Trabalho prático 2 - Socket UDP

Luan Gabriel, Victor Nacle

Maio 2019

Conteúdo

1 Introdução

Após o sucesso estrondoso da internet diversas aplicações começaram a ser desenvolvidas com o fim de se utilizar a rede para se comunicarem. Com o surgimento de tais aplicações, padrões de comunicação foram necessários serem estabelecidos, para garantir maior escalabilidade e compatibilidade entre os sistemas gerados.

Tais padrões, comumente chamados de *protocolos*, foram aplicados a arquitetura de camadas **TCP/IP** da internet, de forma que proporcionassem as normas necessárias para o desenvolvimento de cada camada da arquitetura. Dentre os protocolos criados, o *UDP* da camada de transporte é amplamente utilizado devido a facilidade e simplicidade de implementação.

A biblioteca *sockets* do Unix (Linux) tem como objetivo abstrair a camada de rede para que aplicações possam se comunicar sem ter que se preocupar com detalhes dos protocolos das camadas inferiores à aplicação. Tal biblioteca fui utilizada para se construir uma aplicação na linguagem C que opera no modo cliente-servidor sobre o protocolo *UDP*, de forma que fosse possível transmitir um arquivo, construir um mecanismo de transferência confiável e se verificar o desempenho da transmissão através de um algorítimo *stop and wait*.

2 Metodologia

Nessa seção será abordado o modelo de construção da aplicação, bem como o uso de funções da biblioteca *socket*, além do método de análise do desempenho dos programas. Para a construção para o par de programas cliente e servidor, foi utilizado o algorítimo *stop and wait* do livro didático [1], que pode ser visualizado abaixo:

[1] Redes de computadores e a Internet: uma abordagem top-down. James F. Kurose, Keith W. Ross

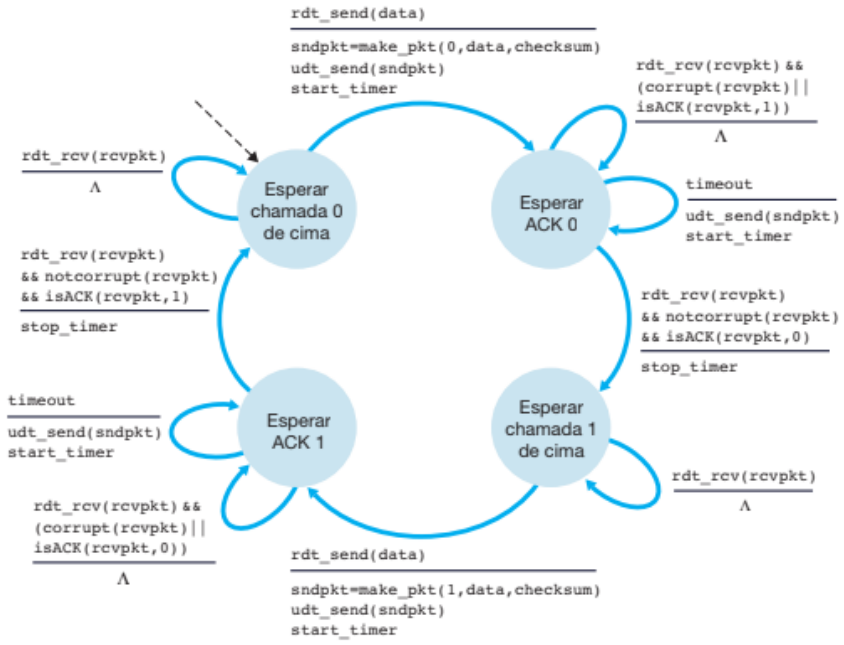

Figura 1: FSM do servidor

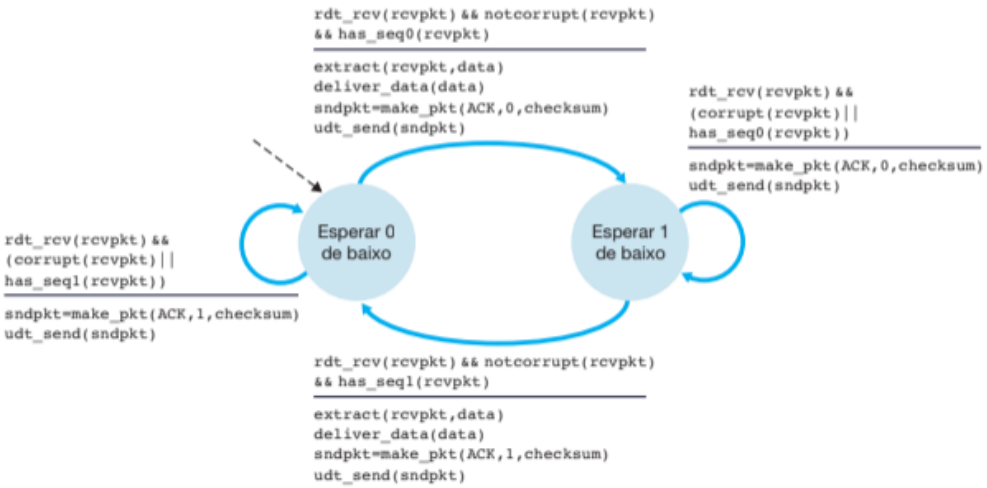

Figura 2: FSM do cliente

2.1 Estruturação e sintaxe

A aplicação construída é basicamente composta por um par de programas em C que funcionam como um servidor e um cliente, além de dois arquivos em C do tipo .c e .h que simulam as funções do *UDP* (fornecidas pelo professor).

2.1.1 Programa cliente

O programa cliente, inicialmente recebe uma 4-Tupla de valores por argumentos e os processa (IP do servidor, porta de entrada no servidor, nome do arquivo, tamanho do *buffer* interno). Em seguida se aloca um espaço de memória para o *buffer* do tamanho de entrada. Para se garantir a transferência confiável foi criada um estrutura chamada "pacote"que possui um cabeçalho formado pelo valor de ACK e ID do pacote além de possuir a parte de dados.

```
typedef struct pacote{
    char numSeq;
    char ack;
    char* dados;
}pkg;
```

Uma estrutura do tipo *sockaddr_in* também é criada para se armazenar o endereço do servidor com base nos dados passados por argumento através da função *tp_build_addr* contida no arquivo *tp_socketc*.

Após esses valores serem processados, grava-se o tempo atual através da função:

```
gettimeofday(&timeInit, NULL);
```

Em que timeInit é uma estrutura do tipo *timeval*. Nesse momento um *socket* é criado através da função:

```
clientSocket = tp_socket(0);
```

Em seguida, o nome do arquivo que se deseja ler é colocado no *buffer* e enviado para *socket* através da função abaixo. Uma vez que foi considerado que não há perda de dados no envio do nome do arquivo, a estrutura "pacote"com acks e Ids, não precisou ser utilizada.:

```
numDadosSocket = tp_sendto(clientSocket, buffer, strlen(
    nomeArquivo), &servidorAddr);
```

Nesse ponto, o programa abre o arquivo onde será gravado os dados (saida.txt). Conseguinte o programa espera receber o primeiro pacote do cliente. Para tal utiliza a função:

```
numDadosSocket = tp_recvfrom(clientSocket, buffer,
   tamBuffer , &servidorAddr);
```

Uma vez que a partir desse momento pode-se haver perda de dados, o valor recebido é deserializado para uma estrutura pacote. Isso é feito através da função *serialize* que transforma a estrutura pacote em um vetor de caracteres ou vice-versa. É importante ressaltar que como o "pacote"tem que ser colocado no *buffer* o tamanho da parte de dados da estrutura tem que ter tamanho *tamBuffer - 2*, já que os caracteres de Ack e Id também devem ser colocados no *buffer*.

O programa cliente em seguida, analisa os valores de Ack e Id recebidos, e caso estejam de acordo com o esperado, envia os dados recebidos para o arquivo de saída e envia para o servidor um pacote vazio (não há dados a serem enviados) com um Ack para o pacote recebido e um número de identificação, pela função abaixo:

```
tp_sendto(clientSocket, buffer, 2, &servidorAddr);
```

Para cada bloco de bytes lidos, o número de bytes recebidos é incrementado. Caso os valores recebidos não estejam de acordo, é reenviado o último pacote de confirmação com os valores de Ack e Id correspondentes ao pacote mais recente. Por fim, caso os valores de Ack e Id recebidos sejam o caractere 'x', o loop atual de recebimento de dados é finalizado, uma vez que significa que o programa servidor terminou de enviar todos os dados. Foi considerada que a mensagem de fechamento de conexão nunca é perdida, para menor complexidade do trabalho.

Após toda a gravação no arquivo, o mesmo e o *socket* são fechados, recebe-se o tempo atual e calcula-se a diferença de tempo entre eles:

```
gettimeofday(&timeEnd, NULL);
timersub(&timeEnd, &timeInit, &timeDelta);
```

Por fim, tendo-se o numero de bytes transmitidos e o tempo gasto, calcula-se a taxa de envio, e imprime-se esses valores na tela.

2.1.2 programa servidor

O programa servidor, funciona independente do programa cliente, de forma que ao iniciar, recebe por parâmetros a porta do servidor e o tamanho

do *buffer* interno. Em seguida, ele aloca um espaço de memória do tamanho do *buffer* para o mesmo e para o vetor de tamanho do arquivo. Uma estrutura "pacote"também é criada no servidor para os pacotes enviados e recebidos com o intuito de se criar uma transmissão confiável de dados, assim como no programa cliente. Uma estrutura do tipo *sockaddr_in* também é criada para armazenar o endereço do cliente.

Após esses valores serem processados, grava-se o tempo atual através da função:

```
1 gettimeofday(&timeInit, NULL);
```

Em que timeInit é uma estrutura do tipo *timeval*. Nesse momento um *socket* é criado através da função:

```
1 serverSocket = tp_socket((unsigned short) portaServidor)
    ;
```

Em seguida, o nome do arquivo que se deseja ler é recebido pelo *buffer* no *socket* através da função abaixo. Uma vez que foi considerado que não há perda de dados na recepção do nome do arquivo, a estrutura "pacote"com acks e Ids, não precisou ser utilizada.:

```
1 numDadosSocket = tp_recvfrom(serverSocket, buffer,
    tamBuffer, &clienteAddr);
```

Nesse momento a estrutura *clienteAddr* fica inicializada com os valores de endereço e porta do programa cliente. Quando o nome do arquivo tiver sido recebido, ocorre um *resize* no vetor com o nome do arquivo, para evitar alocação excessiva. O arquivo com o nome recebido é aberto para ocorrer o envio de dados.

Uma vez que a partir desse momento pode-se haver perda de dados, é lido do arquivo um bloco de dados de tamanho *tamBuffer - 2* para se criar um pacote de envio. Esse pacote é serializado pela função *serialize* já comentada, com os valores de Ack e Id correspondentes, colocados no *buffer* e enviados ao programa cliente pela função:

```
1 numDadosSocket = tp_sendto(serverSocket, buffer,
    numDadosArquivo + 2, &clienteAddr);
```

É criado uma temporização de 1 segundo através do *socket* (uma vez que é bem mais simples) através da função:

```
1 timer.tv_sec = 1;
2 timer.tv_usec = 0;
```

```
3 setsockopt(serverSocket, SOL_SOCKET, SO_RCVTIMEO,(struct
      timeval *)&timer, sizeof(struct timeval));
```

Essa temporização é necessária uma vez que caso o cliente não responda ao envio do pacote em um tempo de 1 segundo é reenviado o último pacote com os valores de Ack e Id correspondentes ao pacote mais recente, de acordo com a lógica abaixo:

```
1 if (errno == EAGAIN){
2             errno = 0;
3             printf("Pacote perdido! Reenviando ultimo
                 pacote ...\n");
4             serialize(buffer, &pkgEnv, 0);
5             tp_sendto(serverSocket, buffer,
                 numDadosArquivo + 2, &clienteAddr);
6         }
```

Conseguinte, caso se receba um pacote antes da temporização de um segundo, se analisa os valores de Ack e Id recebidos, e caso estejam de acordo com o esperado, é lido um novo bloco de dados do arquivo que é enviado ao programa cliente pela função abaixo. Para cada bloco enviado, o numero de bytes enviados total é contabilizado:

```
1 tp_sendto(serverSocket, buffer, numDadosArquivo + 2, &
    clienteAddr);
```

Caso não existam mais dados para serem lidos, é enviado uma mensagem de fechamento de conexão, como foi descrito pelo programa cliente:

```
1 tp_sendto(serverSocket, "xx", 2, &clienteAddr);
```

Quando todo o arquivo já foi enviado, o loop atual é fechado, o mesmo e o *socket* são fechados, recebe-se o tempo atual e calcula-se a diferença de tempo entre eles:

```
1 gettimeofday(&timeEnd, NULL);
2 timersub(&timeEnd, &timeInit, &timeDelta);
```

Por fim, o tempo percorrido, desde o inicio da criação do *socket*, ate o fim da execução é contabilizado e impresso na tela, junto com o número de bytes enviados.

3 Métodos de análise

Para facilitar a compilação e execução em qualquer ambiente, foi criado um repositório no *Github* contendo todos os arquivos utilizados para a construção e validação do sistema [2]. O ambiente de avaliação foi constituído de um laptop com ubuntu 18.03 que realizou a comunicação consigo mesmo (uma vez que não existia outro disponível) entre o cliente e o servidor. Para se estipular a funcionalidade do par de programas quanto a perda de pacotes, foi criada uma pseudo função que simulava a perda de pacotes aleatórios tanto no servidor quanto no cliente para o envio de alguma arquivo.

```
int cont = 0;
int tp_sendto(int so, char* buff, int buff_len, so_addr* to_addr)
{
    int count;
    //fprintf(stderr,"tp_sendto called (%d bytes)\n", buff_len);
    /*****************************************************
     * Aqui seria um bom lugar para injetar erros a fim de *
     * exercitar a funcionalidade do protocolo da camada   *
     * acima (o PJD).                                      *
     *****************************************************/
    if((cont != 8) && (cont != 5) && (cont != 13) && (cont != 12)){
        count = sendto(so, (void*)buff, buff_len, 0,
            (struct sockaddr*) to_addr, sizeof(struct sockaddr_in));
    }

    cont++;
    return count;
}
```

Figura 3: pseudo função que simula perda de pacotes

Para se medir a eficácia e vazão de dados do programa foi utilizado um arquivo para envio de tamanho 3 Megabytes, sendo que foram utilizados buffers de tamanho 2^N bytes, com $1 \leq N \leq 16$. O teste se repetiu 2 vezes, para se estipular um valor médio de taxa de envio o tempo médio e seu respectivo desvio padrão.

4 Resultados

O teste realizado para a perda de pacotes obteve sucesso, sendo transmitido o pacote por completo, podendo-se perceber o uso do temporizador de 1 segundo para o reenvio dos pacotes. Para se comparar a igualdade dos arquivos enviado e recebidos, foi utilizada a função do linux **du -hsb "nome**

[2]Repositório: https://github.com/luangnp98/socketUDPconection/

do arquivo" que retorna o tamanho em bytes do mesmo, de forma que em todos os casos, os tamanhos dos arquivos enviados e recebidos eram iguais.

O teste realizado com o arquivo de 3 Megabytes teve como intuito estipular o número de mensagens enviadas, o tempo total médio medido, o desvio padrão dos tempos medidos e a vazão média observada para cada arquivo de texto enviado. Para isso foram construídas as tabelas e gráficos abaixo, que representam esses valores.

4.1 Tabelas

Tamanho do buffer (bytes)	Tempo Médio	Desvio Padrão	Vazão Média(Kbps)
1	28,47	0,06	129,55
2	14,53	0,29	253,86
4	7,23	0,02	509,85
8	4,82	0,03	765,10
16	2,11	0,01	1751,91
32	1,07	0,10	3446,51
64	0,52	0,02	7160,72
128	0,28	0,00	13170,61
256	0,15	0,00	25389,13
512	0,10	0,01	38818,64
1024	0,05	0,01	72309,24
2048	0,05	0,00	81950,47
4096	0,04	0,02	85762,12
8192	0,03	0,02	117072,10
16384	0,01	0,00	263412,21
32768	0,01	0,00	386154,03

Tabela 1: Tabela de referência para o arquivo de 3 Mb

4.2 Gráficos

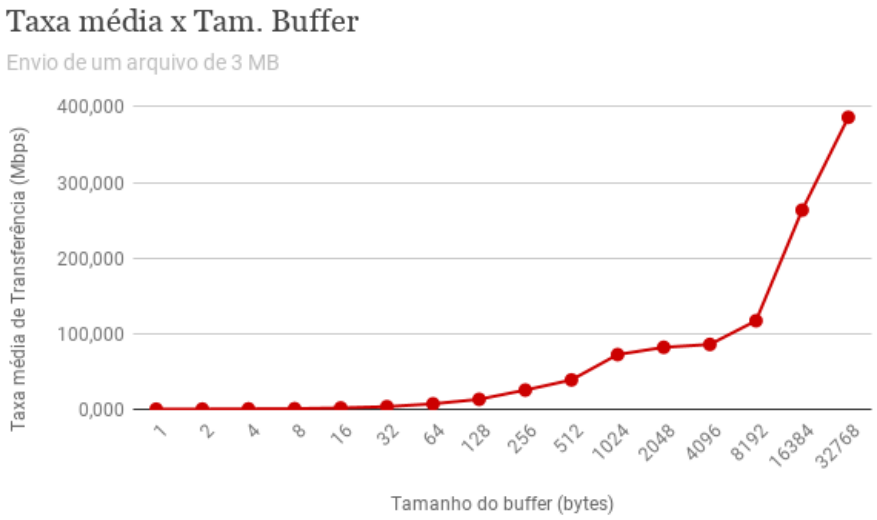

Figura 4: Taxa de transmissão média em Mbps para o arquivo de 3 Mb

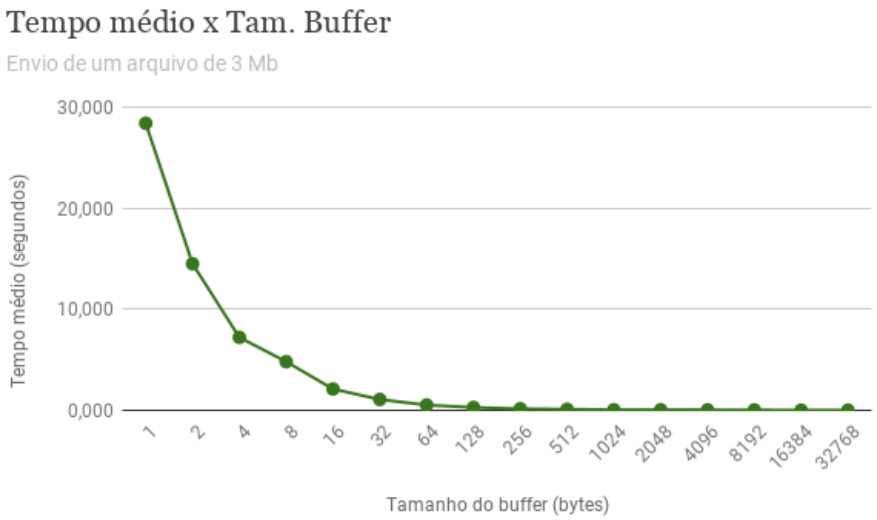

Figura 5: Taxa média de envio em segundos para o arquivo de 3 Mb

5 Análise dos resultados

Os dados das tabelas e dos gráficos exibidos acima, demonstram como o tamanho do *buffer* desempenha um papel importante na taxa de transmissão dos dados. Pode ser analisado que quando o *buffer* aumenta de tamanho de forma exponencial (nos gráficos o eixo das abcissas está linearizado), o resultado da taxa de transferência também tende a crescer de forma exponencial, isso demostra que a taxa de transferência representa uma função quase que **linear** em relação ao tamanho do *buffer*. Quando o buffer começa a ficar de certa forma maior, pacotes maiores começam a trafegar pela rede, o que pode gerar perda de pacotes e alguns atrasos a mais, fazendo o crescimento da taxa de envio perder um pouco da linearidade do crescimento. Em alguns pontos, existem certas descontinuidades da suavidade das curvas, devido provavelmente a alguma flutuação da rede.

O tempo médio também acompanha essa linearidade (nos gráficos o eixo das abcissas está linearizado), sendo que sempre que o tamanho do *buffer* é dobrado, o tempo cai quase que pela metade. Entretanto existe um tempo mínimo de setup dos programas, que impede que isso ocorra indefinidamente.

6 Conclusão

A construção desse trabalho apresentou diversos desafios quanto a implementação de uma entrega confiável para o protocolo *UDP*, uma vez que o mesmo não estabelece conexão, portanto uma nova estruturação do código, assim com o uso de novas ferramentas precisaram ser exploradas. A construção de um cabeçalho e o uso dos Acks, IDs e da temporização foram de suma importância para se obter sucesso no trabalho, de forma que essas ferramentas contribuíram de forma muito significativa para o aprendizado e para o uso em trabalhos futuros.

A construção de tabelas e de gráficos para mais arquivos de tamanhos variados em ambientes diferenciados, era de total interesse, entretanto exigiria muito trabalho e geraria material excessivo para a documentação do trabalho, de forma que se torna uma possível possibilidade no futuro. De uma forma geral o trabalho contribuiu significativamente para o aprendizado de forma que novos projetos possam ser desenvolvidos.