Trabalho da disciplina de Redes I (CI1058) - 2021

Allan Cedric G. B. Alves da Silva[1] **- GRR20190351**

[1]Departamento de Informática - Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 - 81.531-980 - Curitiba - PR - Brasil

acgbas19@inf.ufpr.br

1. Introdução

O foco deste trabalho foi implementar uma rede de comunicação básica entre uma aplicação cliente e uma aplicação servidor através de conexões *Raw Sockets*. A fim de aplicar os conhecimentos relacionados a camada de enlace vistos na disciplina.

O presente documento discorrerá sobre as decisões de projeto para implementação de um editor C remoto simplificado, onde temos uma aplicação cliente que atua como interface das interações de um usuário, e uma aplicação servidor que possui os mecanismos para tratar destas interações em conjunto com a aplicação cliente. A saber, foi utilizada a liguagem de programação C para o desenvolvimento do trabalho.

Para detalhes de compilação e execução do sistema, confira o `README` adjunto ao trabalho.

2. Conexões *Raw Sockets*

Foi desenvolvido uma biblioteca que encapsula a criação e as operações de envio e recebimento de dados de um *Raw Socket*.

2.1. `raw_socket.h`

Compreende os protótipos dos procedimentos de criação (`rawsocket_connection`), envio de dados (`sendto_rawsocket`) e recebimento de dados (`recvfrom_rawsocket`).

2.2. `raw_socket.c`

A implementação do procedimento `rawsocket_connection` foi inspirada na função fornecida pelo professor. Ademais, foi adicionado dentro desse procedimento, uma função para tornar as comunicações no *Raw Socket* como não-bloqueantes (`fcntl`).

As funções `sendto_rawsocket` e `recvfrom_rawsocket` atuam como *wrappers* das funções de sistema `send` e `recv`, respectivamente.

3. Protocolo de comunicação

Foi utilizado o protocolo *Kermit* simplificado com controle de fluxo *Stop-and-Wait*. Além disso, foi desenvolvido uma biblioteca de funções genéricas para criação, impressão formatada, validação básica e detecção de erros de pacotes *Kermit*.

3.1. `kermit.h`

Compreende um conjunto de macros com ênfase nos códigos do protocolo *Kermit* para os pacotes. Essas macros definem os marcadores iniciais, valores de endereçamento, tipo de mensagem, códigos de erro, etc.

Além disso, temos a estrutura de um pacote *Kermit* (`struct kermit_pckt_t`), a qual especifica o corpo de um pacote. E para completar, os protótipos dos procedimentos para criação (`gen_kermit_pckt`), impressão formatada (`print_kermit_pckt`), validação básica (`valid_kermit_pckt`) e detecção de erros (`error_detection`).

3.2. `kermit.c`

Alguns procedimentos, como o `valid_kermit_pckt` e o `error_detection`, merecem um pouco de atenção na sua implementação.

No caso do procedimento `valid_kermit_pckt`, é feita uma verificação básica para saber se um pacote *Kermit* possui o marcador inicial correto e se o escopo dos endereçamentos de origem e destino tange apenas o endereço da aplicação servidor ou da aplicação cliente.

Ao passo que no procedimento `error_detection`, é calculado a paridade vertical do pacote através de um *xor* byte a byte dos campos tamanho, sequência, tipo e dados. Após isso, é realizado um *xor* com o campo paridade, o qual é gerado no procedimento `gen_kermit_pckt`, e se o resultado der 0 quer dizer que o pacote está íntegro e correto.

3.3. Comando `compilar`

Será descrito brevemente o comportamento do comando `compilar` no que tange o protocolo de comunicação. A seguir um diagrama explicando o processo de comunicação:

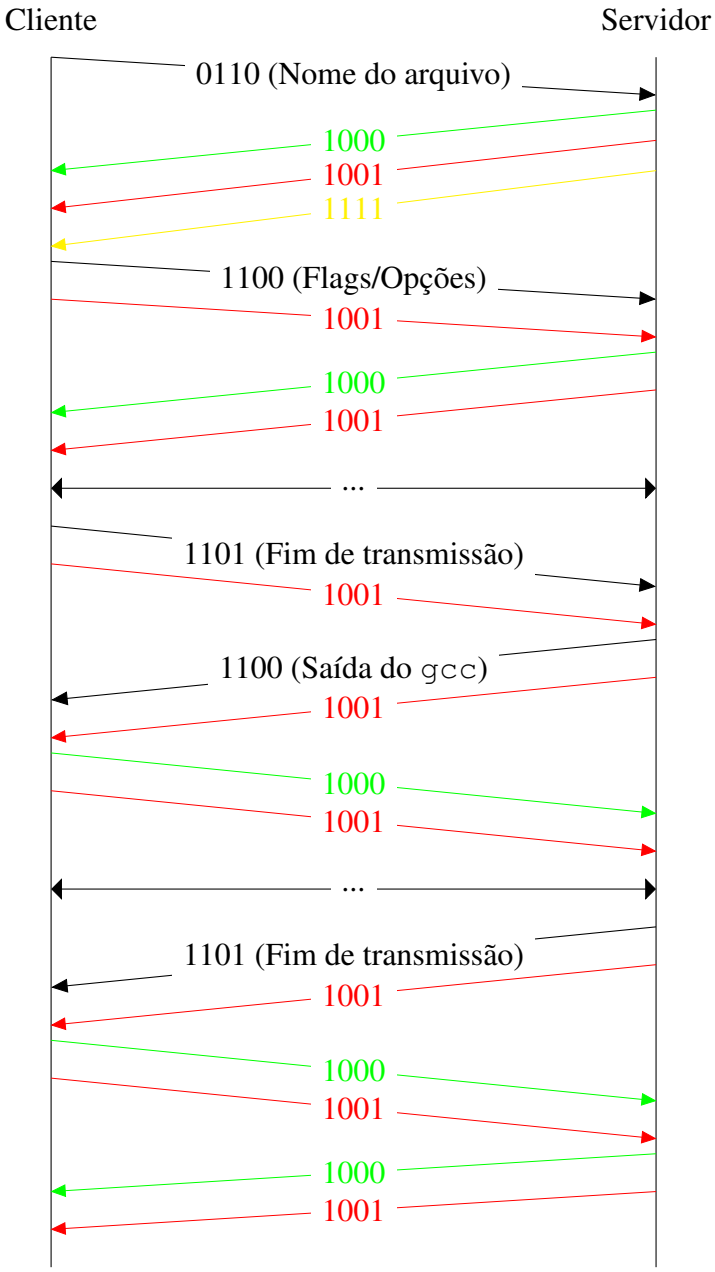

Figura 1. Modelo de comunicação para o comando `compilar`

No trabalho como um todo, sempre finalizaremos uma troca de mensagens entre cliente e servidor com uma mensagem `ACK` para o cliente, como mostra o diagrama acima.

4. Aplicação cliente

Essa aplicação é responsável pela interface que recebe as interações de um usuário. O fluxo de ação básico da aplicação cliente é o seguinte:

1. Entrada de dados do usuário;
2. Interpretação do comando fornecido;
3. Execução de comandos *standalone* ou comunicação com servidor para realizar uma certa operação;

4. Retorna ao item 1;

A partir disso, foi desenvolvido uma biblioteca para a aplicação cliente, a qual se utiliza das bibliotecas citadas anteriormente, para implementar os mecanismos do fluxo de ação acima, além do programa principal que roda o cliente.

4.1. `client.h`

Conjunto de protótipos de procedimentos para inicialização do cliente (`client_init`), entrada de dados (`read_client_input`), execução de comandos *standalone* (`client_standalone_commands`), geração do pacote *Kermit* inicial de um comando (`client_command_kermit_pckt`), envio de dados ao servidor (`send_kpckt_to_server`), recepção de dados advindos do servidor (`recv_kpckt_from_server`), decodificação de pacotes recebidos do servidor (`client_kpckt_handler`), entre outros procedimentos auxiliares.

4.2. `client.c`

Agora serão discutidos alguns detalhes de implementação de alguns dos procedimentos principais.

4.2.1. `read_client_input`

Aguarda uma entrada de dados do usuário. Caso o comando fornecido ou os seus argumentos não forem válidos, o procedimento informa o erro e requisita uma nova entrada. Se o comando e seus argumentos forem lidos com sucesso, ele retorna a codificação inteira para aquele comando como dito na especificação do trabalho.

4.2.2. `client_standalone_commands`

Executa os comandos locais da aplicação cliente, no caso são dois: `lcd` e `lls`. Neste caso, não há necessidade de se comunicar com o servidor, e assim esses comandos rodam diretamente no programa cliente.

O comando `lcd` é realizado a partir do procedimento `chdir` que faz parte da API do *POSIX*, disponível pelo arquivo `unistd.h`.

Ao passo que o comando `lls`, é executado a partir da chamada `popen` que é um procedimento advindo do arquivo `stdio.h`, que serve para criar um *pipe* que descarrega a saída de comandos da *shell* em um descritor de arquivos. Vale ressaltar que o procedimento `popen` foi muito importante para a execução dos outros comandos especificados no trabalho.

4.2.3. `client_command_kermit_pckt`

O cliente possui a iniciativa com a comunicação com o servidor, e por essa razão que existe um procedimento para criar o pacote *Kermit* inicial a partir do comando inserido pelo usuário.

4.2.4. `recv_kpckt_from_server`

Aguarda uma resposta do servidor, além disso como está sendo utilizado comunicações *Raw Sockets* com a interface *loopback*, é necessário filtrar adequadamente os pacotes que chegam ao *socket* no que tange marcadores iniciais e endereçamentos de origem e destino. Caso seja um pacote inválido, ele será ignorado.

Ademais, aqui é onde foi implementado a lógica de *timeout*. Caso o cliente não receba uma resposta do servidor em até 2000ms (2s), será reenviado o pacote por parte do cliente. E caso ocorra 5 *timeouts* consecutivos, o cliente aborta os reenvios e aguarda uma nova entrada do usuário.

Para simulação de *timeout* na aplicação cliente, foi controlado a recepção de pacotes com uma variável que recebe valores aleatórios, a depender do valor aleatório gerado, o cliente ignora as respostas do servidor, a fim de aplicar um *timeout*.

4.2.5. `client_kpckt_handler`

Cuida de toda interação entre o cliente e o servidor. Esse procedimento atua como uma máquina de estados do cliente com base nas respostas vindas do servidor. A partir disso, ele decodifica os pacotes do servidor, realiza a operação necessária na aplicação cliente, e gera ou não novos pacotes para serem enviados ao servidor.

4.3. `main_client.c`

Neste arquivo está implementado o fluxo de ação básico da aplicação cliente, como foi descrito anteriormente. De forma pormenorizada temos o seguinte:

1. Inicialização do cliente (Conexão *Raw Socket*);
2. Entrada de dados do usuário e interpretação do comando fornecido;
3. Execução de comandos *standalone*. Caso seja um comando *standalone*, executa e volta ao item 1;
4. Geração do pacote *Kermit* inicial do comando;
5. Envia um pacote *Kermit*;
6. Espera uma resposta do servidor;
7. Tratamento da resposta. Caso precise enviar uma resposta ao servidor, volta ao item 5, senão volta ao item 2;

5. Aplicação servidor

Essa aplicação é responsável pelos mecanismos que realizam a grande parte das operações solicitadas pela aplicação cliente, com exceção das operações *standalone*. O fluxo de ação básico da aplicação servidor é o seguinte:

1. Aguarda um pedido do cliente;
2. Recebe um pacote;
3. Executa uma operação solicitada e realiza comunicação com o cliente;
4. Retorna ao item 1;

Como feito com a aplicação cliente, foi desenvolvido uma biblioteca para implementar os mecanismos do fluxo de ação acima, além do programa principal que roda o servidor.

5.1. `server.h`

Compreende o conjunto de protótipos de procedimentos para inicialização do servidor (`server_init`), espera de um pacote de um cliente (`wait_kpckt_from_client`), tratamento de pacotes e comunicação com o cliente (`server_kpckt_handler`), entre outros procedimentos auxiliares.

5.2. `server.c`

O procedimento principal desse arquivo fonte é o procedimento `server_kpckt_handler`, o qual é responsável por tratar os pacotes advindos do cliente e as devidas respostas. Cada operação solicitada pelo cliente, vai configurar uma máquina de estados que o servidor vai trabalhar para realizar a operação desejada.

5.3. `server_handler.c`

Neste arquivo é onde temos a implementação das máquinas de estado que são utilizadas no procedimento `server_kpckt_handler`. A título de exemplo, temos o procedimento `ls_state` para tratar do comando `ls`, o `ver_state` para tratar o comando `ver`, e assim por diante.

Cada procedimento que atua como máquina de estados, se utiliza de um procedimento em comum, o `cmd_state`, o qual é responsável pela lógica de comunicação com a aplicação cliente. Além disso, esse procedimento recebe uma função de tratamento (*handler*) a depender da operação realizada.

De forma bijetiva, cada máquina de estados está associada a um procedimento tratador, o qual em suma realiza a operação pedida pela aplicação cliente, e gera os pacotes necessários para se comunicar com o cliente. Todos os comandos, com exceção do `cd`, utilizam o procedimento `popen` citado anteriormente.

Como exemplo, temos o procedimento `linha_state`, o qual encapsula a chamada do procedimento `cmd_state` que recebe como argumento o procedimento tratador `linha_handler`.

5.4. `main_server.c`

Aqui temos o fluxo de ação básico da aplicação servidor dito anteriormente. De forma pormenorizada temos o seguinte:

1. Aguarda um pedido de um cliente (Não possui *timeout*, pois o servidor está esperando algo a fazer);
2. Recebe o pacote do cliente;
3. Executa a máquina de estados do comando pedido;
4. Retorna ao item 1;