

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIENCIAS EXATAS
DEPARTAMENTO DE CIENCIA DA COMPUTAÇÃO

Redes de Computadores

TRABALHO PRÁTICO 1
Sockets, medição de desempenho

Guilherme Saulo Alves

28 de Abril de 2015

Belo Horizonte – MG

1. Introdução

O objetivo deste trabalho prático consiste em implementar um par de programas que operem no modelo cliente-servidor e exercitar a comunicação do tipo requisição-resposta sobre o protocolo TCP. Na implementação foi utilizada a biblioteca de sockets do Unix e os programas funcionam para protocolos IPv4 e IPv6.

O paradigma computacional simulado neste trabalho consiste na Computação na Nuvem, onde os arquivos e processamento estão disponíveis em servidores. Os programas consistem em um cliente e um servidor, que funcionam da seguinte forma resumida: o cliente deve se conectar ao servidor e enviar o nome do diretório a ser transferido ao servidor. O servidor reconhece esse nome do diretório e cria um arquivo em disco que representa esse diretório. Em seguida o cliente começa transferir uma lista de arquivos ao servidor e este começa armazenar no arquivo. Quando não houver mais arquivos a ser transferidos, o cliente fecha a conexão e o servidor salva o arquivo que representa o diretório. Ao final da transferência o cliente imprime na tela informações sobre a transferência, como o tempo gasto e o throughput da transmissão.

2. Metodologia

As soluções dos problemas, configurações e decisões de implementação serão apresentadas a seguir:

2.1. Considerações Iniciais

O ambiente de desenvolvimento do código foi via compilador GNU GCC Compiler via linha de comando no Sistema Operacional Ubuntu Linux 13.04. Para gerar os programas executáveis do servidor e do cliente, utiliza-se o comando make no terminal do Linux. Os programas gerados possuem nome server e client e deve aceitar, na respectiva ordem, os seguintes argumentos:

```
./server <host_do_servidor> <porta_servidor>  
./client <host_do_servidor> <porta_servidor> <nome_diretorio>
```

O código está dividido em seis arquivos principais:

- server.c: Contém o código do servidor;
- client.c: Contém o código do cliente;
- io.c: Realiza a comunicação do programa com o ambiente, criando lista de arquivos de um diretório em um arquivo.
- io.h: Contém as declarações das funções e bibliotecas utilizadas em io.c
- showip.c: Especifica os endereços IPs ipv4 e ipv6 para um dado nome de hospedeiro.
- showip.h: Contém as declarações das funções e bibliotecas utilizadas em showip.c
- time.c: Define a função gettimeofday.
- time.h: Contém as declarações das funções e bibliotecas utilizadas em time.c

2.2. Decisões de Implementação

2.2.1. Argumento <host_do_servidor> <porta_servidor>

Para testar o trabalho, foi realizado testes apenas com o servidor rodando localmente para testar a comunicação consigo mesmas. Portanto o endereço IP a ser passado como parâmetro deve ser localhost ou 127.0.0.1 para protocolos IPv4 ou ::1 para protocolo IPv6. Para o argumento da porta do servidor, foi utilizado portas maiores que 5000. Ser capaz de se comunicar com a máquina atual como se estivesse se comunicando com uma máquina remota é útil para a finalidade de testes, assim como para usar recursos localizados na máquina atual, mas que se esperariam serem remotos. Logo não foi realizado experimentos com outro computador, utilizando rede sem fio.

2.2.2. Tratamento para protocolos IPv4 e IPv6

Para que os programas funcionassem para redes IPv4 e IPv6, foi necessário fazer modificações em algumas estruturas e funções utilizadas inicialmente para o protocolo IPv4. Primeiramente foi criada uma função *especifica_ip(argv)*, que recebe uma string com o endereço do servidor como parâmetro, processa essa string e retorna 4 caso o endereço seja IPv4 ou 6 caso o endereço seja Ipv6. Essa função trabalha com a struct addinfo e a função getaddrinfo da biblioteca socket para reconhecer a família do endereço ip. Ao serem executados, os programas do servidor e do cliente chamam essa função e identificam se o endereço passado por parâmetro é IPv4 ou IPv6, de acordo com o retorno da função. Identificado a família do IP, é necessário certificar-se de que está usando as estruturas e constantes corretas para o protocolo especificado, pois elas sofrem mudança de acordo com a família do IP usado. Segue abaixo as principais funções e estruturas de dados e constantes que sofrem mudança no tratamento do endereço:

Principais estruturas de dados e constantes para IPv4:	Principais estruturas de dados e constantes para IPv6:
<pre>struct sockaddr_in; socket(AF_INET, SOCK_STREAM, 0); serv_addr.sin_family = AF_INET; serv_addr.sin_addr.s_addr = INADDR_ANY; serv_addr.sin_port = htons(porta_servidor); gethostbyname(argv[1]);</pre>	<pre>struct sockaddr_in6; socket(AF_INET6, SOCK_STREAM, 0); serv_addr.sin6_family = AF_INET6; serv_addr.sin6_addr = in6addr_any; serv_addr.sin6_port = htons(porta_servidor); gethostbyname2(argv[1], AF_INET6);</pre>

2.2.3. Cria lista de arquivos de um diretório

Para testar esses programas, foi criado um arquivo que contém uma lista de arquivos presente no diretório passado como parâmetro do cliente. Por exemplo, se o usuário digitar o diretório /home/ como argumento, a função cria_lista_arquivos() percorre todo o diretório e salva o nome dos arquivos em um arquivo chamado “_outro_dir.txt”. Se o diretório passado como argumento for o diretório atual “.”, o nome do arquivo será chamado _dir_local. Esse nome foi pré_estabelecido porque não é permitido criar nomes de arquivos que contem os caracteres “.” e “/” no Linux.

2.2.4. Fim conexão, “bye” e técnica *byte stuffing*

Devido ao tempo corrido, não foi possível implementar o envio “bye” do cliente para o servidor. Tentei implementar, porém o “bye” escrito no buffer estava sendo gravado no arquivo e isso não devia acontecer. Consequentemente não foi usada a técnica de *byte stuffing*. Porém, isso não impede que os programas sejam executados normalmente. Logo para fins de testes, não deve ter nenhum arquivo chamado “bye” no diretório.

2.2.5. Medições de desempenho

Inicialmente o servidor é ligado e o cliente transmite o nomes dos arquivos um por um. Com isso, é computado o tempo total gasto, a quantidade total de bytes transmitidos e a velocidade de transmissão. O tempo gasto foi medido utilizando a função `gettimeofday` e os bytes transmitidos foram medidos com o retorno da função `write`. De posse do tempo gasto e do total de bytes transmitidos, o cliente faz o cálculo da velocidade da transmissão e exibe essas informações no terminal.

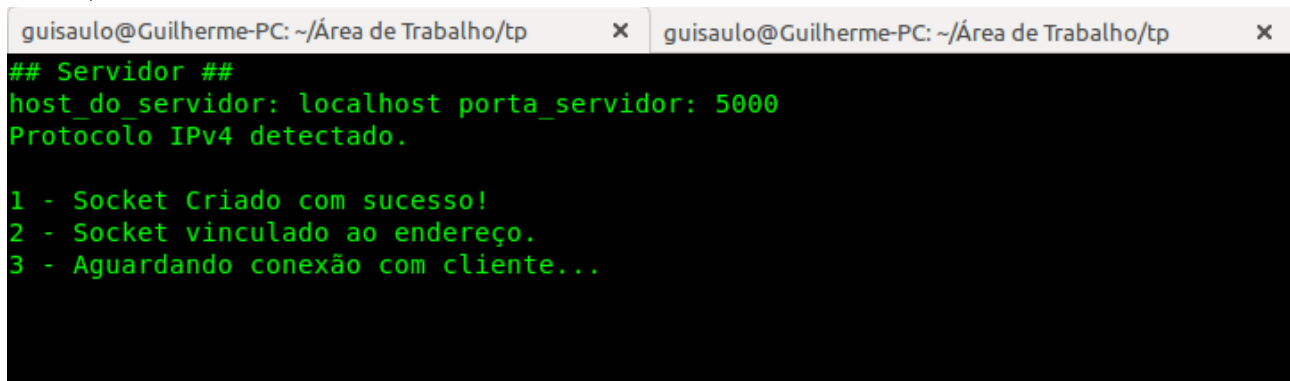
2.2.6. Outras decisões de implementação

Foi implementado no trabalho a confirmação de envio `READY` e `READY ACK`. O cliente só envia a lista de arquivos para o servidor se ele receber o `READY ACK`. O servidor salva a lista de arquivos em um arquivo chamado `<host_do_servidor><nome_diretorio>.txt` em que o nome do diretório respeita as regras do item 2.2.3 deste relatório. Para deixar os testes mais fáceis de serem feitos, o servidor foi modificado para não ficar em loop, aceitando a conexão de apenas um cliente. Isso foi feito pois a especificação do trabalho não determina que o servidor deve encerrar a execução após salvar o arquivo.

2.2.7. Funcionamento do programa

Segue abaixo alguns testes realizados de funcionamento dos protocolos IPv4 e IPv6 e para diferentes diretórios:

```
1 - ./server localhost 5000
```



```
guisaulo@Guilherme-PC: ~/Área de Trabalho/tp x guisaulo@Guilherme-PC: ~/Área de Trabalho/tp x
## Servidor ##
host_do_servidor: localhost porta_servidor: 5000
Protocolo IPv4 detectado.
1 - Socket Criado com sucesso!
2 - Socket vinculado ao endereço.
3 - Aguardando conexão com cliente...
```

2 - ./client localhost 5000 /home/guisaulo

```
guisaulo@Guilherme-PC: ~/Área de Trabalho/tp
## Cliente ##
host_do_servidor: localhost porta_servidor: 5000 nome_diretorio: /home/guisaulo
Protocolo IPv4 detectado.

1 - Socket criado com sucesso.
2 - Conexão com o servidor estabelecida.
READY ACK recebido do servidor.
3 - Enviando lista de arquivos ao servidor...
Fim da transmissão. Enviando 'bye' ao servidor.
4 - Fim Conexão.

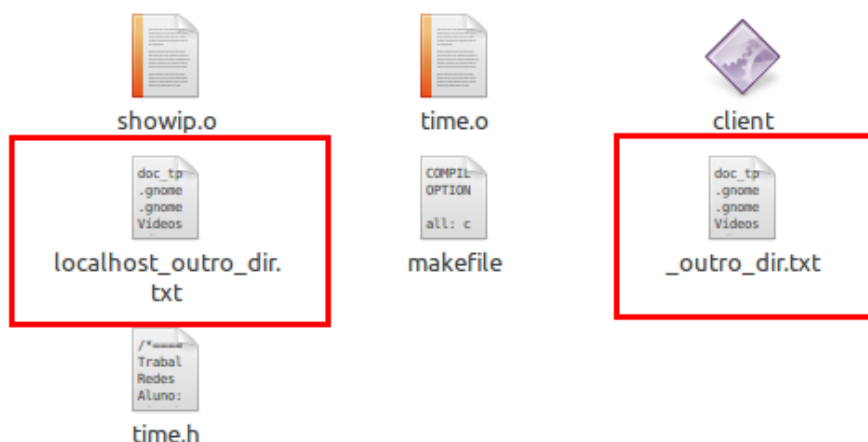
Dados da execução:
400 bytes enviados em 0.002271s
Throughput: 172.01 kbps
guisaulo@Guilherme-PC:~/Área de Trabalho/tp$
```

3 - Servidor é fechado

```
guisaulo@Guilherme-PC: ~/Área de Trabalho/tp
## Servidor ##
host_do_servidor: localhost porta_servidor: 5000
Protocolo IPv4 detectado.

1 - Socket Criado com sucesso!
2 - Socket vinculado ao endereço.
3 - Aguardando conexão com cliente...
4 - Conexão estabelecida.
READY recebido do cliente.
Arquivo localhost_outro_dir.txt_outro_dir.txt criado.
Armazenando lista de arquivos...
Salvando arquivo...
5 - Fim Conexão.
guisaulo@Guilherme-PC:~/Área de Trabalho/tp$
```

4 - Arquivo _outro_dir.txt (lista no diretório do cliente) e localhost_outro_dir.txt (armazenado pelo servidor) é gerado.



5 - ./server ::1 5000 (o mesmo se aplica para protocolo IPv6)

```
guisaulo@Guilherme-PC: ~/Área de Trabalho/tp x guisaulo@Guilherme-PC: ~/Área de Trabalho/tp x
## Servidor ##
host_do_servidor: ::1 porta_servidor: 5000
Protocolo IPv6 detectado.

1 - Socket Criado com sucesso!
2 - Socket vinculado ao endereço.
3 - Aguardando conexão com cliente...
█
```

6 - ./cliente ::1 5000 .(este ponto representa o diretório local)

```
guisaulo@Guilherme-PC: ~/Área de Trabalho/tp x guisaulo@Guilherme-PC: ~/Área de Trabalho/tp x
## Cliente ##
host_do_servidor: ::1 porta_servidor: 5000 nome_diretorio: .
Protocolo IPv6 detectado.

1 - Socket criado com sucesso.
2 - Conexao com o servidor estabelecida.
READY ACK recebido do servidor.
3 - Enviando lista de arquivos ao servidor...
Fim da transmissao. Enviando 'bye' ao servidor.
4 - Fim Conexão.

Dados da execucao:
238 bytes enviados em 0.001681s
Throughput: 138.26 kbps
guisaulo@Guilherme-PC:~/Área de Trabalho/tp$ █
```

3. Resultados

Alguns testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2 Gb de memória. Para esses testes, o programa cliente foi levemente modificado para receber um arquivo .txt em vez de um nome de diretório. Foram criados arquivos de texto de tamanho 2^i bytes, onde $1 \leq i \leq 16$. Abaixo seguem os resultados em uma tabela dos testes realizados no mesmo computador com buffer de 512 bytes.

Tamanho (bytes)	Tempo (s)	Velocidade (kbps)	Bytes enviados
2	0.001710	1.14	2
4	0.001435	2.72	4
8	0.001432	5.46	8
16	0.001658	9.42	16
32	0.001610	19.41	32
64	0.001491	41.92	64
128	0.001911	65.41	128
256	0.001734	144.17	256
512	0.001568	318.89	512
1024	0.000642	1557.52	1024
2048	0.000666	3003.29	2048
4096	0.000694	5763.07	4096
8192	0.000737	10853.38	8192
16384	0.000971	16477.96	16384
32768	0.001556	20564.07	32768
65536	0.001541	41533.38	65536

4. Análise

Ao observar os resultados dos testes acima, verifica-se que a velocidade de transmissão dobra quando o tamanho das mensagens também dobra. Outro fato destacado é que devido os programas estarem na mesma maquina, o tamanho da mensagem não é o que limita o tempo de transmissão, mas sim o processamento necessário para enviar a mensagem. Além disso, percebe-se que a velocidade de transmissão nos testes feitos localmente ficaram bem mais alta que o padrão. Isso acontece porque ambos programas estão na mesma maquina.

5. Conclusão

Com esse trabalho foi possível implementar um par de programas que operem no modelo cliente-servidor e exercitar a comunicação do tipo requisição-resposta sobre o protocolo TCP. Além disso, foi possível praticar a programação com a biblioteca de sockets do Unix para ter uma noção de como funciona o paradigma de comunicação de computação na nuvem. Os resultados obtidos com esse trabalho estava de acordo com o esperado. Todavia, por questão de tempo não foi possível realizar o gráfico e testes com rede sem fio. No mais, o desenvolvimento desse trabalho foi muito importante para consolidar a matéria vista em sala de aula e colocar em pratica a teoria por trás das redes de computadores.

6. Referências

[1] Slides preparados pelos Monitores Erick e Ivan e disponibilizados para os alunos de Redes de Computadores.

[2] Porting Ipv4 applications to Ipv6.

<http://uw714doc.sco.com/en/SDK_netapi/sockC.PortIPv4appIPv6.html> Acessado em 26 de Abril de 2015.

[3] stackoverflow - Tell whether a text string is an IPv6 address or IPv4 address using standard C sockets API <<http://stackoverflow.com/questions/3736335/tell-whether-a-text-string-is-an-ipv6-address-or-ipv4-address-using-standard-c-s>> Acessadp em 26 de Abril de 2015