

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS**  
**CENTRO DE CIÊNCIAS EXATAS, AMBIENTAIS E DE TECNOLOGIA**  
**FACULDADE DE ENGENHARIA DE COMPUTAÇÃO**

BRUNO PEDROSO	12662136
JOSÉ C.	15000000
LUIZ F. Z. MASSON	15000000

**REDES DE COMPUTADORES A**

Atividade 1

Campinas

2018

# SUMÁRIO

## 1 INTRODUÇÃO

## 2 DESENVOLVIMENTO

### 2.1 Servidor

#### 2.1.1 Função sendto():

#### 2.1.2 Adiciona a capacidade de executar comandos shell:

#### 2.1.3 Diretivas do compilador:

### 2.2 Cliente

#### 2.2.1 Função recvfrom():

## 3 RESULTADOS

## 4 ANÁLISE

## 5 CONCLUSÃO

## 1 INTRODUÇÃO

Este experimento tem como objetivo aprender a importância e a necessidade de trabalhar com protocolos, com isso, efetuar a correta comunicação via software entre cliente e servidor, adquirindo base intelectual para atividades correlatas futuramente.

Para obter êxito, requisita-se a elaboração de uma aplicação cliente-servidor que simule um prompt.

A partir dos programas exemplos disponibilizados, houve a necessidade de complementá-los para um correto funcionamento. Usando protocolo UDP que possibilita a comunicação entre dois pontos. Tal protocolo consiste em tratamentos que controlam a transmissão de dados via rede.

Essa atividade nada mais é que a comunicação entre dois lados (cliente UDP e servidor UDP) de uma rede de acesso remoto, no qual o primeiro envia requisições a serem tratadas pelo servidor e aguarda as respostas.

## 2 DESENVOLVIMENTO

Implementando um protocolo UDP (User Datagram Protocol), é um protocolo simplório da camada de transporte. Este permite que seja enviado um pacote Ipv4 ou Ipv6 sem qualquer garantia de entrega, ou seja, pouco confiável.

Neste tipo de comunicação cliente-servidor, o cliente pode enviar um pacote para um servidor e imediatamente enviar outro para um servidor diferente fazendo uso do mesmo socket. O mesmo acontece do lado do servidor, ele pode receber vários pacotes de clientes diferentes usando um mesmo socket. Isso é possível devido ao fato que o UDP não necessita manter relacionamento longo entre cliente e servidor, logo, um serviço sem conexão.

Esta atividade é composta em duas etapas, Servidor UDP e Cliente UDP.

### 2.1 Servidor

Uma aplicação de servidor tem como função principal processar dados oriundos de um cliente e retornar ao mesmo uma resposta de erro ou a informação tratada.

#### 2.1.1 Função *sendto()*;

Esta função é chamada quando o servidor necessita enviar ao cliente uma mensagem, no caso, uma *string*.

```
sendp = sendto(s, buf, BUFF_SIZE, 0, (struct sockaddr *)&client, sizeof client );  
//printf("%i\n", countsp);  
//5o passo: enviar para o cliente  
if(sendp < 0){  
    perror("sendto()");  
    exit(1);  
}
```

Figura 1 - Função *sendto()*.

### 2.1.2 Adiciona a capacidade de executar comandos shell;

Realizou-se uma adaptação que permite a cópia de cada caractere retornado pelo comando, até o tamanho máximo determinado de 2000 caracteres. Isso permite que até mensagens grandes (como por exemplo o retorno de um comando man), sejam retornadas até o limite ser atingido, possibilitando ao usuário um resultado ao menos parcial.

```
void getOutput(char line[]){  
  
    FILE *fp;  
    char content[2000] = {0};  
    char letter;  
    int i;  
  
    /* Open the command for reading. */  
    fp = popen(line, "r");  
    if (fp == NULL) {  
        printf("Failed to run command\n" );  
        exit(1);  
    }  
  
    i = 0;  
    while(i < BUFF_SIZE - 1) {  
        line[i] = fgetc(fp);  
  
        if(line[i] != EOF){  
            if(line[i] == '\\0')  
                line[i] = '0';  
            content[i] = line[i];  
            i++;  
        }  
        else  
            break;  
    }  
  
    /* close */  
    pclose(fp);  
    strcpy(line, content);  
}
```

Figura 2 - Função getOutput.

### 2.1.3 Diretivas do compilador;

Fez-se extremamente necessário o uso de tais diretivas (*string.h* e *unistd.h*), pois é sabido que as mesmas detêm funções fundamentais para a implementação correta da aplicação, tais como, *close()* e ademais relacionadas a manipulação de string.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
```

Figura 3 - Diretivas inclusas.

## 2.2 Cliente

Um *client-side* nada mais é do que uma aplicação que requisita ao servidor um tratamento de dados e com isso aguarda uma resposta adequado do requisitado.

### 2.2.1 Função *recvfrom()*;

Para o lado cliente da aplicação ser capaz de receber alguma resposta, faz-se uso da função *recvfrom()*. Neste caso, uma resposta de até 2000 caracteres.

```
length = sizeof(server);
recv = recvfrom(s, buf, BUFF_SIZE, 0, (struct sockaddr *)&server, &length);
if(recv < 0){
    perror("recvfrom()");
    exit(1);
}
```

Figura 4 - Função *recvfrom()*.

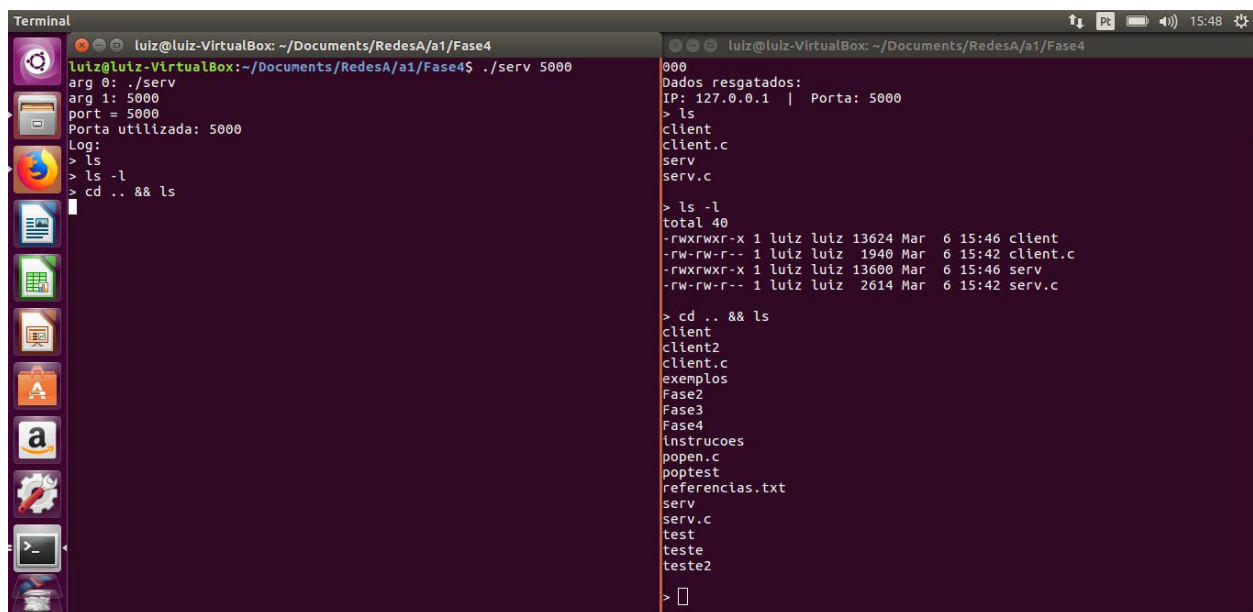
**Nota:** Fez-se necessário atribuir um tratamento de exceção na string enviada, devido ao fato de que o caractere '\0' estava sendo lido como um caractere, consequentemente impedindo o desempenho efetivo da comparação.

```
// Realizando o "trim" do '\0'
lenMessage = strlen(message);
message[lenMessage-1] = 0;
```

Figura 5 - Tratamento de exceção.

### 3 RESULTADOS

Ao final da atividade foi o resultado alcançado foi a inicialização de um servidor UDP em uma porta especificada e um cliente UDP inicializado com um IP e uma porta também especificada, a partir disso foi feito uma troca de dados entre ambos os programas na qual o conteúdo passado primeiramente do cliente para o servidor é uma string contendo um comando prompt, esse comando é então tratado pelo servidor que armazena sua resposta em uma outra string e a envia ao cliente para que o mesmo possa mostrá-la, como exibido nas imagens abaixo com os comandos `ls`, `ls -l`:



```
Terminal
luiz@luiz-VirtualBox: ~/Documents/RedesA/a1/Fase4
luiz@luiz-VirtualBox:~/Documents/RedesA/a1/Fase4$ ./serv 5000
arg 0: ./serv
arg 1: 5000
port = 5000
Porta utilizada: 5000
Log:
> ls
> ls -l
> cd .. && ls

luiz@luiz-VirtualBox: ~/Documents/RedesA/a1/Fase4
000
Dados resgatados:
IP: 127.0.0.1 | Porta: 5000
> ls
client
client.c
serv
serv.c
> ls -l
total 40
-rwxrwxr-x 1 luiz luiz 13624 Mar 6 15:46 client
-rw-rw-r-- 1 luiz luiz 1940 Mar 6 15:42 client.c
-rwxrwxr-x 1 luiz luiz 13600 Mar 6 15:46 serv
-rw-rw-r-- 1 luiz luiz 2614 Mar 6 15:42 serv.c
> cd .. && ls
client
client2
client.c
exemplos
Fase2
Fase3
Fase4
instrucoes
popen.c
poptest
referencias.txt
serv
serv.c
test
teste
teste2
> □
```

Em casos em que a resposta ultrapassou o limite 2000 caracteres o programa copia apenas a quantidade necessária para preencher a string de retorno como mostrado no exemplo abaixo onde foi enviado o comando *man printf*:

```
Terminal
luiz@luiz-VirtualBox: ~/Documents/RedesA/a1/Fase4
luiz@luiz-VirtualBox:~/Documents/RedesA/a1/Fase4$ ./serv 5000
arg 0: ./serv
arg 1: 5000
port = 5000
Porta utilizada: 5000
Log:
> man printf

```

```
\xHH byte with hexadecimal value HH (1 to 2 digits)
\UHHHH Unicode (ISO/IEC 10646) character with hex value HHHH
(4 digits)
\UHHHHHHHH Unicode character with hex value HHHHHHHH (8 digits)
%% a single %
%b ARGUMENT as a string with '\' escapes interpreted,
except that octal escapes are of the form \0 or \0NNN
%q ARGUMENT is printed in a format that can be reused as
shell input, escaping non-printable characters with
the proposed POSIX '$' syntax.
and all C format specifications ending with one of
dIouXfEgGcs, with ARGUMENTS converted to proper type first.
Variable widths are handled.
NOTE: your shell may have its own version of printf, which
usually supersedes the version described here. Please refer
to your shell's documentation for details about the options
it supports.
AUTHOR
Written by David MacKenzie.
REPORTING BUGS
GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
Report prman printf
> 
```



## 4 ANÁLISE

Como pôde ser observado que ao executar um comando que retornasse mais do que 2000 bytes (tamanho dos buffers utilizados nas aplicações), seria necessário a divisão do resultado em pacotes do tamanho desejado, o que resultaria em uma das desvantagens do UDP quando se fala do envio de pacotes sucessivos: nessa situação, os pacotes não necessariamente são recebidos (por conta do tempo de processamento do *client-side*) e, se recebidos, não necessariamente estariam em ordem. Ressaltando que esse comportamento de envio de pacotes sucessivos complementares deve ser implementado na aplicação.

## **5 CONCLUSÃO**

Com a realização dessa atividade, foi possível adquirir conhecimentos necessários para a implementação de um protocolo UDP, que se faz presente nesse experimento, mostrando suas características de funcionamento e limitações, considerando que quando se envia um pacote demasiadamente grande, o mesmo requer uma ajuda "externa" para tratamento do pacote. Em suma o objetivo deste experimento foi alcançado com sucesso.