Aluno: André Miguel Sikora Marques GRR20182593
Professor: Luiz Carlos Pessoa Albini
Relatório Trabalho Prático - Editor de Texto Remoto

pacote* criarPacote (int origem, int destino, int tamanho, int tipo, unsigned char *dados);
Essa função organiza os dados do pacote em uma só estrutura.
unsigned char* colocarBuffer (pacote* p);
Essa função é usada para colocar as informações do pacote em um buffer. Posteriormente Esse buffer é usado no envio. Como destino, origem, tamanho, sequência, e tipo são menores que 1 byte, é preciso fazer esses shift's e or's antes de colocá-los no buffer.
pacote* tirarBuffer(unsigned char *buffer);
Essa função faz o contrário de colocarBuffer, ou seja, tira as informações do buffer e coloca no pacote.
int enviar(int socket, pacote *p, int server);
Essa função envia um pacote.
pacote* receber(int socket, int server);
Essa função recebe um pacote. Ao receber um ACK, a sequência é incrementada, e ao chegar em 15, o valor máximo, volta para 0.
comando* lerComando();
Essa função lê o input do usuário e organiza essa informação em uma estrutura.
void cd(int socket, comando *cmd);
Executa o cd, saindo do cliente. A primeira parte verifica se o tamanho do nome do diretório de destino é maior que o tamanho máximo do pacote (15 bytes). Se for, quebra em pedaços menores para enviar. Após isso, a função recebe um pacote do servidor. Se for nulo o cd foi executado corretamente. Se não, o pacote traz uma mensagem com o erro ocorrido.
void serverCd(int socket, unsigned char* diretorio);
Essa função executa o cd no server. Se o pacote for menor que 15 bytes, o nome do diretório já vem de server.c, e o pacote recebido dentro desta função é do tipo FIM, então a função não precisa tentar receber mais partes do nome. Após isso o server efetua o cd com o nome recebido, enviando de volta um erro se o diretório não existir.
void lcd(comando *cmd);
Efetua o cd no client.
void ls(int socket, comando *cmd);
Envia ao server um pedido de ls. Depois recebe o(s) pacote(s) com a informação do que está contido no diretório do server.
void serverLs(int socket, unsigned char* dados);
Executa a função listar(), que retorna o conteúdo do diretório em um char*. Após isso, se for necessário, quebra em partes menores antes de enviar.
void lls(comando *cmd);
Imprime o retorno da função listar().

void ver(int socket, comando *cmd);
Envia um pedido de ver ao servidor, quebrando o nome do arquivo em menores partes se preciso, e recebe o conteúdo do arquivo requisitado. Imprime erro caso o arquivo não exista.
void serverVer(int socket, unsigned char* arquivo);
Recebe o pedido de ver, recebendo vários pacotes caso o nome do arquivo seja maior que 15 bytes. Em seguida verifica se o arquivo existe, enviando um pacote de erro de volta caso necessário. Em seguida usa a função lerArquivo para colocar o conteúdo em um char*. Após isso quebra o Conteúdo do arquivo em partes menores para enviar ao client.
void linha(int socket, comando *cmd);
Envia o pedido do comando linha ao server. Primeiro o nome do arquivo é enviado, sendo quebrado em partes menores caso necessário. Após isso o número da linha é enviado, sendo também quebrado em partes menores, embora isso dificilmente ocorra. Em seguida verifica se foi recebido pacote de erro, sendo possível que a linha e/ou o arquivo não existam. Caso não ocorram erros, recebe os pacotes da linha pedida.
void serverLinha(int socket, unsigned char* arquivo);
Executa o comando linha no servidor. Primeiro recebe o nome do arquivo, depois o número da linha requisitada. Em seguida coloca o conteúdo do arquivo em um char* através de lerArquivo(). Se o arquivo não existir envia um pacote de erro. Em seguida o número de linhas do arquivo é calculado. Se a linha requisitada não existir envia pacote de erro. Após isso busca pela linha requisitada, copia o conteúdo desta linha para outro char*, e envia o(s) pacote(s) de volta para o client.
void linhas(int socket, comando *cmd);
Envia o pedido de linhas para o servidor. Primeiro envia o nome do arquivo, depois a linha inicial, e depois a linha final. Em seguida verifica se foi recebido pacote de erro. Caso o arquivo exista e a linha inicial seja válida, recebe as linhas requisitadas.
void serverLinhas(int socket, unsigned char* arquivo);
Executa o pedido de linhas no servidor. Primeiro recebe o nome do arquivo, em seguida a linha inicial e depois a linha final. Em seguida verifica se o arquivo existe e se a linha inicial é válida, enviando erro caso contrário. Após isso busca o conteúdo pedido, e envia ao client.
void edit(int socket, comando *cmd);
Envia o pedido de edit ao servidor. Primeiro envia o nome, depois o número da linha, e por último o novo texto. Após isso a função recebe um novo pacote para verificar se ocorreu algum erro. Essa função tem um bug estranho, em que após editar uma linha de um arquivo, se for usado o comando ls, será impresso em tela o arquivo que foi editado, além do ls.
void serverEdit (int socket, unsigned char* arquivo);
Executa o edit no arquivo pedido. Recebe primeiro o nome, depois a linha, por último o texto. Em seguida verifica possíveis erros se o arquivo ou a linha não existirem. Em seguida é feita a concatenação do que vem antes da linha a ser editada, a nova linha, e o que vem depois da nova linha, e por fim a função

reescreve o arquivo. Essa função tem o bug mais inexplicável de todos: se for recebido um texto de 25 caracteres (incluindo aspas), a última aspa não é removida, ao contrário de outros tamanhos de texto. É preciso um if específico pra isso.

unsigned char* listar();

Armazena o conteúdo do diretório em um char*.

unsigned char* lerArquivo(unsigned char* nome);

Armazena o conteúdo de um arquivo em um char*.

O trabalho foi feito tentando ficar o mais próximo possível das especificações exigidas.

No geral, tudo funciona, embora seja possível que alguns problemas ocorram se o terminal está há muito tempo aberto.

Sobre o arquivo **ConexaoRawSocket.c** disponibilizado, só foi possível utilizá-lo passando **"lo"** como argumento, **"127.0.0.1"** retornava erro em ioctl.

Bugs, problemas e erros:

O primeiro erro foi declarar **dados** como **char*** ao invés de **void*** no **struct pacote**. Isso ocasionou um pequeno problema na implementação de **linhas()**, pois não foi possível enviar a linha inicial e a linha final como dois **int**s em um único pacote. As linhas foram então enviadas através de dois pacotes como **char*** com o uso de **atoi()** antes de serem usadas em **serverLinhas()**. Isso não causa nenhum problema na execução, mas desvia um pouco do protocolo pedido.

No server, eu tentei fazer algo parecido com o client onde era impresso em tela o diretório atual após a execução de cada comando. Porém por motivos desconhecidos o server imprime com delay de um comando o diretório, ou ocasionalmente imprime várias vezes. Porém isso não parece afetar a execução dos comandos.

Durante a execução de comandos que podem trazer grandes volumes de dados, como **ver** e **linhas**, dependendo do tamanho do arquivo o cliente pode se perder no meio da execução e parar de imprimir os dados que estava recebendo. Ao parar de imprimir, o client pode enviar outro comando, mas se for um comando que imprima informações vindas do server, pode ser que ele não imprima nada, ou não imprima o que devia, neste caso imprimindo o que ele parou de imprimir antes. Por exemplo, se o comando **ls** for digitado após **ver**, é possível que durante a execução de **ls** sejam impressas partes de **ver**, ou que o client pare de imprimir. Isso tende a ocorrer quando o terminal está aberto a um certo tempo, ou seja, se **ver** é executado logo após abrir o terminal e executar o client isto não ocorre.

Também não consegui implementar nack.