Compressed Network Communication

Guilherme A. de Araújo Gomes, Jasson M. Fontoura Júnior, Marcos V. Melo da Silva

Departamento de Ciência da Computação – Universidade Federal do Roraima (UFRR)
Boa Vista – RR – Brazil

Guibr.com@gmail.com, juniorjr01@hotmail.com.br, marcoswork77@outlook.com

Abstract. *This article aims to explain both the theoretical and the practical part carried out for the final project of the operating systems discipline, which aimed to produce and implement a CMC (Compressed Network Communication)*

Resumo. *Este artigo tem como finalidade explicar tanto a parte teórica quanta a prática realizada para o projeto final da disciplina de sistemas operacionais, que tinha como objetivo produzir e implementar uma CMC (Compressed Network Communication)*

1. Introdução

O Servidor telnet existe há mais de 40 anos, muito antes de aparecer na Internet. Este sistema de transmissão de dados foi inventado pelas Forças Armadas Americanas para transmissão de dados entre bases militares. Foi disponibilizado ao público em 1977, tendo sido os radioamadores os primeiros a aproveitá-lo. O Telnet é um protocolo que permite emular um terminal à distância, isto é, que permite executar comandos escritos no teclado de um computador remoto.. Com Telnet podemos fazer "login" em outros computadores da Internet e utilizar os seus recursos. Por exemplo, executar aplicações e/ou aceder a serviços existentes nos computadores remotos. O nosso computador passa a ser como um terminal (não inteligente) diretamente ligado ao computador remoto (onde é executado todo o processamento).

2. Comunicação entre Cliente e Servidor TCP

As conexões TCP/IP funcionam de maneira semelhante a uma chamada telefônica, na qual alguém precisa iniciar a conexão discando o telefone. Na outra extremidade da conexão, alguém deve estar ouvindo as chamadas e, em seguida, pegar a linha quando uma chamada for recebida. Nas comunicações TCP/IP, o endereço IP é análogo a um número de telefone e o número da porta seria análogo a um ramal específico assim que a chamada for atendida. O “Cliente” em uma conexão TCP/IP é o computador ou dispositivo que “disca o telefone” e o “Servidor” é o computador que está “escutando” as chamadas. Endereço IP de qualquer servidor ao qual deseja se conectar e também precisa saber o número da porta pela qual deseja enviar e receber dados após a conexão ser estabelecida.

Uma vez que uma conexão através de uma porta TCP/IP tenha sido estabelecida entre um cliente TCP/IP e um servidor TCP/IP, os dados podem ser enviados em qualquer direção exatamente da mesma forma que os dados são enviados através de qualquer outro tipo de porta em um PC (série, paralelo, etc.). A única diferença é que os dados são enviados pela sua rede. A conexão entre um cliente e um servidor permanece aberta

até que o cliente ou o servidor encerre a conexão (ou seja, desligue o telefone). Um benefício extremamente bom do protocolo TCP/IP é que os drivers de baixo nível que implementam o envio e recebimento de dados executa a verificação de erros em todos os dados para que você tenha a garantia de que não haverá erros em nenhum dado enviado ou recebido.

No caso, no nosso projeto, o TCP foi usado para fazer a comunicação entre os nossos códigos de “Cliente” e “Servidor”, e com o auxílio também do Socket, que é usado para troca de informações entre processos na mesma máquina ou em uma rede, distribuem o trabalho para a máquina mais eficiente e permitem o acesso fácil a dados centralizados, pois o nosso “Cliente” e “Servidor” estão sendo executados por via de localhost. O esquema que foi idealizado para o funcionamento desse projeto foi baseado na figura abaixo:

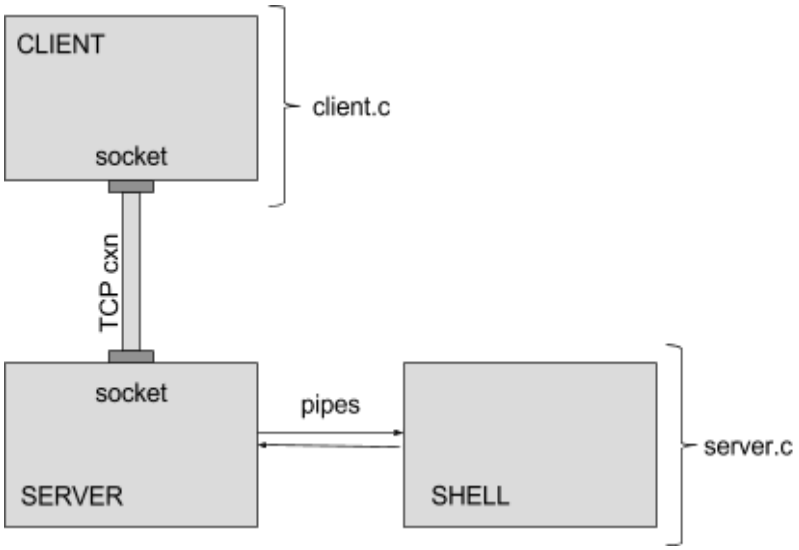


Figure 1. Esquema de Funcionamento

3. Fork

A função fork é uma função que copia o processo atual no sistema operacional. O processo que originalmente chamou a função fork é chamado de processo pai. Um novo processo criado pela função fork é chamado de processo filho. Todas as áreas de um processo são replicadas no sistema operacional (código, dados, pilha, memória dinâmica).

Se a função fork retornar 0 (zero), o processo filho está em execução. Se a função retornar um valor positivo em vez de 0 (zero), o processo pai está em execução. O valor de retorno representa o PID do processo filho criado. A função retorna -1 em caso de erro (provavelmente porque foi atingido o número máximo de processos por usuário configurado no sistema).

O uso do Fork no nosso código do “Servidor” foi para justamente trazer a questão do “Multicliente” para o nosso código. Então, para usar a função fork(), nós incluímos as bibliotecas <sys/types.h> e <unistd.h> . O uso do Fork no nosso código foi feito para que sempre que um novo cliente tentar se conectar ao servidor TCP, criaria um

novo processo filho que seria executado em paralelo com a execução de outros clientes. O uso dele no código, e também de forma geral em outros códigos, pode ser mais ou menos exemplificado por meio dessa figura:

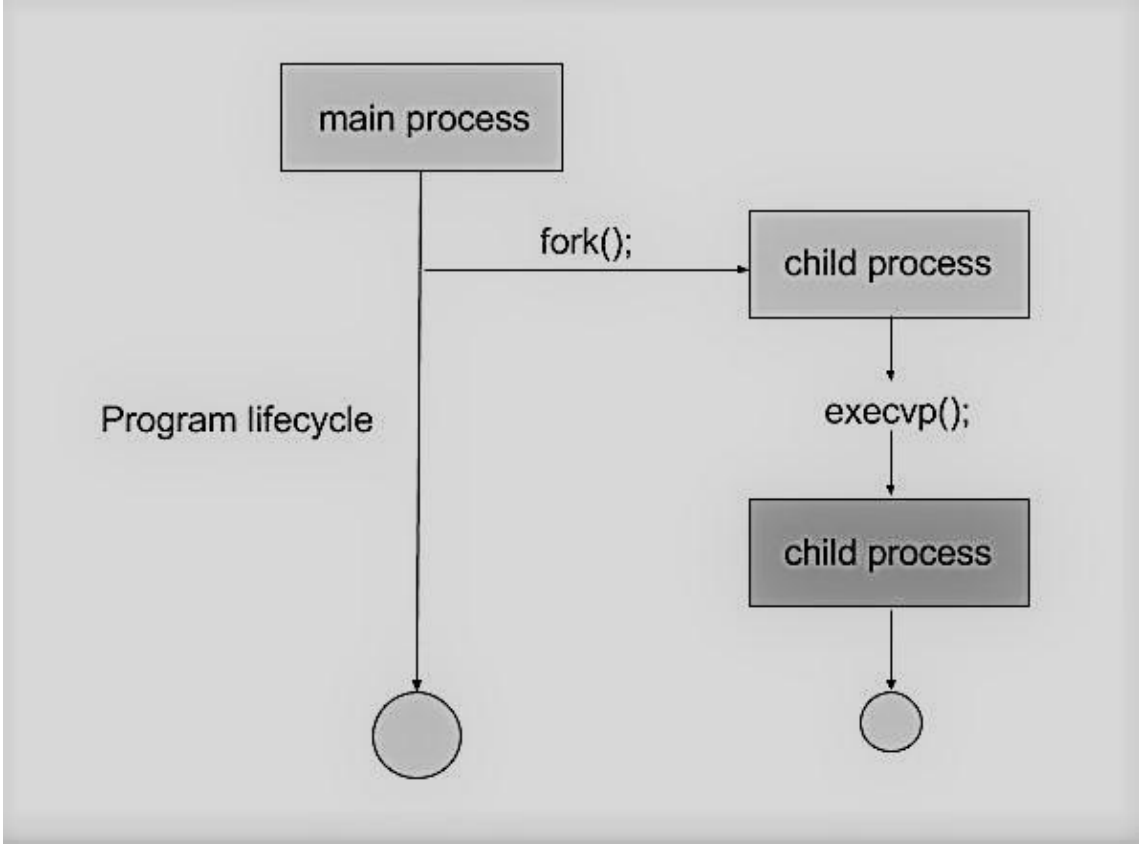


Figure 2. Exemplo do uso do Fork

3.1. Fork e Daemon

Respondendo a pergunta feita pelo nosso professor desse projeto, Os daemons têm várias características, como processos de longa execução, e os daemons podem ser iniciados na inicialização do sistema. O daemon pode ser controlado por comandos do usuário, forçando-o a encerrar, pausar ou até mesmo desabilitar na inicialização. Embora, cenários comuns exigem que um daemon seja encerrado no desligamento do sistema usando algum script específico do sistema.

Existem várias maneiras de criar um daemon e monitorá-lo, mas aqui discutiremos a criação, onde chamamos a função fork para criar um processo filho. O processo pai sai e o processo filho continua a execução porque se torna filho do processo init (em sistemas Linux, o processo init é o primeiro processo na inicialização). O processo filho chama a função setsid para iniciar uma nova sessão e remover o processo do terminal de controle. Por fim, chamamos fork novamente e saímos do processo pai para garantir que nosso daemon não obtenha o endpoint de controle. Agora estamos executando o daemon e é importante registrar um manipulador de sinal sigterm para qualquer limpeza de recurso e saída normal quando o sistema ou usuário fornecer a interrupção correspondente.

4. Desenvolvimento do Projeto

Para o desenvolver os códigos do “Cliente” e “Servidor”, nós usamos algumas “ferramentas”. Durante a criação do código tivemos alguns desafios em fazer com que o “Servidor” mandasse as mensagens para o “Cliente”em forma de confirmação que estaria recebendo essas mensagens, mas então conseguimos achar uma solução na criação de um “Echo Server”, que é um “Servidor” que envia de volta a mesma mensagem que o “Cliente” enviou ao “Servidor”. E como já foi dito antes, queríamos um “Servidor” que conseguisse atender “Multiclientes”, decidimos utilizar a ajuda do Fork.

4.1. Cliente

Aqui no código do “Cliente” foi incluído as bibliotecas necessárias para usarmos algumas funções, como a do Socket por exemplo. Conseguimos com sucesso fazer a criação do Socket no cliente e também fazer a verificação. A configuração com o “Servidor” também foi realizada com sucesso , assim como a verificação da conexão também. Outra coisa que conseguimos definir no nosso código foi a porta de acesso ao servidor.

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

#define PORT 4950

int main(){

    int clientSocket, ret;
    struct sockaddr_in serverAddr;
    char buffer[1024];

// Verifica se o socket foi criado corretamente, caso contrario, imprime o erro
    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if(clientSocket < 0){
        printf("[-] Erro na conexão.\n");
        exit(1);
    }
    printf("[+] Socket do cliente criado!.\n");

// Configuração para estabelecer conexao (endereco)
    memset(&serverAddr, '\0', sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

// Verifica se a conexao foi feita com sucesso, caso contrário, imprime o erro
    ret = connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
    if(ret < 0){
        printf("[-] Erro na conexão.\n");
        exit(1);
    }
    printf("[+] Conectado no servidor.\n");

    // Diálogo entre servidor e cliente
    while(1){
        // Cliente digita a mensagem e envia ao servidor
        printf("Client: \t");
        scanf("%s", &buffer[0]);
        send(clientSocket, buffer, strlen(buffer), 0);
```

Figure 3. Código do Cliente

```c
            // Analisa se foi solicitado encerramento da conexao
            if(strcmp(buffer, ":exit") == 0){
                close(clientSocket);
                printf("[-] Conexão encerrada com o servidor.\n");
                exit(1);
            }

            if(recv(clientSocket, buffer, 1024, 0) < 0){
                printf("[-] Erro no recebimento de dados.\n");
            }else{
                printf("Server: \t%s\n", buffer);
            }
        }

        return 0;
    }
```

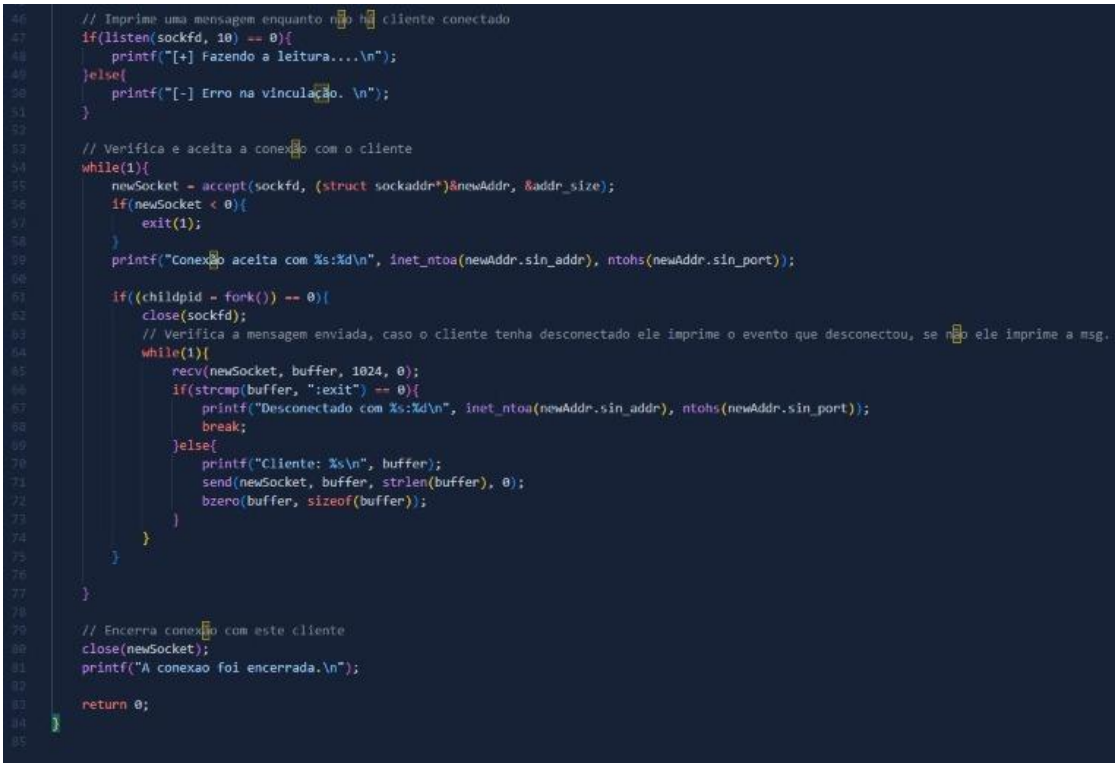
Figure 4. Código do Cliente

4.2. Servidor

Já no código do “Servidor” encontramos alguns impasses que logo foram resolvidos após algumas pesquisas. Dessa forma, conseguimos declarar as variáveis do Socket e Buffer.Definimos e configuramos os parâmetros de endereçamento pro “Cliente” e fizemos também a associação do Socket com o localhost. E como foi dito, usamos o fork para conseguir produzir um “Servidor” que fosse “Multicliente”, fazendo assim cada um ter seu IP de acesso.

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

#define PORT 4950

int main(){
    // Declaracao das variaveis utilizadas no processo
    int sockfd, ret;
     struct sockaddr_in serverAddr;

    int newSocket;
    struct sockaddr_in newAddr;

    socklen_t addr_size;

    char buffer[1024];
    pid_t childpid;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0){
        printf("[-] Erro na conexão.\n");
        exit(1);
    }
    printf("[+] Socket do cliente criado!\n");

    // Definição e configuração de parametros de enderecamento
    memset(&serverAddr, '\0', sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Associa o socket com o endereco local
    ret = bind(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
    if(ret < 0){
        printf("[-] Erro na vinculação. \n");
        exit(1);
    }
    printf("[+] Vinculado a porta %d\n", 4950);
```

Figure 5. Código do Servidor



```
    // Imprime uma mensagem enquanto não há cliente conectado
    if(listen(sockfd, 10) == 0){
        printf("[+] Fazendo a leitura....\n");
    }else{
        printf("[-] Erro na vinculação. \n");
    }

    // Verifica e aceita a conexão com o cliente
    while(1){
        newSocket = accept(sockfd, (struct sockaddr*)&newAddr, &addr_size);
        if(newSocket < 0){
            exit(1);
        }
        printf("Conexão aceita com %s:%d\n", inet_ntoa(newAddr.sin_addr), ntohs(newAddr.sin_port));

        if((childpid = fork()) == 0){
            close(sockfd);
            // Verifica a mensagem enviada, caso o cliente tenha desconectado ele imprime o evento que desconectou, se não ele imprime a msg.
            while(1){
                recv(newSocket, buffer, 1024, 0);
                if(strcmp(buffer, ":exit") == 0){
                    printf("Desconectado com %s:%d\n", inet_ntoa(newAddr.sin_addr), ntohs(newAddr.sin_port));
                    break;
                }else{
                    printf("Cliente: %s\n", buffer);
                    send(newSocket, buffer, strlen(buffer), 0);
                    bzero(buffer, sizeof(buffer));
                }
            }
        }

    }

    // Encerra conexão com este cliente
    close(newSocket);
    printf("A conexao foi encerrada.\n");

    return 0;
}
```

Figure 6. Código do Servidor

5. Problemas e Faltas no Projeto

Infelizmente, nosso projeto não conseguiu atender todos os pedidos que foram feitos pelo nosso professor. Tivemos alguns problemas na implementação de algumas funções e de alguns comandos que seriam feitos pelo nosso “Cliente” ao “Servidor”. Aqui iremos falar sobre cada parte do projeto que não foi atendida e os problemas enfrentados durante a tentativa de implementação, e também iremos falar como faríamos para tentar implementá-las.

5.1. –Log, –Host e –Compress

Aqui teríamos que colocar alguns comandos no “Cliente”, que seriam respondidos pelo “Servidor”. Esses comandos seriam o –log, –host e –compress. O –log seria para criar um arquivo que armazena as informações que ocorreram durante a comunicação do “Servidor” ao “Cliente” e vice-versa, o –host seria para alterar o host da conexão e o –compress serviria para o fazer a compressão e descompressão. A forma que tínhamos achado para tentar implementar a questão do –log e –host mas principalmente o –log seria seguindo a ideia de que, por meio do ARGC e ARGV no “Servidor”, poderíamos criar um FOR para cada elemento que eles lerem e assim ver se algum desse elemento, ou desses elementos, pertence aos comandos e assim ele conseguiria responder o comando solicitado pelo “Cliente”, e o problema que enfrentamos foi justamente na criação desse arquivo pro log pois ele ou não armazenava ou armazena de forma errada os dados. Já o –compress não foi possível a implementação pois não conseguimos fazer a compreensão da comunicação do “Cliente” pro “Servidor”.

5.2. Pipe

O pipe seria a forma de comunicação que foi proposta a ser feita entre o “Servidor” e o Shell, como mostra a Figure 1, mas por falta de tempo e na tentativa de fazer as outras coisas funcionarem nós acabamos não fazendo essa comunicação. Mas sabemos como se pode fazer essa conversa, seria pela chamada de um fork() e assim poderia ser feita a chamada de duas mains. Na primeira chamada seria para escrever o pipe e na segunda chamada seria para ler o pipe, e assim por meio do execl() a ligação entre o “Servidor” e o Shell aconteceria.

5.3. Compressão

A compressão seria o ato de reduzir o espaço ocupado por dados num determinado dispositivo. Essa operação é realizada através de diversos algoritmos de compressão, reduzindo a quantidade de Bytes para representar um dado, sendo esse dado uma imagem, um texto, ou um arquivo (ficheiro) qualquer. E ela iria funcionar, no projeto, como uma forma de diminuir os pacotes usados entre a comunicação do “Cliente” e o “Servidor”. Essa parte deveria ter sido feita pela biblioteca Zlib que, por meio das funções compress() e uncompress(), seria feito a compressão e a descompreensão do buffer enviado para elas, e isso não foi feito por falta de conhecimento da nossa parte.

6. Conclusão

Este trabalho apresentou o projeto e implementação do servidor telnet usando o protocolo TCP, sockets, fork e sendo ele para multi-clientes . O processo de criação foi difícil e complicado de início, mas logo algum desses problemas foram enfrentados e vencidos com muito esforço, porém outros não e vamos buscar aprender mais sobre eles e resolvê-los com o tempo. Vale ressaltar que, o servidor consegue executar a comunicação básica com o cliente.
Apesar de tudo, foi um processo interessante e inovador e com muito aprendizado. Nós, Guilherme, Jasson e Marcos, aprendemos muitas coisas durante a criação do projeto e com certeza serão informações muito valiosas no futuro.

Referências

COOLER_. [Portuguese] Sockets em linguagem C. ExploitDatabase, 29 de julho de 2022. Disponível em: <https://www.exploit-db.com/papers/13634>. Acesso em: 19 de julho de 2022.

SILVA, Cristiano. fork, exec, e daemon. Embarcados, 13 de abril de 2021. Disponível em: <https://embarcados.com.br/fork-exec-e-daemon/#fork>. Acesso em: 22 de julho de 2022.

SILVA, Cristiano, Biblioteca de Socket TCP. Embarcados, 21 de maio de 2021..Disponível em: <https://embarcados.com.br/socket-tcp/#Biblioteca> . Acesso em: 20 jul. 2022.

GEEKSFORGEEKS. Manipulando vários clientes no servidor com multithreading usando programação de soquete em C/C++, 20 de julho de 2022. Disponível em: <https://www.geeksforgeeks.org/handling-multiple-clients-on-server-with-multithreading-using-socket-programming-in-c-cpp/>. Acesso em: 24 de julho de 2022.

INGALLS, Robert. Sockets Tutoria, 14 de setembro de 1998. Disponível em: <http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html>. Acesso em: 20 de julho de 2022.

IDIOT DEVELOPER. TCP Client Server Implementation in C | Socket Programming in C. Youtube, 25 de abril de 2021. Disponível em: <https://www.youtube.com/watch?v=io2G2yW1Qk8>. Acesso em: 19 de julho de 2022.