Redes de Computadores - Trabalho Prático 1

25/05/2018

Daniel da Silveira Ferro Campos - 2016020584
Vítor Mourão Hanriot - 2016108422

## 1. Introdução:

O trabalho consiste em promover uma comunicação cliente-servidor por meio de um par de programas. O cliente solicita conexão com o servidor, envia ao servidor uma string contendo o nome do arquivo requisitado, e o servidor envia ao cliente o arquivo. O protocolo utilizado é o UDP, ou seja, pacotes podem ser perdidos. Dessa forma, há de se garantir que pacotes perdidos serão reenviados.

## 2. Implementação:

O trabalho é uma síntese de dois códigos rodando em paralelo: o código do cliente e o código do servidor.

No código do cliente, existem 4 argumentos a serem passados para a função main do programa: IP do servidor - **nome_servidor**(argv[1]), porta do servidor - **porta_do_servidor**(argv[2]), nome do arquivo requisitado - **nome_do_arquivo** (argv[3]) e tamanho do buffer do cliente - **tam_buffer**(argv[4]). Checa-se se o usuário passou 4 argumentos no terminal de comando: se não, pede-se que o usuário digite novamente o comando de forma correta.

No código do servidor, existem 2 argumentos a serem passados para a função main do programa: porta do servidor - **porta_do_servidor**(argv[1]) e o tamanho do buffer do servidor - **tam_buffer**(arvg[2]). Da mesma forma que no cliente, checa-se se o usuário passou 2 argumentos corretamente no terminal de comando: se não, pede-se que o usuário digite novamente o comando de forma correta.

No cliente, cria-se uma variável **server** do tipo so_addr, que conterá o endereço IP do servidor à conectar.

Após a checagem de argumentos no cliente, o socket **sockfd** é criado para o servidor, à porta fixa 2000, utilizando a função **tp_socket**. Dessa forma, ao fazer os testes, ***a porta do servidor escolhida como argumento (argv[2] no código do cliente e argv[1] no código do servidor) deve ser diferente de 2000***.

Em seguida, conecta-se a porta do servidor ao seu IP por meio da função **tp_build_addr.** Se a conexão não for estabelecida, o programa retorna -1 com uma mensagem “ERRO”.

Após as definições de porta e IP, o cliente deve mandar o nome do arquivo para o servidor. Para isso a variável **pacote** foi criada. Essa variável é definida com o seu primeiro caracter sendo zero (pacote[0] = ‘0’), definindo que o primeiro bit ACK de envio será 0. Para enviar o **nome_do_arquivo**, usa-se a função memmove, que

define os caracteres de pacote[1] até pacote[sizeof(nome_do_arquivo)+1] como sendo a string nome_do_arquivo. Para enviar o pacote, usa-se a função tp_sendto com parâmetros sockfd, o buffer a ser enviado (pacote), o tamanho do buffer e o endereço do servidor (&server). Uma função tp_recvfrom com os mesmos parâmetros é chamda logo em seguida. Essa função recebe o nome do arquivo enviado pelo servidor, porém sem o bit ACK, e então imprime-se o nome do arquivo. Tal implementação é feita para mostrar que o servidor consegue separar o bit ACK dos dados.

Feito isso, o cliente começa a receber os dados do servidor. O loop de recepção de dados acontecerá enquanto a variável **tamanho_recebido** for diferente de **tamanho_do_cabecalho = 1**. A variável tamanho_recebido recebe o valor de retorno da função tp_recvfrom. Quando não tiver mais dados para enviar, o servidor enviará apenas um caracter de ACK. Dessa forma, 1 será atribuído à tamanho_recebido e o loop acaba.

No loop, uma variável **Tamanho_Arquivo** é incrementada em tamanho_recebido. Isso é feito para a análise de tempo de gettimeofday, que precisa do tamanho do arquivo para calcular a taxa de transferência.

Para checar se não há perda de dados, o bit 0 do pacote (recebido por tp_recvfrom) deve ser igual ao ACK.

Se sim, não houve perda de dados, e os dados podem ser escritos no arquivo (**received_file**). Para isso, uma variável **pacoteaux** é criada, e por memmove recebe apenas os dados de **pacote**. Essa variável é escrita no arquivo. Após isso, o cliente deve mudar o ACK, e enviar apenas o ACK(sem dados) para o servidor. Usa-se memset para limpar pacote e pacoteaux, troca-se o valor de ACK e esse valor é atribuído à pacote[0]. Finalmente, pacote (que contém apenas o ACK) é enviado ao servidor.

Se não, então os dados chegaram no cliente, porém o ACK enviado ao servidor não voltou corretamente (a figura 1 indica esse comportamento). Dessa forma, o cliente deve enviar o mesmo ACK, esperando receber o dado perdido. Além disso, se houver perda de dados Tamanho_Arquivo será incrementado mais de uma vez para o mesmo dado. Dessa forma, Tamanho_Arquivo é decrementado se houver essa perda.

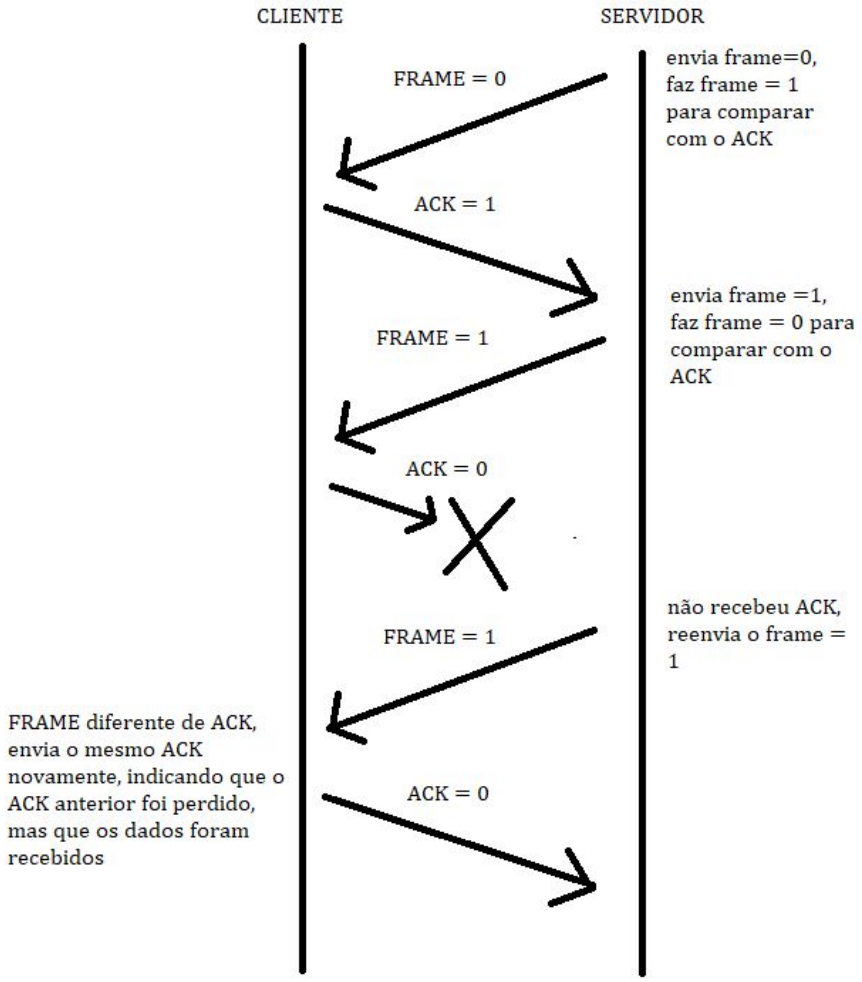



Figura 1: perda do pacote do cliente

Após isso, a lógica de análise de tempo é feita e o socket e o arquivo são fechados.

No servidor, cria-se uma variável **client** do tipo so_addr, que conterá o endereço do cliente que mandou o dado inicial.

Após a checagem dos argumentos, o socket **socket** é criado com a porta do servidor passada pelo argumento. O servidor aguarda receber dados do cliente, e quando isso acontece, a estrutura client recebe o endereço do cliente. Os primeiros dados enviados correspondem ao nome do arquivo. O servidor separa o bit frame ACK do nome do arquivo e reenvia o nome do arquivo para o cliente.

Uma variável **tv** do tipo struct timeval é criada. **tv.tv_sec** define por quantos segundos ocorre uma operação e **tv.tv_usec** define por quantos microssegundos ocorre a operação.

A função de temporização é criada usando a função **setsockopt.** Essa função define o limite de tempo da operação para socket passada pelo argumento &tv.

Após isso, o servidor começa a enviar os dados para o cliente. Isso acontece enquanto a variável **tamanho_lido** que corresponde ao tamanho do arquivo lido com a função fread for maior que zero. Nesse loop, há uma condição especial relacionada ao primeiro envio de dados, que não precisa conferir o ack. Para essa condição especial ocorrer uma vez, a variável conferir_primeira_leitura recebe um novo valor no final da condição.

Após o primeiro envio, se o servidor receber -1 na função tp_recvfrom, significa que o limite de temporização do socket passou e nada foi recebido (evento descrito na figura 2). Dessa forma, o servidor envia a variável **auxbuffer**, que contém o valor antigo de buffer.

Caso a função tp_recvfrom não retorne -1, significa que o ACK está correto e não foram perdidos dados. Nesse caso, o servidor atribuiu à auxbuffer os dados do arquivo, faz com que o **buffer** a ser enviado tenha em sua posição 0 o caracter frame e buffer[1] até buffer[tam_buffer-1] receba os dados lidos do arquivo (com a função memmove). Após isso, a variável auxbuffer recebe o valor de buffer, caso o servidor precise enviar os dados novamente, envia buffer e depois o zera. Finalmente, o caracter frame muda de valor, para ser comparado com o ACK posterior.

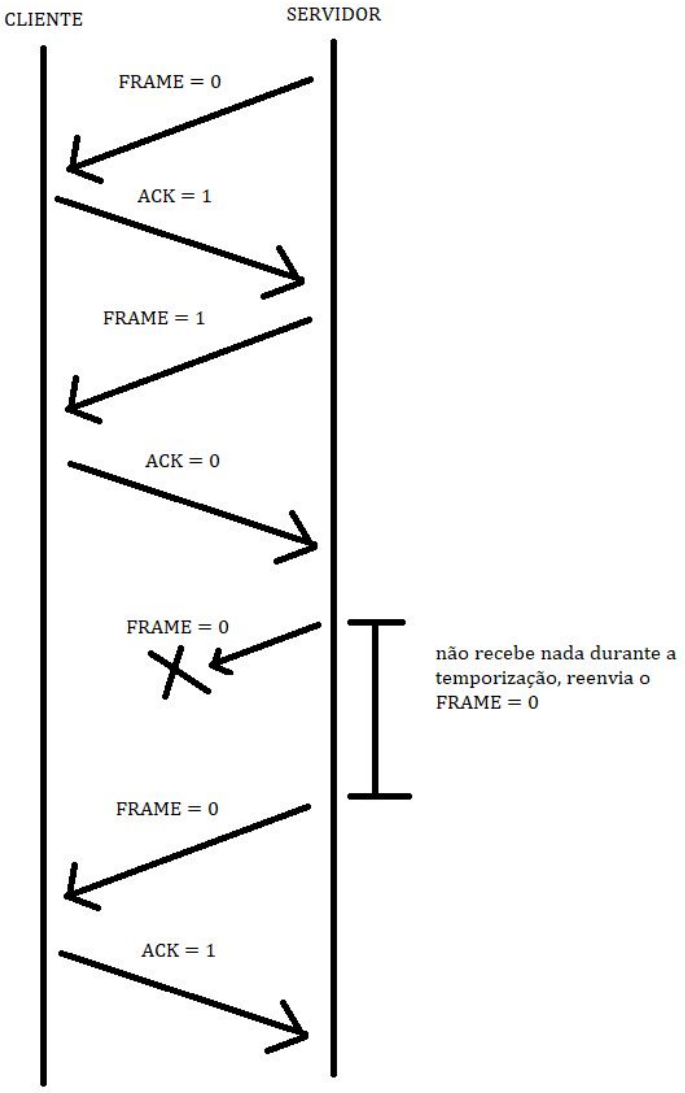



Figura 2: perda de pacote do servidor

## 3. Metodologia:

Para essa prática, foram utilizados dois computadores, um é um Sony com Processador Intel Core 2 Duo com 2GB de memória RAM, outro é um HP com processador AMD Turion 2 com 4GB de memória RAM. Para enviar os dados, usamos comunicação de uma mesma rede de WiFi, o da UFMG.

Como o foco do projeto é de enviar um arquivo de texto, testamos dois tipos de arquivos para enviar os dados de um para o outro, o conteúdo desse arquivo é um grande poema só para conferir se foi enviado corretamente sem nenhuma perda dos dados. Um dos arquivos tem 35,5kB e o outro tem 3,4MB de dados, com a

intuição de conferir a resposta do nosso programa para um arquivo consideravelmente pequeno e outro grande.

Para ser possível fazer análise de desempenho do nosso código, usamos a struct timeval para nos auxiliar na contagem do tempo em que o programa é executado com a função gettimeofday, um no início do código e outro no final e fizemos a diferença desses dois dados.

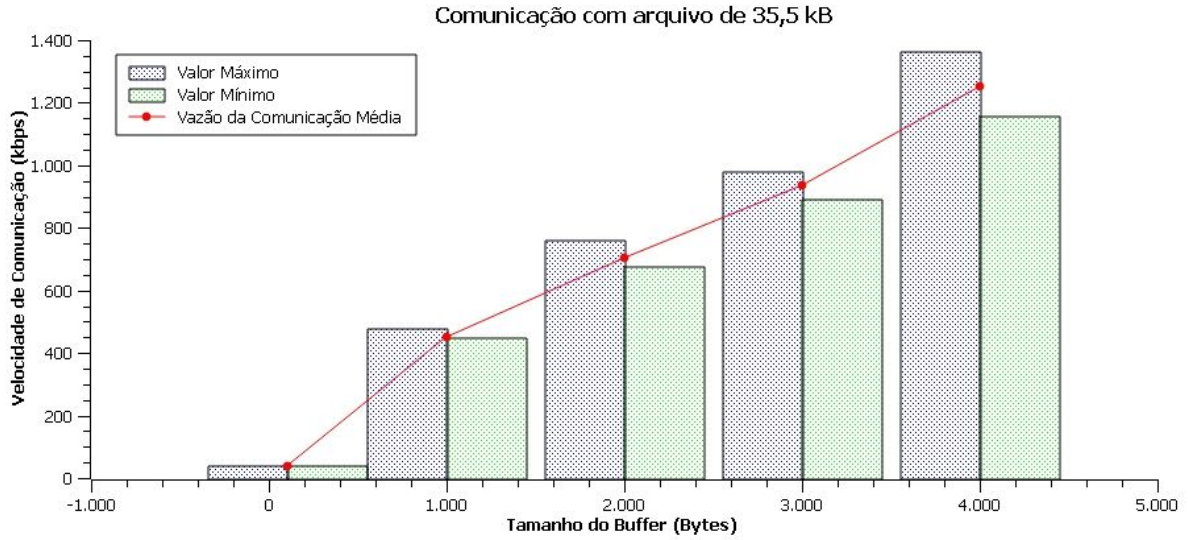
```
long int Tempo_micro = (end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 + start.tv_usec);
float Tempo_mili = Tempo_micro/1000;
float taxa_transmissao = Tamanho_Arquivo/Tempo_mili;
```

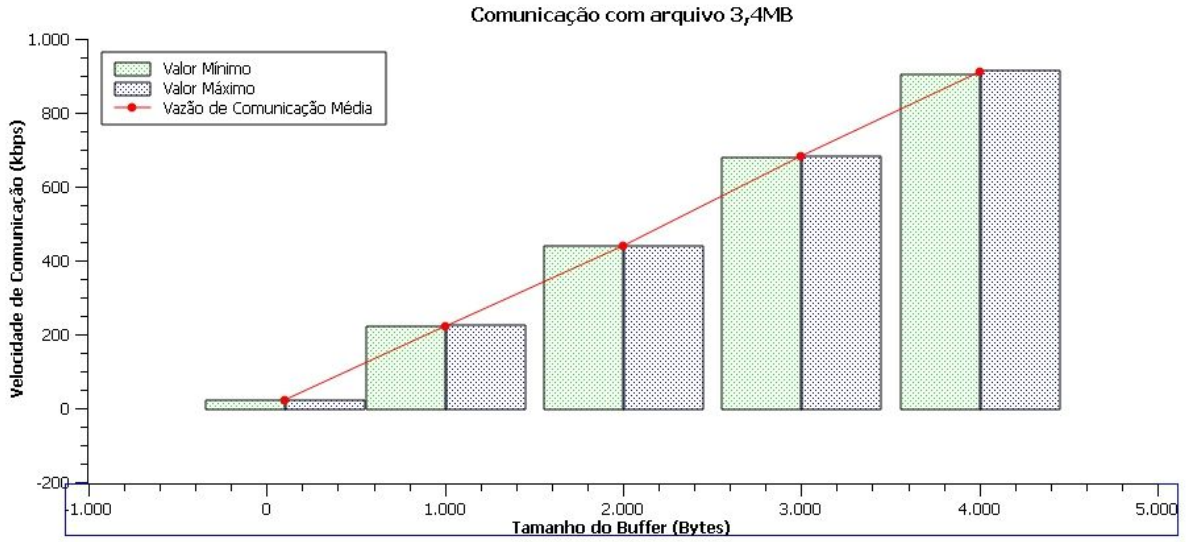
Como é mostrado na figura acima, após pegar esses dados, fazemos a diferença entre os dois tempos e em seguida fazemos a taxa de transmissão como o tamanho do arquivo dividido pelo tempo executado. Com a lógica do tempo feito, fizemos várias execuções repetidas deste código. Para cada arquivo, testamos 5 tamanhos de buffers diferentes, 100Bytes, 1000Bytes, 2000Bytes, 3000Bytes e 4000Bytes, e para cada tamanho de buffer, foi feito o envio dos dados 5 vezes para obtermos 5 tempos diferentes.

Assim, foi possível ver a transmissão média enviada pelo nosso código e conseguir ótimos valores para junção dos dados. Vale ressaltar que no nosso código, ao enviar uma vez o arquivo ele finaliza o código, então não foi criado um loop para que o programa pudesse enviar mais de um arquivo por vez.

## 4.Resultados:

Após registrar todos os dados, foi então colocado esses dados em gráficos para ficar visualmente mais fácil de analisar. Esses gráficos se encontram a seguir muito bem detalhados:






## 5.Análise:

Ao observar os gráficos, podemos ver que o nosso código tem um bom desempenho ao que foi proposto pela atividade, já que era esperado obter uma uma curva linear, e como pode observar, obtivemos algo próximo disso se levarmos em conta os valores médios de vazão de comunicação.

As curvas obtidas são lineares acontecem devido ao fato de que a relação ideal entre tamanho do buffer e vazão de comunicação é proporcional, ou seja, quanto maior o buffer, mais rápido será de enviar os dados. Mas sabemos que para esse projeto com comunicação UDP, temos que é muito fácil acontecer perda de dados ao longo do envio de dados.

Esse comportamento é observável quando observamos como varia o tempo decorrido do projeto para mesmas condições iniciais de simulação. Quando foi feito um teste de 5 vezes de simulação para cada caso, conseguimos ver uma variação do tempo de execução. Mas o que é interessante ver é que a taxa média respeita a ideia linear.

Uma outra relação observado ao coletar esses dados é como que ao aumentar o buffer aumentou a variação dos valores. Ou seja, quanto menor o tamanho do buffer, menos variava o tempo e quanto maior era o buffer mais variava. Isso também já era esperado, em que ao aumentar o buffer, cada envio de pacote possui um tempo maior, e se um pacote é perdido, então termos que gastar esse tempo maior novamente para reenviar esse pacote.

Assim, ao aumentar o tamanho do buffer, mais rápido é a transmissão, mas também menos próximo o tempo gasto em relação ao da média.

Existe mais um caso para ser analisado que é comparação entre enviar um arquivo pequeno com o arquivo maior. O resultado esperado continua sendo o mesmo, ou seja, uma proporção linear. Mas podemos ver que para o buffer 4000, o

arquivo menor teve uma transmissão de dados mais rápido que a transmissão de dados de um arquivo grande. Esse caso mostra que no nosso código apesar da proporcionalidade entre o tamanho do buffer com a vazão de comunicação, esses parâmetros não são proporcionais ao tamanho do arquivo.

## 6.Conclusão:

Assim, com esse trabalho pudemos aprofundar nossos conhecimentos na linguagem C de programação, como também aprofundar em nossos conhecimentos sobre a matéria de Redes de Computadores. Nisso, tivemos que aprender a fazer corretamente o funcionamento da comunicação de um servidor e cliente e tivemos que reforçar o que foi ensinado sobre o método de segurança para não ser perdido os dados na comunicação UDP, já que é uma comunicação não confiável.

Nisso, fizemos corretamente a lógica do stop and wait simples, em que o servidor espera a resposta do cliente que recebeu corretamente o pacote, e também conseguimos aplicar o temporizador além de configurar o melhor tempo para esse temporizador, para não tornar esse processo de enviar dados lento.