultado da camada superior). Tem o tamanho máximo (*Maximum Transmission Unit*, MTU) de 1500 *bytes* e mínimo de 46 *bytes*. Caso o *datagram* seja maior, o *host* deve fragmentá-lo. Caso seja menor, o campo *Data Field* é preenchido (*stuffed*) até o mínimo (o campo *length* presente no *header* do *datagram* indicará o seu tamanho correto).

2. *Destination Address*: Esse campo contém o MAC *Address* de destino (endereço de 6 bytes).

3. *Source Adress*: Esse campo contém o MAC *Address* de origem da mensagem (endereço de 6 bytes).

4. *Type Field* (2 *bytes*): Indica o protocolo utilizado no *datagram* (como IP e ARP).

5. *Cycic redundancy check* (CRC) (4 *bytes*): permite o receptor identificar erros no *frame*.

6. *Preamble* (8 bytes): o *frame* inicia com o *preamble*. É utilizado para "acordar" o NIC receptor e sincronizar os seus relógios.

O Ethernet é *conectionless* ou seja, não requer de *handshaking* anterior ao envio de uma mensagem. Após enviado uma mensagem, não há respostas confirmando sua chegada (*acknoledgments*). Assim, seu serviço é dito como não confiável, algo que torna o Ethernet simples e barato.

13.4 Switch

O *Switch* é transparente para os dispositivos da sub-rede (os dispositivos não sabem da presença do *switch*), e seu princípio de funcionamento (*match plus action*) é baseado no *Filtering and Fowarding*, no qual o *Filtering* determina se um *frame* deve ser descartado ou transmitido, e o *Fowarding* define qual interface o *frame* deve ser transmitido. O *Filtering and Fowarding* utiliza uma tabela chamada

de *switch table*, a qual armazena registros com os campos *address*, referente ao MAC *Address*, *Interface*, indicando qual interface o endereço MAC está anexado, e *Time*, que marca o tempo no qual o registro fora armazenado.
Existem 3 casos para o *Filtering and Fowarding*:

1. Não há registro para o MAC *Address* de destino: dispara para todos os dispositivos (*broadcast*).
2. O MAC *Address* de destino é o mesmo da interface no qual o *frame* fora recebido: discarta o *frame* (*filtering*).
3. Há um registro referente ao MAC *Address* de destino com a interface sendo diferente a da origem do *frame*: põe o *frame* no *buffer* que corresponde à interface de destino (*fowarding*).

É interessante perceber que o *switch* é *self learning* (aprende sozinho). Essa capacidade é obtida da seguinte forma:

1. A *switch table* inicializa vazia.
2. Para cada *frame* recebido, o *switch* armazena na sua tabela: o MAC *Address* da origem; a interface no qual o *frame* foi recebido; o tempo atual. Dessa maneira, eventualmente, o *switch* preencherá completamente sua tabela.
3. Um registro é deletado caso não seja recebido *frames* com o seu MAC *Address* de origem após um certo período de tempo (*aging time*).

O *switch* apresenta algumas vantagens sobre o *hub*:

1. Eliminação de colisões: não há perda no comprimento de banda por consequência das colisões.
2. *Links* heterogêneos: os diferentes *links* podem operar com velocidades e mídias diferentes.
3. Gerenciamento: além de melhorar a segurança, os *switches* coletam estatísticas de rede que podem ser usados para melhorá-la.

Em comparação com os roteadores, os *switches* apresentam:
Vantagens:

1. *Plug-and-play.*
2. Altas taxas de *filtering and fowarding*

Desvantagens:

1. Topologia de rede restrita
2. Suscetível ao *broadcast storm*: caso um *host* dispare incontáveis *frames* em *broadcast*, a operação do *switch* pode saturar a rede, podendo colapsá-la.

Roteadores:
Vantagens: 1. *Datagrams not cycle*: mesmo com caminhos redundantes, os *datagrams* não se mantém vivos vagando pela rede (proteção vinda de campos como TTL). 2. Topologia de rede não restrita 3. *Firewall* contra *broadcast storm* (não suscetível ao *broadcast storm*). 4. Provê um isolamento mais robusto
Desvantagens:

1. Não é *plug and play*
2. Maior tempo de processamento: pois os roteadores devem processar até a terceira camada.

14 Roteadores.

14.1 Analogia

Imagine uma via de veículos com pedágio. Suas entradas direcionam todos os carros de diferentes origens para um mesmo pedágio. Durante o pedágio, os carros aguardam pacientemente em uma fila a execução de uma série de operações, como o pagamento da taxa de uso da via. Após essas operações, os carros atravessam a via de transporte encaminhando-se até as suas respectivas saídas.

O *data plane* de um roteador funciona de forma análoga a essa via imaginada. Os *datagrams*, análogos aos carros, após entrarem no roteador (pelos *inputs*), são enfileirados e ficam no aguardo de serem processados. Após uma série de operações, os mesmos são direcionados para as suas respectivas saídas (*outputs*).

Assim, o roteador é um caso específico da abstração mais genérica *match plus action* (corresponder e agir), o qual é performado também por outros dispositivos, como os *switches*, e não apenas pelos roteadores.

A Figura 14.1 apresenta a arquitetura de um roteador, separada em *control plane* e *data plane*, que implementam o *routing* e o *fowarding, respectivamente*.

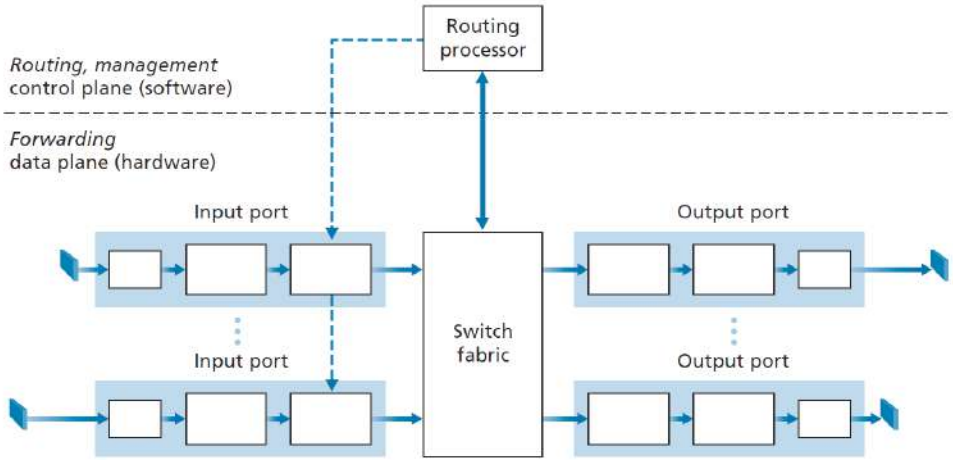

Figura 14.1: Arquitetura de um roteador
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 311.

14.2 Descrição

De forma mais descritiva, o roteador é composto por:

1. *input ports*: armazena os *datagrams* recém chegados e performa funções da camada física e de enlace, como o *lookup*, a determinação da porta de saída (para tal, é necessário consultar a *fowarding table*).

2. *Switching fabric*: conecta os *input ports* com os *Output ports*.

3. *Output ports*: armazena os *datagrams* recebidos pela *Switching fabric* e transmite-os para fora do roteador (performando funções da camada física e de enlace essenciais).

4. *Routing processor*: performa funções do *control plane*, como computar as rotas e manter a *fowarding table* (em roteadores SDN, as operações mencionadas são feitas remotamente).

É importante mencionar que os pontos 1, 2 e 3 são quase sempre implementados em *hardware*.

14.2.1 Tipos de forwarding

O *output link* pode ser determinada de duas formas:

1. *Destination-based fowarding*: no qual a saída é baseada no destino.
2. *Generalized forwarding*: em que o *output link* é definido com base em N fatores, como a origem dos dados.

Voltando para a analogia, o *Destination-based fowarding* seria o atendente do pedágio direcionar o carro a uma saída específica baseado no destino que o motorista almeja. No caso do *generalized forwarding*, fatores como o tipo do veículo pode impactar na orientação dada pelo atendente.

14.3 Questões a cerca do forwarding.

E se: 1. O atendente somente for capaz de atender 1 carro por minuto, mas chegarem 2 carros por minuto ? 2. Todos os carros que entrarem forem orientados para a mesma saída ? 3. Tiver mais carro entrando do que saindo ? 4. For necessário tornar prioritário alguns veículos (como ônibus) e bloquear a entrada de outros (como caminhões acima de um certo peso) ?
Para as perguntas 1, 2 e 3, fica claro que gerará tráfego na entrada, na saída e na via, respectivamente. Por fim, a resposta da pergunta 4 passa pela criação de regras prévias a partir da definição das políticas de filtro.
O mesmo pode ocorrer em um roteador, com a chegada de múltiplos *datagrams* em uma mesma *input port* sendo maior que sua capacidade de processá-los podendo gerar filas, atrasos e perda de dados (com o mesmo ocorrendo na saída). A quantidade limitada de *datagrams* suportada pelo *switch fabric* pode não suportar a alta demanda de fluxo de dados, causando o bloqueio de novos entrantes e, consequentemente, filas e atrasos.

Essas questões serão debatidas posteriormente em mais detalhes.

14.4 Input Port

A Figura 14.2 mostra as execuções ocorridas na *input port*. Inicia-se com a ação de funções relativas a *physical layer* (em *Line Termination*), seguida de funções da *link layer* (em *Data link processing*), por fim o *lookup, fowarding* e *queuing* (em *lookup, fowarding, queuing*).

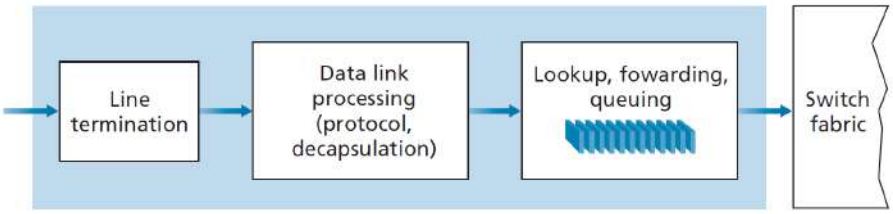

Figura 14.2: Execuções na Input Port
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 314.

A função central da porta de entrada é o *lookup*, no qual o roteador procura (*look up*) na *forwarding table* qual porta o *datagram* recém chegado deve ser direcionado. Apesar da proeminente importância do *lookup*, outras ações também devem ser tomadas, como a checagem do *version number*, *checksum* e *Time to Live*
Como pode-se supor, em uma arquitetura de banco de dados centralizado, os registros da *forwarding table* estariam disponíveis em um único local do roteador, com a operação de *look up* devendo fazer uma requisição de registro à esse banco de dados, algo que gera um possível gargalo, afinal, caso o módulo do banco de dados não conseguir suprir a demanda das requisições, deverão ocorrer filas e atrasos.
Para evitar esse gargalo, cada linha da tabela é copiada para cada um dos *input ports* presentes no roteador, de forma que o acesso à tabela de transmissão

(*forwarding table*) seja feita localmente (no *input port*), sem a necessidade de requisições.

No *look up*, o roteador identifica a saída (*link interface*) utilizando a regra de maior correspondência (*longest prefix matching rule*) de prefixo do IP *Address* de destino do *datagram*. Um exemplo dessas correspondências podem ser vistas na Tabela 01. (O prefixo é formado por x primeiros dígitos do IP *Address* de destino referencia o endereço da sub-rede.)

Prefix	Link Interface
11001000 00010111 00010	0
11001000 00010111 00011000	1
11001000 00010111 00011	2
Prefix	Link Interface

Tabela 14.1: Exemplo de forwarding table

Por exemplo, o endereço de IP:

```
11001000 00010111 00011000 10101010
```

Tem o prefixo correspondendo ao *link interface* 1 e 2, com o *link* 1 sendo a maior correspondência e, portanto, sendo os dados direcionados ao mesmo.

14.5 Switching

O *switching fabric* tem a função de conectar as *input ports* com as *output ports*. Há diferentes formas de implementá-lo, mas três delas, mostradas na Figura 14.3, destacam-se:

1. *Switching via memory*: o *input port* sinaliza, a partir de uma interrupção, para o processador do roteador a chegada de novos *datagrams*. Após a interrupções, os dados recém chegados são copiados para a memória, processados (determinando-se a interface apropriada), e por fim copiados

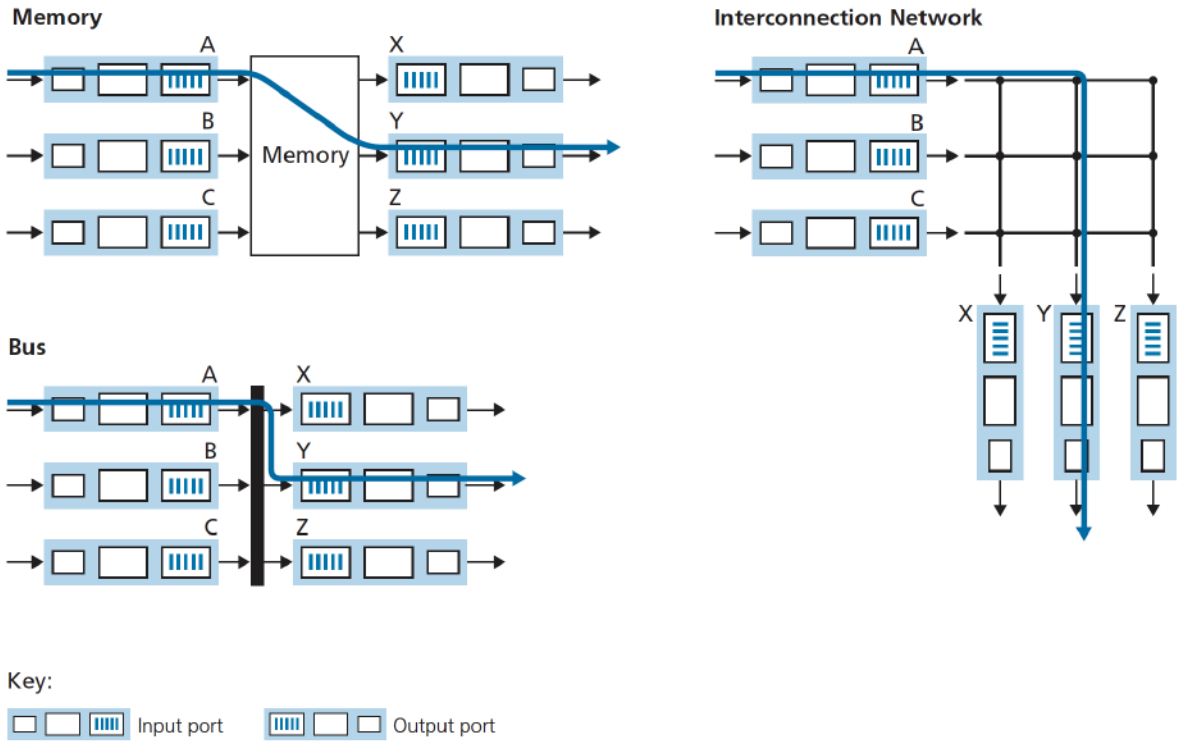

Figura 14.3: Tipos de Switching

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 317.

para o *buffer* de saída. Isso é usualmente feito com processos com memória compartilhada.Como desvantagem pode-se citar, primeiro, a impossibilidade de transmissão de mais de um *datagram* ao mesmo tempo, pois somente uma operação de leitura e escrita na memória pode ser feita por vez. Segundo, com a largura de banda sendo B *datagrams* por segundo para memória (B *datagrams* podem ser escritos ou lidos em 1 segundo), significa dizer que a taxa de transferência está limitada em B/2 (pois será necessário fazer 1 operação de escrita e uma de leitura).

2. *Switching via bus*: a *input port* envia os *datagrams* diretamente para a sua respectiva *output port* através de um barramento compartilhado, sem a intervenção do processador do roteador. Isso é usualmente feito agregando-se um *header* com um rótulo informando sua respectiva interface. Ao ser transmitido, todas as interfaces recebem esses dados, porém somente aquela indicada pelo rótulo irá manter o mesmo, retirando o seu rótulo, processando-o, e transmitindo-o para fora do roteador. O primeiro fato inconveniente dessa arquitetura é que somente 1 pacote de dados podem pecorrer o barramento. Isso significa que se chegarem múltiplos *datagrams* em diferentes *input ports*, todos menos 1 devem esperar o barramento tornar-se disponível. O segundo problema é que a velocidade do roteador estará limitada pela velocidade do barramento barramento. Esse tipo de arquitetura é indicado para LAN.

3. *Switching via an interconnection network*: também chamado de cruzamento de barras, esse tipo de switch, como mostrado na Figura 02, pode alternar cada cruzamento de barra (ou nó) entre aberto e fechado de forma independente. Dessa maneira, multiplos *datagrams* podem atravessar em paralelo o *switch fabric* ao mesmo (ou seja, *non-blocking* para uma *output port* específica). Por exemplo, para emitir um dado do *input* A para o *output* Y, basta fechar o cruzamento dos mesmos. Esse fechamento não impedirá que os dados do *input* B alcance a saída X, porém o *output* Y estará somente disponível para o *input* A (estando bloqueado para os outros *inputs*).

14.6 Output port

A Figura 14.4 mostra as 3 principais execuções do *output port*. Inicia-se com o enfileiramento (*Queuing*) dos *datagrams* recebidos. Em seguida, são performados ações necessárias relativas as camadas físicas e enlace (*Data link processing*). Por fim, os dados são enviados para fora do roteador (*Line termination*).

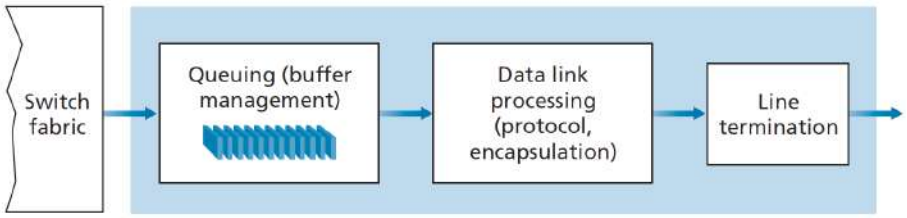

Figura 14.4: Output Port
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 319.

14.7 Filas

As filas podem se formar na entrada e na saída do roteador.

As filas na entrada podem ser formadas pela diferença de velocidade entre o *input* e o *switching fabric* (como citado anteriormente, de forma análoga, na questão 1 do "E se" no tópico" Questões a a cerca do forwarding"). Se um *input* for capaz de lidar com uma taxa de *datagrams* mais alta do que o *switching fabric*, os mesmos ficarão acumulados na entrada até serem executados pelo *switching fabric*.

Um outro evento que têm como consequência a geração de filas é o chamado *head-of-the-line blocking* (HOL *blocking*). A Figura 14.5 mostra um exemplo de como o HOL *blocking* pode ocorrer. Os *datagrams* azuis-escuros estão destinados às saídas superiores, enquanto que os azuis-claros estão destinados às saídas centrais. Assim, um azul-escuro oriundo do *input* superior pode bloquear a passagem do *datagram* azul-escuro oriundo do *input* inferior, travando a fila e impedindo que o *datagram* azul-claro seja processado paralelamente, apesar do caminho até sua saída estar livre.

Na saída, as filas podem se formar quando múltiplos *datagrams* dos *inputs* são direcionados para o mesmo *output*, como mostrado na Figura 14.6. Esse evento pode preencher o *buffer* de saída, ocasionando a "derrubada" de novos *datagrams*, política chamada de *drop-tail*, ou a remoção de um já enfileirado, para assim

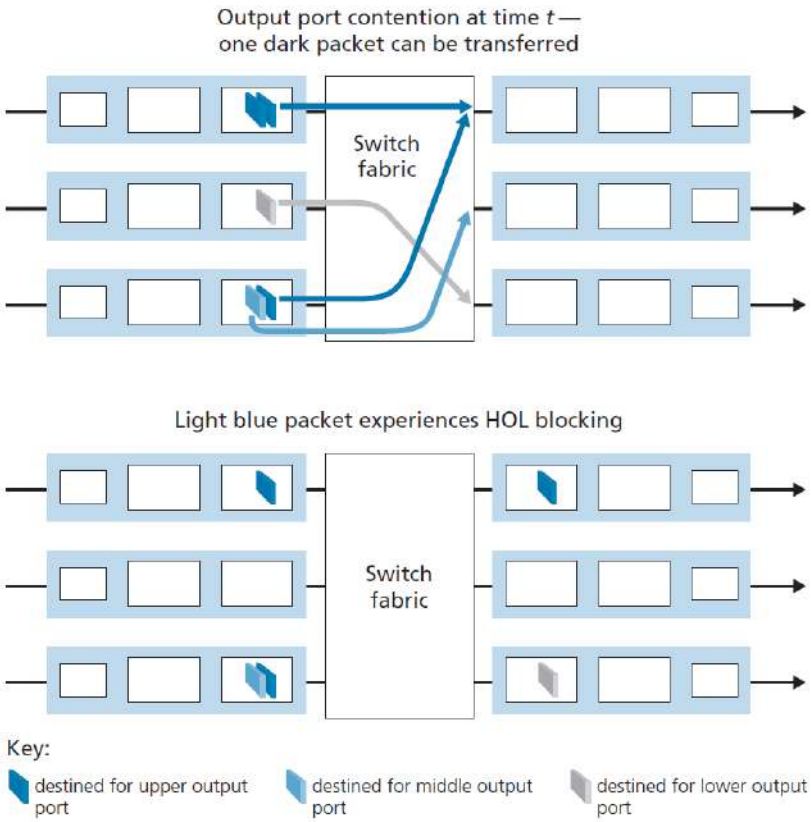

Figura 14.5: Hol Blocking

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 321.

criar espaço para os dados recém chegados. Em alguns casos, pode ser vantajoso remover um pacote de dados antes que fila fique cheia, de forma a enviar um sinal de congestionamento para o emissor. Os algoritmos responsáveis por isso são chamados de *Active Queue Management* (AQM), ou gerenciador de filas ativo, com o *Random Early Detection* (RED) sendo um algoritmo dessa classe amplamente implementado.

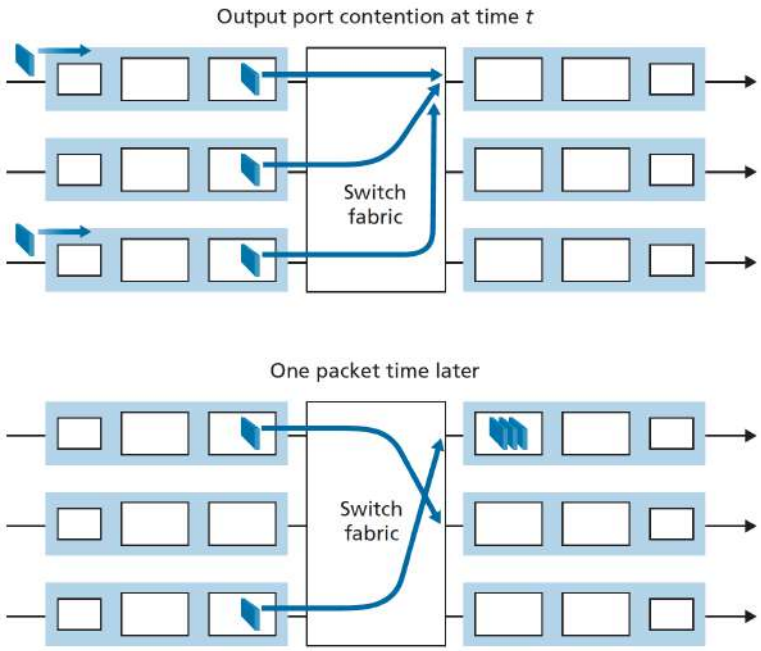

Figura 14.6: Fila na saída
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 322.

O tamanho do espaço destinado para as filas sofrem de uma dicotomia. Enquanto que *buffers* pequenos podem não apresentar espaço suficiente para lidar com picos de demanda, algo que aumenta o número de dados perdidos, filas grandes podem representar um longo tempo de espera, aumentando o atraso na entrega dos *datagrams*.
Para equilibrar esses diferentes pontos, o tamanho do *buffer* (`B`) fora relacionado com o *round-trip time* (RTT) médio (período entre a emissão dos dados e a recepção do seu ACK) e a capacidade do link (`C`).

```
B = RTT . C
```

```
B = 2.5G bits, para RTT = 250 ms, e C = 10 Gbps
```

Atualmente, também relaciona-se o número de fluxos independentes de TCP (`N`):

```
B = RTT . C / sqrt{N}
```

14.8 Prioridades dos dados

Na discussão dos *buffers* fora deixado implícito a política *First-in-First-Out* (FIFO), ou primeiro a chegar, primeiro a sair (modelo mostrado na Figura 14.7), mas outras regras também podem ser utilizadas, como a classificação dos dados em diferentes filas a partir de sua prioridade (modelo mostrado na Figura 14.8).

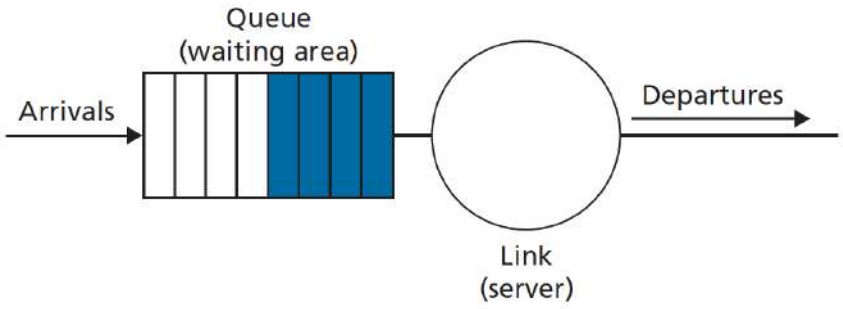

Figura 14.7: Modelo FIFO
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 325.

Uma forma generalizada de classificação e priorização dos dados que tem sido bastante implementada é o chamada de *Weighted Fair Queuing* (WFQ), na qual as filas de maior peso são tratadas primeiro, seguindo para as de menor peso, até reiniciar o ciclo, como mostrado na Figura 14.9.

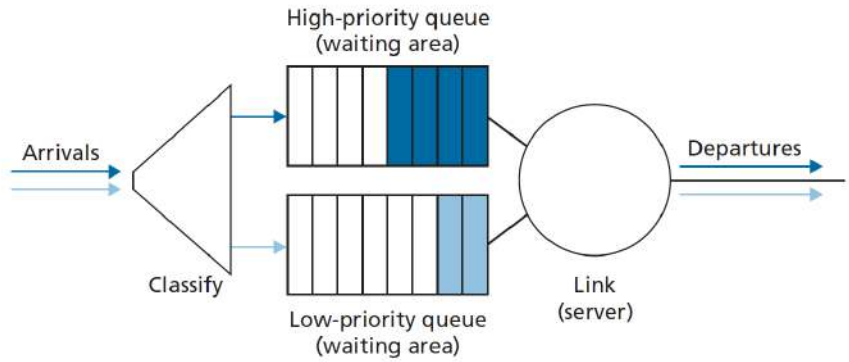

Figura 14.8: Modelo classificação e priorização

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 326.

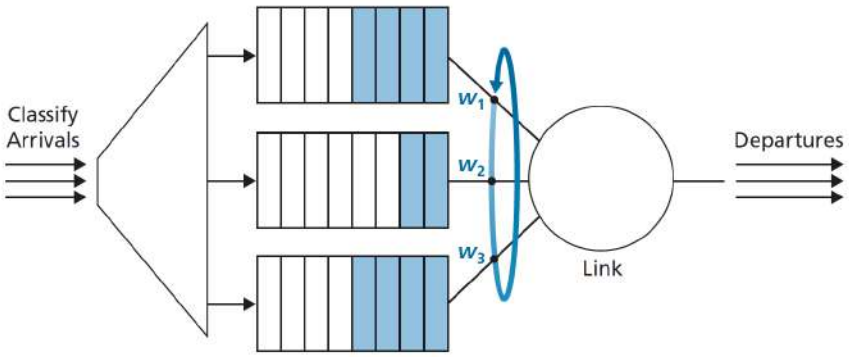

Figura 14.9: Weighted Fair Queuing

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 326.

14.8.1 Neutralidade das redes

Como qualquer regra pode ser imposta para a classificação dos dados, os ISP's podem não ser neutros no oferecimento de seus serviços, algo que vem resultando leis que regulamentam o que os ISP's podem ou não fazer.

Em geral, a defesa da neutralidade das redes pode ser resumido em 3 pontos:

1. Conteúdos não devem ser bloqueados (*No Blocking*).

2. Tráfico de Internet não deve ser penalizado (*No Throttling*).

3. Não deve haver priorização paga (*Paid Prioritization*).

15 Generalized Forwarding

O *Generalized Forwarding* é a forma genérica do paradigma *match plus action*, no qual utiliza os *headers* associados aos protocolos de diferentes camadas para determinar o que deve ser feito com os dados recebidos. Para a sua implementação, é utilizado o padrão OpenFlow, o qual baseia-se no SDN (*Software Defined Network*), que aplica um controlador remoto para o cálculo das regras da rede.

A base do *Generalized Forwarding* está na *Flow Table*, tabela no qual está contido as regras do *match plus action*. Cada regra têm:

1. Um conjunto de *headers* à serem verificados.
2. Um conjunto de contadores.
3. Um conjunto de ações à serem tomadas.

Perceba que essa tabela é uma forma limitada de programação. Atualmente, vem ganhando força uma alternativa mais rica de possibilidades de implementação (com variáveis e funções), a linguagem *Programming Protocol-independent Packet Processors* (P4).

Dentro do padrão OpenFlow, os *headers* podem ser oriundos das camadas de enlace, rede e transporte, como mostrado na Figura 15.1. É importante

perceber que nem todos os *headers* podem ser escolhidos para a ação de correspondência (*match*) (assim fora implementado como uma forma de equilibrar a funcionalidade e complexidade. É melhor fazer algo simples e bem, do que muita coisa e ruim).

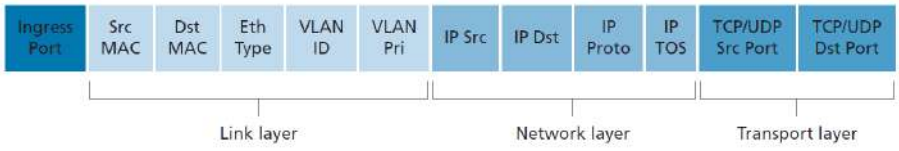

Figura 15.1: OpenFlow Headers
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 356.

As ações a serem tomadas são:

1. *Forwarding*: a transmissão dos dados.
2. *Dropping*: o "deixar de lado".
3. *Modify-field*: a modificação de algum *header*.

15.1 Camada de rede: Control Plane

Na aula anterior, foi debatido a importância da *Fowarding Table* para o *Data Plane*. Os dados registrados nessa tabela são computados pela *Control Plane*, a qual tem como objetivo controlar a rota global que os *datagrams* precisarão percorrer para sair de uma ponta à outra da rede (end-to-end). A *Control Plane* também configura e gerencia os componentes e serviços fornecidos pela camada de rede.
Uma rede pode ser vista como um grafo, no qual os vértices (ou nós) são os roteadores e as arestas são a conexão entre dois vértices, como pode ser visto na Figura 15.2. Dessa forma, os algoritmos de roteamento determinam o melhor caminho que um dado pode percorrer para sair de um vértice da rede até outro vértice.

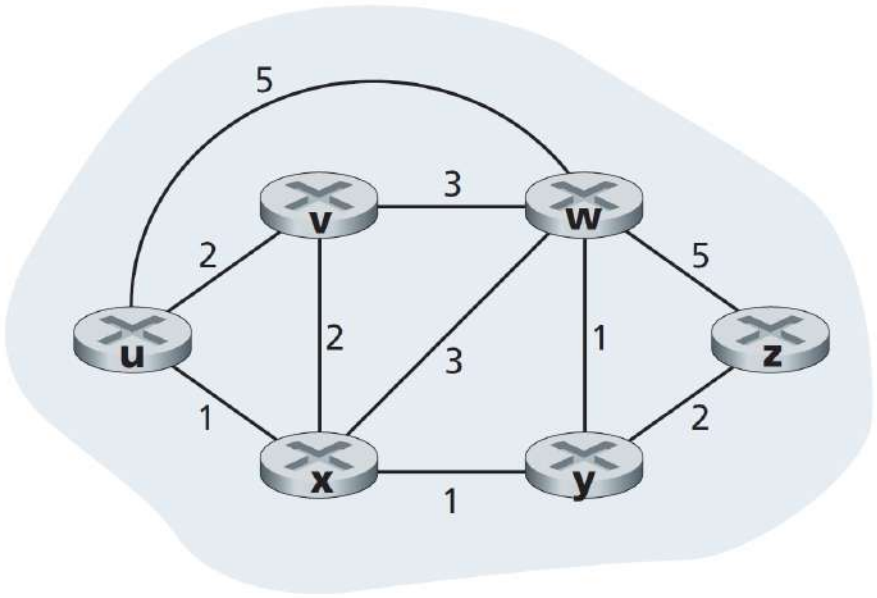

Figura 15.2: Grafo com pesos
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 381.

As características de cada conexão (velocidade, tarifas financeiras, etc.) são contabilizadas (a partir de métricas estabelecidas pela instituição dona da rede), resultando no custo (ou peso) agregado à conexão. Como cada aresta apresenta características diferentes, serão agregados pesos diferentes ao uso de cada uma. Assim, os algoritmos de roteamento, como o OSPF e o BGP (conhecido como a "cola" da Internet), tem o objetivo de encontrar um caminho entre dois nós que apresente o menor custo de ser percorrido (custo total do caminho).

É importante perceber que o caminho de menor custo (*least-cost path*) é diferente do caminho mais curto (*shortest path*), pois o primeiro é caracterizado por aquele que apresenta o menor somatório dos pesos das conexões inseridas no mesmo, enquanto que o segundo é determinado pela menor quantidade de nós que deve ser percorrido.

Ambos os algoritmos citados (OSPF e BGP) utilizam a abordagem *per-router control* (mostrado na Figura 15.3), em que o algoritmo de roteamento é processado dentro de cada roteador, sendo necessário interações entre *routers* para a determinação das rotas. Outra possível abordagem é a *Logically centralized control* (mostrado na Figura 15.4), em que há uma centralização computação em um servidor e distribuição dos parâmetros determinados para os nós da rede, como adotado pelo SDN (*Software Defined Network*).

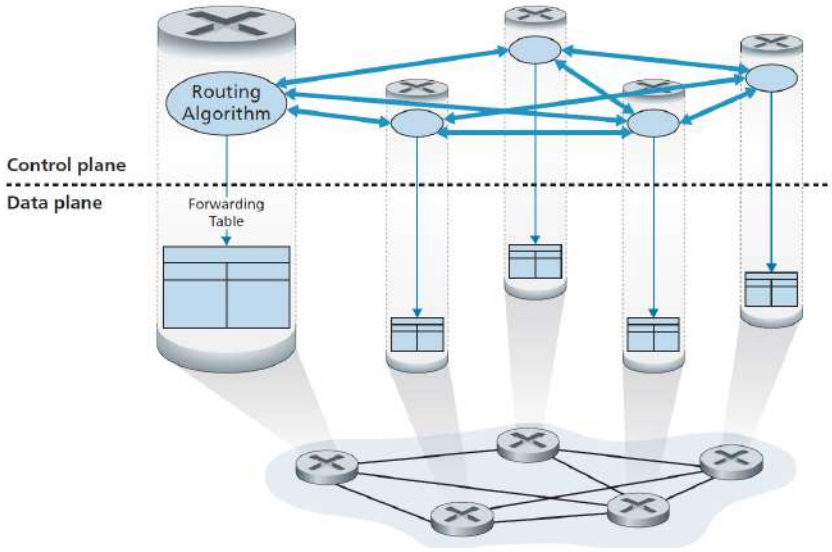

Figura 15.3: Per Router Control
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 378.

15.2 Classificação dos algoritmos

De forma abrangente, podemos classificar os algoritmos de roteamento em:

1. *Centralized routing algorithm*: os algoritmos dessa categoria, comumente referidos como *link-state* (LS) *algorithms*, computam o caminho a partir

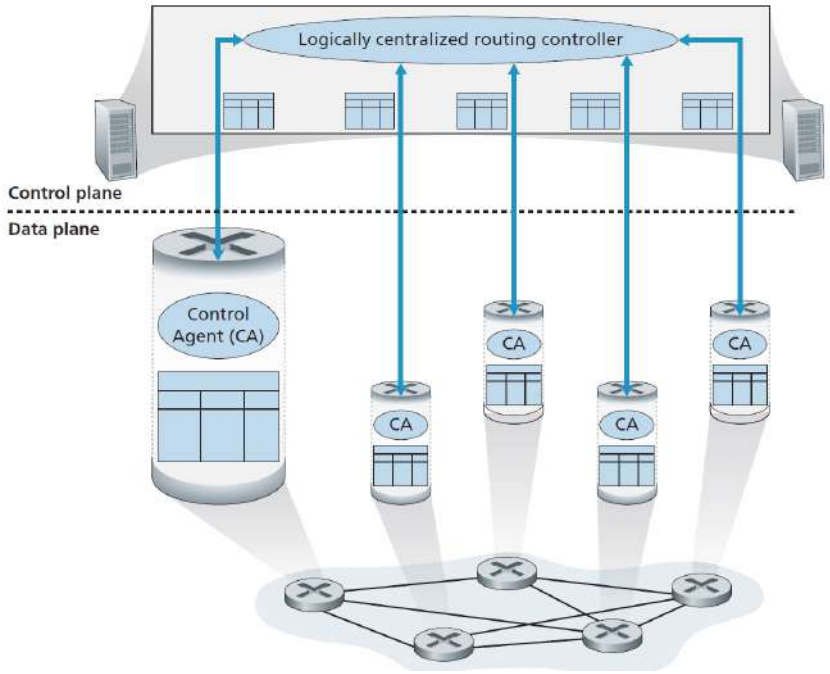

Figura 15.4: Logically Centralized Controller
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 379.

de um conhecimento completo a cerca da conectividade e custos de cada conexão da rede (tendo um conhecimento global da rede). Um exemplo é o algoritmo de Dijkstra.

2. *Decentralized routing algorithm*: a determinação do caminho de menor custo é feita de forma iterativa e distribuída em cada roteador. Como, inicialmente, cada roteador só têm conhecimento dos custos de seus próprios *links*, o cálculo do menor caminho necessitará da troca de informações entre os outros roteadores da rede. Isso ocorre de forma iterativa e gradual. Um exemplo é o *Distance-Vector Routing Algorithm* (DV), um algoritmo iterativo, assíncrono e distribuído, com cada nó mantendo

um vetor de estimativas de custos de todos os outros nós da rede (com a atualização ocorrendo conforme mudanças na rede acontecem). Podemos citar alguns protocolos que utilizam o DV: Internet's RIP, BGP, ISO IDRP, Novel IPX e o original ARPAnet.

Uma segunda forma de classificação é:

1. *Static Routing Algorithms*: os roteadores mudam muito pouco no decorrer do tempo (frequentemente como resultado de uma intervenção humana).

2. *Dynamic routing algorithms*: as rotas mudam conforme a ocorre mudança na topologia da rede ou carga de tráfego (apesar dessa categoria apresentar maior responsividade em mudanças na rede, também estão mais suscetíveis a problemas como *loops* e oscilações).

Uma terceira forma:
Load-Sensitive algorithm: os custos dos *links* variam dinamicamente conforme o nível de congestionamento. *Load-Insensitive algotihm*: os custos dos *links* não refletem explicitamente o nível de congestionamento.

15.3 LS vs DV

Alguns pontos são importantes para a comparação entre os *link-state algorithms* e o *Distance-Vector Routing Algorithm*:

1. *Message complexity*: o LS requer que cada nó saiba os custos de cada conexão da rede, com uma mudança em um custo devendo ser enviada para todos os nós. Já no DV, o envio do novo custo somente ocorre quando o mesmo impacta no caminho de menor custo dos nós anexados ao *link* relativo à alteração (DV melhor que LS).

2. *Speed of convergence*: LS converge mais rápido do que o DV (LS melhor que DV).

3. *Robustness*: no DV, um caminho de menor custo calculado incorretamente por um nó será publicado para todos os nós da rede, diferentemente do LS, no qual os caminhos são calculados em cada nó, provendo assim um certo nível de robustez (LS melhor que DV).

15.4 Intra-Autonomous Sistems Routing: OSPF

Um sistema autônomo (Autonomous Sistems, AS) consiste de um conjunto de roteadores que estão sob um mesmo controle administrativo. Isso torna possível a escalabilidade e a autonomia administrativa do sistema. Os algoritmos de roteamento que rodam em um AS são chamados de *intra-autonomous system routing protocol*. Um exemplo de algoritmo é o *Open Shortest Path First*, um protocolo LS no qual as especificações do protocolo de roteamento está disponível publicamente (a parte *Open* do nome). No OSPF, cada roteador monta um mapa topológico completo de todo o AS, e então executa o algoritmo de Dijkstra para a determinação da árvore de caminhos de menor custo para todas as sub-redes, com os custos das conexões sendo configurados pelo administrador da rede.
Observe que is roteadores que conectam-se com outros de diferentes ASs são chamados de *gateway routers* (eles estão nos limites da AS). Já aqueles que estão no interior da AS, são chamados de *internal routers*.
Alguns pontos importantes de mencionar:

1. *Security*: as trocas de informações entre os roteadores OSPF podem ser autenticadas por meio da geração de um *hash* MD5 (utilizando-se uma chave secreta, que está contida no roteador e não é compartilhada) por ambos (emissor e receptor da mensagem). Em seguida o *hash* do emissor é comparado com o do receptor. O emissor pode ser considerado autêntico caso os *hashs* gerados sejam iguais. Caso contrário, a mensagem dese ser ignorada.

2. *Multiple same-cost path*: O tráfego de dados poderá ser distribuído em todas as rotas que contenham o custo mínimo.

3. *Support for hierarchy within a single AS*: o sistema pode ser configurado hierarquicamente em áreas.

15.5 Inter-Autonomous Sistems Routing: BGP

Por necessitar a coordenação de múltiplos ASs, a comunicação entre ASs deve ocorrer utilizando-se o mesmo protocolo. O protocolo usado é o *Border Gateway Protocol* (BGP), e é conhecido também por ser a "cola" da Internet (por unir os diferentes ASs).
O roteamento pelo BGP ocorre entre sub-redes e não para um endereço específico da rede. Dessa forma, a *forwarding table* do roteador toma a forma de `(x, I)`, onde `x` é o prefixo e o `I` é a interface do roteador.
O BGP provê para os roteadores meios para:

1. Obter o prefixo de AS vizinhos: com a publicação da existência de cada rede para o resto da Internet.

2. Determinar a melhor rota para cada prefixo: no qual a melhor rota é baseada nas políticas determinadas pelo administrador da rede e na acessibilidade da informação.

As conexões BGP entram em duas categorias (graficamente representado na Figura 15.5):

1. iBGP (*internal* BGP): conexão BGP internas aos ASs

2. eBGP (*external* BGP): conexão externa aos ASs (entre ASs)

A publicação no BGP ocorre de forma bem direta. A Figura 15.6 mostra a adição de uma sub-rede (`x`) em uma rede com 3 ASs. Primeiro o AS3 envia uma BGP *message* (`AS3 x`) para o AS2 dizendo que a sub-rede `x` existe e é acessível através dele. Em seguida, o AS2 avisa para o AS1 a existência de `x` e que o mesmo é acessível através do caminho AS2-AS3 (`AS2 AS3 x`).

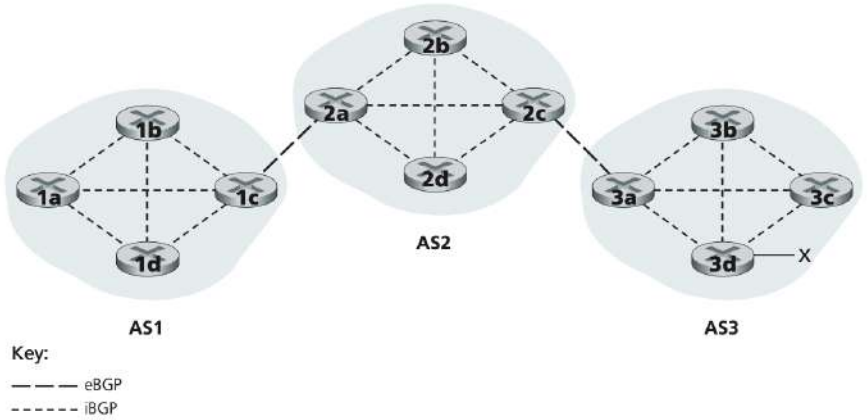

Figura 15.5: eBGP e iBGP

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 402.

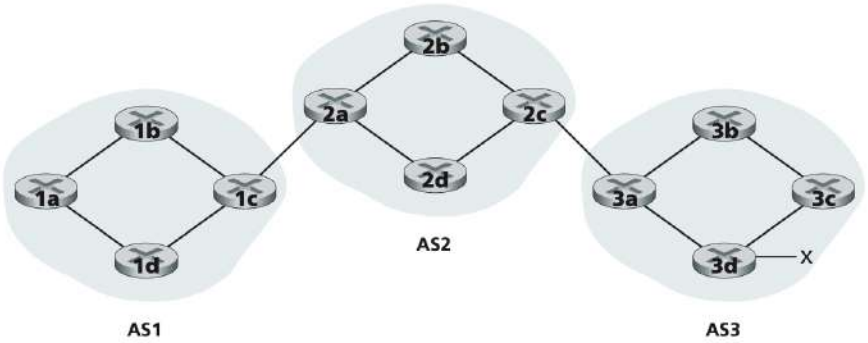

Figura 15.6: ASs com adição de x

Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 401.

O BGP *message* é composto pelo prefixo e outros múltiplos atributos, como o *AS-PATH*, que explicita a lista de ASs na qual a mensagem publicação de existência da

nova rede precorreu (como mostrado no exemplo anterior), e *NEXT-HOP*, que é o endereço da interface do roteador que inicia a o *AS-PATH*.

15.6 Hot Potato

Os ASs operam com a bordagem *Hot Potato*, o qual os roteadores objetivam transmitir os dados para fora do AS o mais rápido possível. Para tal, o roteador com os dados disparará para o endereço do *NEXT-HOP* que tiver o menor custo de conexão, sem se preocupar com o resto do trajeto desses dados. Assim, apesar de localmente eficiente, a rota global escolhida pode não ser a mais rápida. A Figura 15.7 mostra duas possibilidades de *NEXT-HOP*. A rota escolhida será aquela que apresentar o menor custo de conexão relacionado ao *NEXT-HOP*. A Figura 15.8 mostra os passos para a adição de um destino externo ao AS na *forwarding table*

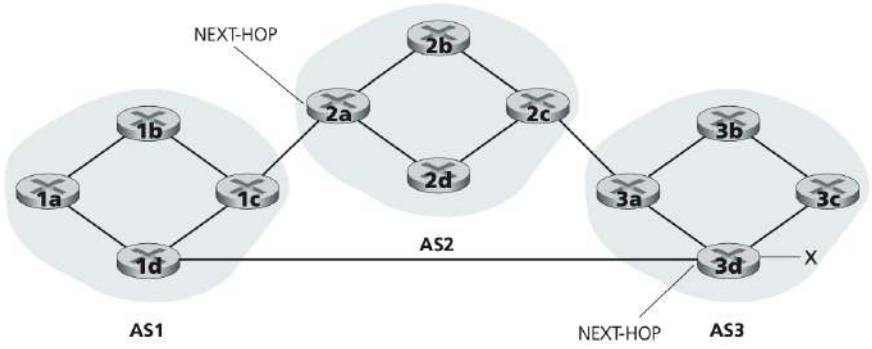

Figura 15.7: Duas possibilidades de NEXT-HOP
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 403.

15.7 Seleção da rota.

Na prática, a rota selecionada utiliza outros parâmetros além do *Hot Potato*:

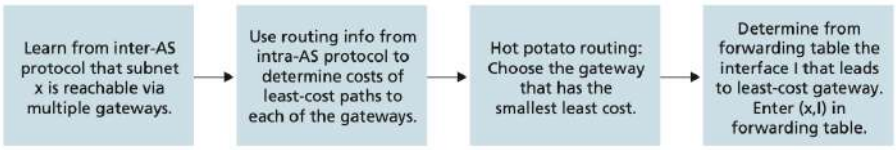

Figura 15.8: Passos para a adição de um destino externo na forwarding table
Imagem retirada de: Computer Networking a top-down approach. 8th ed. Pearson, página 404.

1. Política de decisão da preferência local de transmissão: essa política é escolhida pelo administrador da rede

2. Para os restantes, será escolhido o que tiver o menor *AS-PATH*.

3. Para os restantes, o *Hot Potato* entra em ação, escolhendo a rota com o menor custo de transmissão para o *NEXT-HOP*.

4. Para os restantes, é utilizado os identificadores BGP para a seleção da rota.

15.8 Política de preferência local e Protocolos Intra vs Inter

O AS é dito como sistema autônomo pois o mesmo apresenta uma independência administrativa. Isso é algo que pode gerar alguns problemas de confiança, com decisões como não permitir que dados originário de um AS passe através de outro AS específico. E, de forma similar, um AS pode ter interesse em querer controlar o tráfego de dados entre outros ASs. Um reflexo no cenário atual é a desconfiança dos americanos sob empresas chinesas, que, comumentemente, recebe interferência do governo chinês, um inimigo declarado dos EUA (assim, deve-se ser evitado que dados críticos do governo americanos sejam transmitidos através de empresas chinesas).

Outra questão gira em torno do problema de uma sub-rede de um cliente fazer parte da rota dos dados de outros clientes. Um cliente deve ser origem ou destino das mensagens e não um meio.
A política é uma questão tão forte na comunicação inter-ASs, que ela é um dos maiores motivos para que os protocolos inter-ASs sejam diferentes dos intra-ASs, tanto que até a qualidade das rotas (externas) utilizadas é uma preocupação secundária. Por fim, a questão da escala não é tão forte no intra-ASs quanto é em inter-ASs. Assim, os 3 motivos para que os protocolos sejam diferentes são:

1. Política: intra-ASs é secundário, inter-ASs é primário.
2. Escala: intra-ASs é secundário, inter-ASs é primário.
3. Performance: intra-ASs é primário, inter-ASs é secundário (impactado pelas políticas).