

Redes de Computadores

Trabalho Prático 2

Carolina Coimbra Vieira

Novembro de 2017

1 Introdução

O presente trabalho objetiva apresentar e analisar a metodologia e os resultados da implementação de um par de programas que operam no modelo cliente-servidor e exercitam tanto a transmissão unidirecional quanto a comunicação do tipo requisição-resposta sobre o protocolo UDP, implementando um protocolo de confirmação. Para isso, faz-se necessário utilizar a biblioteca de sockets do Unix (Linux).

A tecnologia cliente-servidor é uma arquitetura na qual o processamento da informação é dividido em módulos ou processos distintos. Um processo é responsável pela manutenção da informação (servidores) e outros responsáveis pela obtenção dos dados (os clientes). Os processos cliente enviam pedidos para o processo servidor, e este por sua vez processa e envia os resultados dos pedidos [2]. Nesse caso, em particular, o cliente será responsável por enviar o nome do arquivo que o servidor deverá enviar para ele, dividindo em buffers de tamanho determinado a priori.

Os Sockets são uma abstração de endereços de comunicação que consistem de um nome de host e um número de porta. Os sockets surgiram no sistema operacional BSD Unix e, atualmente, são responsáveis pela interação e comunicação de programas ao longo da Internet. Além disso, os sockets são a interface padrão para a comunicação entre processos em redes TCP/IP [4], sobretudo por fornecer uma interface para a camada de aplicação se comunicar com a camada de transportes dessa arquitetura. Ou seja, os sockets UDP e TCP são a interface provida pelos respectivos protocolos na interface da camada de transporte [1] e, programar com sockets pode ser visto como desenvolver um protocolo de aplicação.

A comunicação entre cliente-servidor se dá sobre o protocolo UDP (*User Datagram Protocol*). Esse protocolo não é orientado para a conexão e é considerado muito simples, uma vez que não fornece controle de erros [3]. Entretanto, a confirmação dos datagramas, inexistente no protocolo UDP, deverá ser implementada no trabalho de forma a tratar possíveis erros ou perdas.

Dessa forma, a implementação do programa deve levar em consideração todas as especificações acima. Primeiro dois programas principais são criados,

bem como a criação dos sockets em cada um deles. Após isso, ocorre a troca de dados que foi implementada de forma confirmada sobre o protocolo UDP. Para isso, um cabeçalho foi criado a fim de definir o número de sequência e número de reconhecimento de cada um dos datagramas enviados. Além disso, um temporizador foi acionado a cada mensagem enviada para que ocorra a retransmissão dos datagramas quando ocorrer um evento de *timeout*. E, por fim, várias análises e testes são realizados para avaliar o desempenho dessa comunicação variando o tamanho do buffer e do arquivo que será enviado e, principalmente, avaliando o impacto de um canal de transmissão com erros sobre o tempo e o desempenho dessa comunicação.

O trabalho se organiza da seguinte forma. Primeiro, na Seção 2 é apresentado o cabeçalho do protocolo, detalhes da implementação do protocolo *stop and wait*, decisões de projeto envolvidas e o tratamento de temporizações. Na Seção 3 é apresentada a metodologia contendo dados sobre os experimentos como a configuração da máquina utilizada, a localização da mesma na rede além da descrição de como foram realizadas as medições. Então, na Seção 4 são apresentados os resultados obtidos após a realização de diversos experimentos. A análise mais aprofundada sobre cada um dos experimentos é descrita na Seção 5. Finalmente, na Seção 6, é feita uma breve conclusão sobre o trabalho.

2 Implementação

A implementação consiste de um par de programas que operam no modelo cliente-servidor com comunicação via protocolo UDP, utilizando uma biblioteca de sockets do Unix. Entretanto, mesmo utilizando o UDP que, não garante a entrega das mensagens, essa garantia deverá ser implementada. Sendo assim, os programas cliente e servidor são implementados seguindo um padrão simples, conforme descrito a seguir:

Cliente:

```
processa argumentos da linha de comando:
host_do_servidor porto_servidor nome_arquivo tam_buffer
chama gettimeofday para tempo inicial
envia string com nome do arquivo (terminada em zero)
abre arquivo que vai ser gravado - pode ser fopen(nome,"w+")
loop recv buffer até que perceba que o arquivo acabou
escreve bytes do buffer no arquivo (fwrite)
atualiza contagem de bytes recebidos
fim loop
fecha arquivo
chama gettimeofday para tempo final e calcula tempo gasto
imprime resultado:
"Buffer = %5u byte(s), %10.2f kbps (%u bytes em %3u.%06u s)
fim cliente.
```

Servidor:

processa argumentos da linha de comando:
porto_servidor tam_buffer
faz abertura passiva e aguarda conexão
recebe o string com nome do arquivo
abre arquivo que vai ser lido – pode ser `fopen(nome,"r")`
se deu erro, fecha conexão e termina
loop lê o arquivo, um buffer por vez até `fread` retornar zero
envia o buffer lido
se quiser, contabiliza bytes enviados
fim loop
fecha arquivo
chama `gettimeofday` para tempo final e calcula tempo gasto
se quiser, imprime nome arquivo e no. de bytes enviados
fim servidor.

De forma resumida, o cliente deve enviar um string para o servidor com o nome do arquivo desejado, receber o arquivo um buffer de cada vez e salvar os dados no disco à medida que eles chegam. Quando não houver mais bytes para ler o cliente fecha o arquivo e gera uma linha com os dados da execução. O servidor por sua vez deve operar de forma complementar, conforme exemplificado na Figura 1.

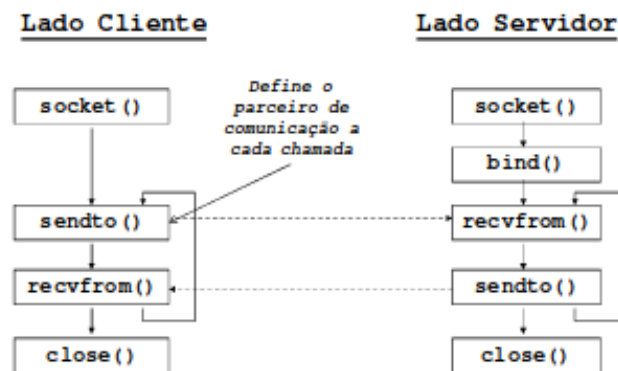


Figura 1: Diagrama cliente-servidor UDP

A comunicação entre esses processos está representada, de forma geral, na Figura 1. Nela, é possível perceber que, as estruturas são criadas e, ao contrário do TCP, não ocorre o estabelecimento da conexão. Após a criação dos sockets, os mesmos já podem iniciar a comunicação. Nesse caso, cada um dos processos deve especificar nas funções de envio e recebimento o interlocutor que desejam alcançar.

A maior dificuldade, no entanto, é garantir que as mensagens serão confirmadas mesmo utilizando o protocolo UDP. Dessa forma, foi necessário adicionar um cabeçalho extra em todas as mensagens trocadas entre cliente e servidor, para adicionar o número de sequência da mensagem, bem como o número de reconhecimento. Essa alteração é necessária tendo em vista que o cabeçalho do protocolo UDP é bem simplificado, conforme mostrado na Figura 3, uma vez que não há garantias de entrega. Como referência a um protocolo que garante a entrega das mensagens foi analisado o TCP que, devido prover esse tipo de serviço, possui um cabeçalho mais completo. Sendo assim, analisando e comparando os cabeçalhos, chegou-se à conclusão de que adicionando os campos de número de sequência e número de reconhecimento seria suficiente para confirmar as mensagens trocadas.

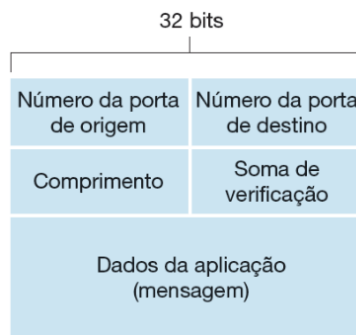


Figura 2: Cabeçalho do protocolo UDP

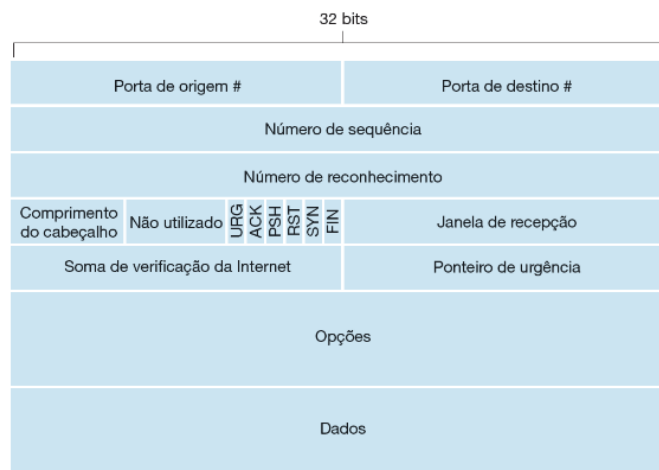


Figura 3: Cabeçalho do protocolo TCP

Além disso, por se tratar de um canal não confiável, é necessário tratar possíveis perdas de datagramas. Sendo assim, um temporizador é ativado pelo servidor com duração de 1 segundo para que ocorra um evento de *timeout* e o datagrama seja reenviado para o cliente caso não tenha sido confirmado.

No que diz respeito ao cabeçalho das mensagens, por se tratar de um protocolo *stop-and-wait*, similar ao representado na Figura 4, o número de sequência e reconhecimento foram incorporados nas posições iniciais do vetor de dados. Esses valores são números inteiros e sequenciais, incrementados de um em um na medida em que os datagramas são confirmados pelo programa destino. Por se tratar de um valor que ocupa uma posição em um vetor de caracteres, cada número é representado por um byte (8 bites). Dessa forma, a cada vez que eram incrementados, foi necessário fazer o módulo desses valores por 127, a fim de se evitar um *overflow*. Sendo assim, a cada datagrama recebido, o cliente, por exemplo, enviava um datagrama de reconhecimento, incrementando sempre em uma unidade o campo que representa o número de sequência, bem como de reconhecimento. De forma complementar, a cada novo datagrama enviado pelo servidor, o número de sequência do mesmo era incrementado, para ambos os casos, se o número de sequência do datagrama recebido anteriormente era o esperado pelo programa.

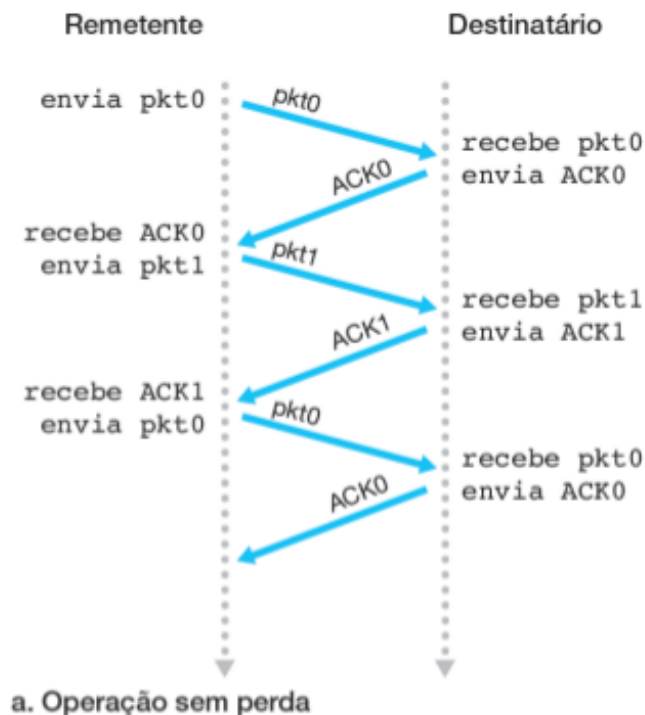


Figura 4: Funcionamento sem perdas de um protocolo *stop-and-wait*

A implementação do cabeçalho com o número de sequência e número de reconhecimento como valores inteiros e sequenciais sobre o protocolo do tipo *stop-and-wait*, baseou-se no princípio de que novos datagramas só são enviados depois que os anteriores são confirmados. Além disso, uma decisão tomada nesse projeto foi considerar que o envio do nome do arquivo pelo cliente se dará de forma segura. Ou seja, uma função de recebimento seguro foi implementada a fim de garantir que, com certeza, o servidor irá receber o nome do arquivo enviado pelo cliente. Essa mensagem inicial representaria uma "falsa conexão" entre o par de programas. A troca de mensagens após esse envio do nome do arquivo se dá sobre um canal com probabilidade p de haver erros e os datagramas são serem recebidos. E, por fim, para encerrar a comunicação entre cliente e servidor, foi assumido que o recebimento do reconhecimento do fim da transmissão do arquivo por parte do servidor também será garantida. Sendo assim, o início e o fim da transmissão são assumidos como datagramas que, garantidamente, chegarão ao seu destino.

O temporizador, por sua vez, baseou-se no conceito de temporização por tratamento de sinal. Dessa forma, para cada nova mensagem enviada pelo servidor durante o envio dos dados do arquivo, um temporizador era inicializado de forma que, no caso de um *timeout* e um não recebimento de uma mensagem de confirmação do cliente, o servidor deveria reenviar o último pacote enviado ao cliente. Foi definido um tempo de espera igual a 1 segundo, ou seja, se em 1 segundo, o servidor não recebe uma confirmação do cliente, sua última mensagem enviada ao cliente é reenviada. Um evento de *timeout* pode estar associado a alguns fenômenos, como mostram as Figuras 5, 6 e 7. As causas mais comuns são: perda do pacote enviado, perda do pacote de confirmação recebido ou temporização prematura e, em ambos os casos, o evento de *timeout* faz com que o servidor reenvie o último pacote, quantas vezes for necessário.

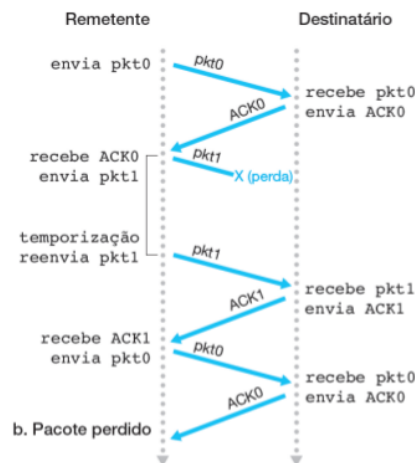


Figura 5: Retransmissão por causa de um pacote perdido

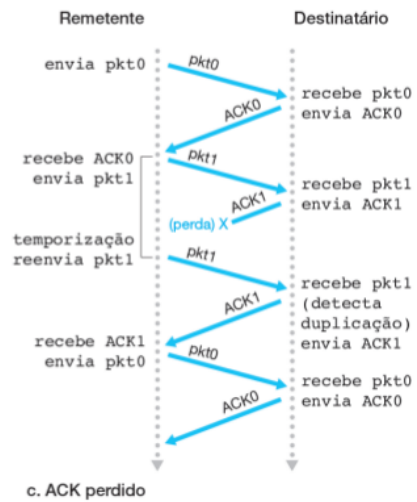


Figura 6: Retransmissão por causa de um ack perdido

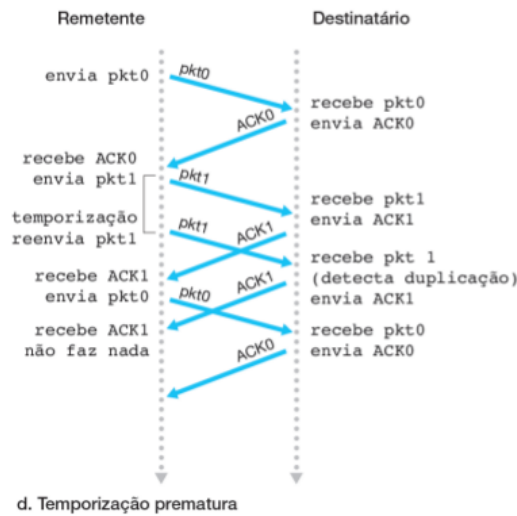


Figura 7: Retransmissão por causa de uma temporização prematura

Sendo assim, a implementação se resume em utilizar um cabeçalho com número de sequência e de reconhecimento de datagramas aliado ao conceito de temporização para garantir a confirmação durante a troca de mensagem entre cliente e servidor. Dessa forma, as configurações dos testes e máquinas usadas para testar essa implementação são apresentadas na seção seguinte.

3 Metodologia

A metodologia apresentada se refere à implementação e testes de um par de programas que operam no modelo cliente-servidor com comunicação via protocolo UDP, utilizando uma biblioteca de sockets do Unix. Entretanto, mesmo utilizando o UDP que, não garante a entrega das mensagens, essa garantia foi implementada, conforme descrito na seção anterior.

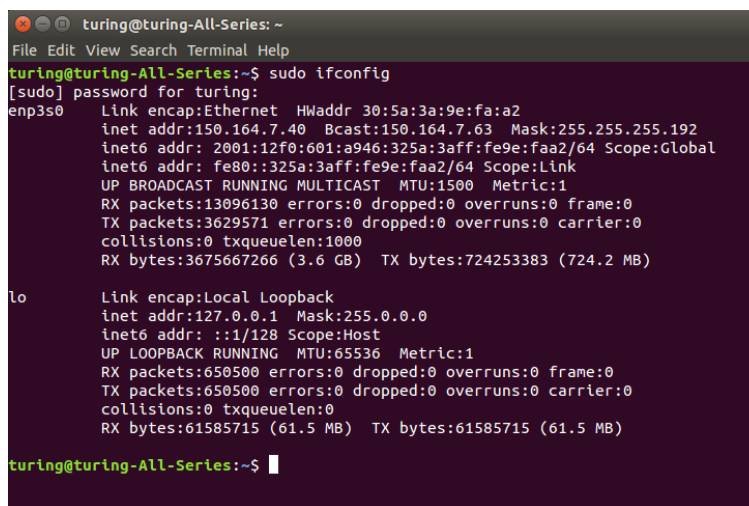
Todos os testes a serem descritos foram executados utilizando a mesma máquina, cujas informações principais referentes ao poder de processamento, memória e placa de rede são descritas a seguir:

Mémoire RAM: 16GB

Processador: Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz

Placa de rede: 03:00:0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller (rev 0c)

E, por fim, a máquina está localizada no laboratório *WISEMAP* do Departamento de Ciência da Computação na UFMG e possui o seguinte endereço IP: 150.164.7.40. Na Figura 8 é possível visualizar algumas configurações de rede da máquina.

A terminal window titled 'turing@turing-All-Series: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The user has run 'sudo ifconfig' and entered the password 'turing'. The output shows details for the 'enp3s0' interface (Ethernet) and the 'lo' interface (Local Loopback).

```
turing@turing-All-Series:~$ sudo ifconfig
[sudo] password for turing:
enp3s0    Link encap:Ethernet  HWaddr 30:5a:3a:9e:fa:a2
          inet addr:150.164.7.40  Bcast:150.164.7.63  Mask:255.255.255.192
          inet6 addr: 2001:12f0:601:a946:325a:3aff:fe9e:faa2/64 Scope:Global
          inet6 addr: fe80::325a:3aff:fe9e:faa2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13096130 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3629571 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3675667266 (3.6 GB)  TX bytes:724253383 (724.2 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:650500 errors:0 dropped:0 overruns:0 frame:0
          TX packets:650500 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:61585715 (61.5 MB)  TX bytes:61585715 (61.5 MB)

turing@turing-All-Series:~$
```

Figura 8: Especificações da rede do computador utilizado nos testes

O servidor irá rodar na máquina, ou seja, ele usa o IP da máquina e, é necessário definir a porta através da qual o cliente e o servidor irão se comunicar. Já para o cliente, é necessário especificar o IP onde o servidor está rodando e a porta correspondente. Dado essas especificações, a comunicação pode ser exemplificada conforme mostra a Figura 9.

O servidor deve ser o primeiro a ser inicializado e, a partir de então, ele estará sempre "escutando" a porta. Ao contrário do protocolo TCP, o servidor UDP não aceita conexões assim como o cliente UDP não se conecta com o

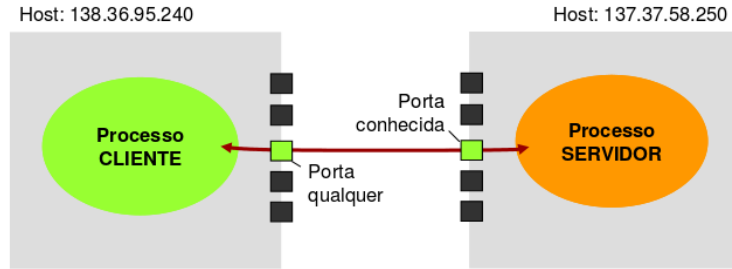


Figura 9: Diagrama representando como se dá a comunicação entre o cliente e servidor dado as especificações do IP e a porta de comunicação com o servidor

servidor. O que ocorre é que o servidor recebe diretamente as mensagens do cliente, assim como o servidor envia mensagens diretamente ao cliente. Ou seja, a comunicação do cliente com o servidor ocorre sem o estabelecimento de uma conexão a priori. Os parâmetros que o cliente recebe são: host do servidor (`argv[1]`), número da porta (`argv[2]`), nome do arquivo (`argv[3]`) e o tamanho do buffer (`argv[4]`) enquanto que o servidor recebe o número da porta (`argv[1]`) e o tamanho do buffer (`argv[2]`).

Vale ressaltar que, após o envio do nome do arquivo do cliente para o servidor, o servidor é responsável por abrir e enviar o mesmo para o cliente, buffer por buffer. Ao receber, o cliente salva esse conteúdo em um outro arquivo de extensão ".out". Dessa forma, ambos os arquivos, terão, ao fim da comunicação, o mesmo conteúdo, podendo ser verificado pela função *md5sum*.

Os testes sobre essa comunicação cliente-servidor, foram executados medindo o tempo decorrido desde o envio da primeira mensagem do cliente para o servidor até a finalização da comunicação, após o servidor enviar todo o arquivo solicitado pelo cliente. Sendo assim, o mesmo código foi executado repetidas vezes variando alguns parâmetros como o tamanho do arquivo, tamanho do buffer e probabilidade de erros no canal de comunicação. Nas Seções 4 e 5 esses testes serão melhor apresentados e analisados.

4 Resultados

Os principais experimentos executados após a implementação do par cliente-servidor envolveram testes de desempenho medindo o tempo que leva a comunicação e completa transferência do arquivo. As hipóteses levantadas sugerem que esse desempenho é afetado principalmente com a variação do tamanho do arquivo que se deseja transferir, o tamanho do buffer através do qual o mesmo será transferido e, principalmente, a probabilidade de haver perdas de datagramas durante a transferência.

No que diz respeito à relação entre o tamanho do buffer e o tempo de comunicação entre cliente-servidor, foram realizados testes variando o tamanho

do buffer em: 100 bytes, 1.000 bytes e 4.000 bytes. Nesse teste, o tamanho do arquivo considerado era de 3MB e para cada tamanho de buffer e foi assumida probabilidade de 25% de erro no canal. A Figura 10 representa o gráfico obtido com tempos encontrados para cada tamanho de buffer usado. O eixo y representa o tempo médio da troca de dados, medido em segundos, e o eixo x representa o tamanho do buffer utilizado.

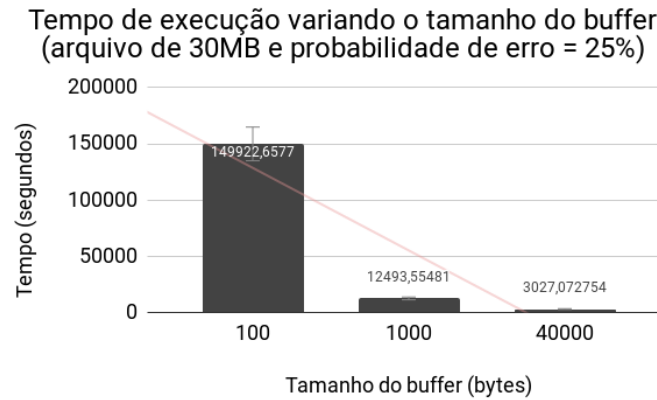


Figura 10: Gráfico representando o tempo médio de comunicação dado a variação no tamanho do buffer usado. Fixado um tamanho de arquivo igual a 3MB e uma probabilidade de erros igual a 25% e variando o tamanho do buffer em: 100, 1.000 e 4.000 bytes.

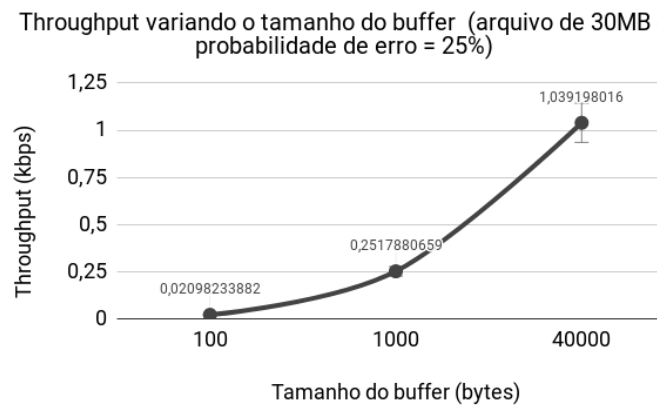


Figura 11: Gráfico representando o *throughput* dado a variação no tamanho do buffer usado. Fixado um tamanho de arquivo igual a 3MB e uma probabilidade de erros igual a 25% e variando o tamanho do buffer em: 100, 1.000 e 4.000 bytes.

Similarmente ao gráfico representado na Figura 10, a Figura 11 apresenta um gráfico com o *throughput* médio da comunicação, utilizando os dados do teste anterior. Porém, o eixo y representa agora a razão entre o número total de bytes enviados pelo tempo medido no cliente. Nesse gráfico, é possível visualizar o crescimento da taxa de envio com o aumento do tamanho do buffer.

Outro fator que também influencia no tempo de comunicação é o tamanho do arquivo que se deseja enviar. Nesse caso, fixado um tamanho de buffer igual a 4.000 bytes e uma probabilidade de erros igual a 1%, escolhidos de forma arbitrária, foram realizados testes para verificar a influência dos arquivos de tamanho 3MB, 6MB e 9MB no tempo de comunicação. A Figura 12 mostra o gráfico resultante desses testes realizados. Nela é possível se ter uma noção do quanto o tamanho do arquivo contribui para o aumento do tempo de troca de dados entre o cliente e o servidor. O eixo y representa o tempo médio da troca de dados, medido em segundos, e o eixo x representa o tamanho do arquivo enviado, dado um buffer constante igual a 4.000 bytes e uma probabilidade de erros igual a 1%.

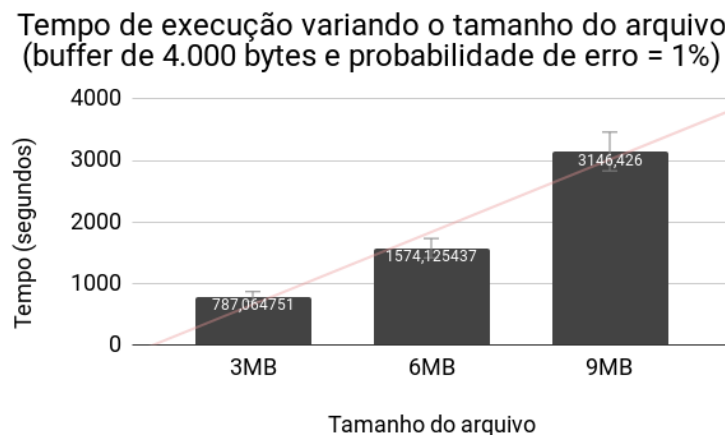


Figura 12: Gráfico representando o tempo médio de comunicação dado a variação no tamanho do arquivo a ser enviado. Fixado um tamanho de buffer igual a 4.000 bytes e uma probabilidade de erros igual a 1% e variando o tamanho do arquivo em: 3, 6 e 9 MB.

E, por fim, é possível comparar a influência da probabilidade de erros, como por exemplo, perda de datagramas, sobre o tempo de comunicação, conforme mostra o gráfico da Figura 13. É possível perceber que, de fato, o aumento da probabilidade de erros influenciam muito no aumento do tempo de comunicação. O eixo y representa o tempo médio da troca de dados, medido em segundos, e o eixo x representa as probabilidade de erro consideradas, sendo elas: 0%, 25%, 50% e 75%.

Além da relação entre as probabilidades de erro e o tempo de comunicação entre cliente e servidor para a completa transferência do arquivo, a Figura 14

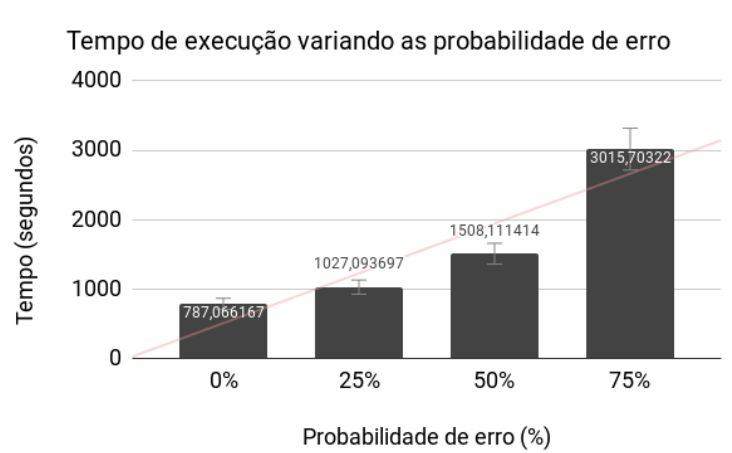


Figura 13: Gráfico representando o tempo médio de comunicação dado a variação na probabilidade de erros no canal. Fixado um tamanho de arquivo igual a 3MB e um buffer de 4.000 bytes e variando as probabilidades de erro em: 0%, 25%, 50% e 75%.

apresenta um gráfico com o *throughput* médio da comunicação, utilizando os dados do teste anterior. Porém, o eixo y representa agora a razão entre o número total de bytes enviados pelo tempo medido no cliente. Nesse gráfico, é possível visualizar o decrescimento da taxa de envio com o aumento das probabilidades de erro.

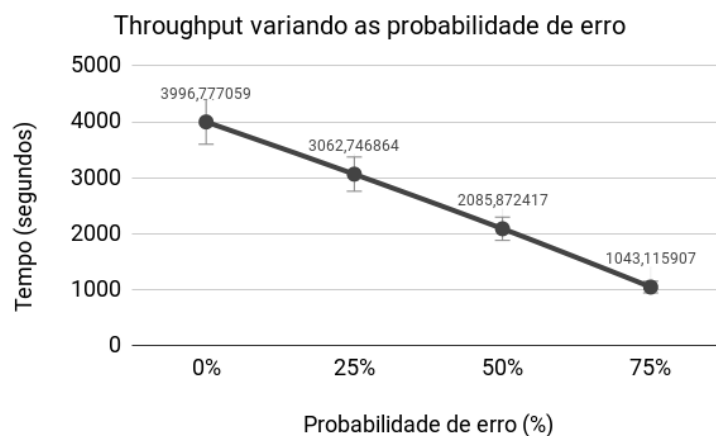


Figura 14: Gráfico representando o *throughput* dado a variação na probabilidade de erros no canal. Fixado um tamanho de arquivo igual a 3MB e um buffer de 4.000 bytes e variando as probabilidades de erro em: 0%, 25%, 50% e 75%.

5 Análise

Os resultados encontrados na Seção 4 refletem bem o que era esperado sobre essa comunicação. O fato do tempo médio de comunicação aumentar com a diminuição do tamanho do buffer mostra que, de fato, foi necessário fazer mais envios ao cliente, aumentando o tempo de comunicação. Esse resultado é sustentado também pelos testes em que mostram o *throughput* da comunicação, isto é, a taxa de transferência obtida entre cliente e servidor (basicamente, o número total de bytes enviados dividido pelo tempo medido no cliente) com a variação do tamanho do buffer.

Assim como o tamanho do buffer, o tamanho do arquivo que se deseja enviar ao cliente influencia no tempo de comunicação. Entretanto, essa relação é direta, uma vez que o aumento no tamanho do arquivo provoca um aumento no tempo que leva a fase de troca de dados. Esse resultado também foi comprovado pelos testes realizados.

E, finalmente, os testes referentes à variação na probabilidade de erros no canal, comprovam que, de fato, quando maior a probabilidade de erros, maior será o tempo necessário para a completa transferência do arquivo. E, dessa forma, menor será o *throughput*. Esse resultado comprova a importância de um canal confiável para uma comunicação cliente-servidor, uma vez que a não garantia de um canal confiável aumenta consideravelmente o tempo de comunicação, reduzindo drasticamente o *throughput* ou desempenho da mesma. Entretanto, devido ao fato da maioria dos canais serem suscetíveis a erros, esses testes mostram que, para aplicações que são tolerantes a perdas, de fato, não vale a pena implementar um protocolo de confirmação de mensagens devido ao tempo que o mesmo acrescenta à comunicação. Porém, de forma inversa, é possível perceber que, para aplicações que exigem a entrega, é muito importante tratar essas perdas, mesmo que isso implique em um aumento nesse tempo.

6 Conclusão

O presente trabalho foi desenvolvido com o objetivo de implementar um par de cliente-servidor que se comunicam via sockets UDP, implementando uma comunicação com confirmação de mensagens. Todo o código foi desenvolvido em linguagem C, usando a biblioteca de sockets.

Os testes realizados comprovam hipóteses formuladas anteriormente, referentes à influência do tamanho do buffer e o tamanho do arquivo nessa comunicação. Nesse caso, conforme esperado, o aumento no tamanho do buffer reduz significativamente o tempo que demora a comunicação. Assim como o aumento no tamanho do arquivo está diretamente relacionado ao aumento no tempo necessário para a completa transferência do mesmo. E, além disso, o impacto da confirmação de datagramas, perdas ao longo da comunicação, *timeout* e retransmissão de mensagens.

De modo geral, todas as hipóteses levantadas foram comprovadas pelos testes realizados. E, o mais interessante é perceber o impacto negativo causado por um

canal de comunicação com erros sobre o desempenho da comunicação cliente-servidor. De fato, os canais não confiáveis são uma realidade que devemos lidar com ela da melhor maneira possível. E, através desse trabalho, ficou claro que, para aplicações que são altamente tolerantes a falhas ou que ocorrem sobre canais com baixa taxa de erros, não vale a pena implementar um protocolo de confirmação. Porém, o impacto que uma perda de datagrama tem sobre certas aplicações pode fazer com que esse aumento no tempo de comunicação se torne vantajoso.

Por fim, o trabalho foi uma oportunidade de aprender e conviver com problemas práticos que são vividos no dia-a-dia e enfrentados no mercado de trabalho e que nos ensinam a ser autodidatas e transpor barreiras por necessidade e desejo próprio. E, de fato, após muita pesquisa e aprendizado, é possível desenvolver um trabalho bem estruturado e com resultados interessantes e consistentes.

Referências

- [1] Cesar Augusto Tacla. Sockets udp, tcp e multicast. Slides de aula.
- [2] UFRGS. Protocolos cliente/servidor. Página Web.
- [3] USP. Socket udp. Slides de aula.
- [4] Marcos Augusto M. Vieira. Sockets. Slides de aula.