

Redes de Computadores - Relatório TP1

Aluno: Rodrigo Ferreira Araújo

Matrícula: 2020006990

Soluções Adotadas:

Para este TP, inicialmente, a parte de configuração e inicialização das conexões (IPv4 e IPv6) entre servidor e cliente foi inspirada nas implementações vistas na playlist de vídeos no youtube do Ítalo Cunha disponibilizada na especificação. Esses conteúdos incluem mensagens de erro caso haja passagem incorreta de parâmetros, inicialização do endereço IPv4 ou IPv6 do servidor, adaptação dos endereços para strings e inicialização/uso dos sockets e buffers de troca de mensagens nos *main* de cada arquivo (server.c e client.c, não foi adotado um arquivo header common.h para esta entrega)

Inicialmente, a partir da perspectiva do cliente, criamos uma string na stack chamada *buffer*, de tamanho máximo de 500 bytes, que será responsável pela comunicação *Plain Text* com o servidor. Assumindo que o servidor já esteja inicializado, então, entramos em um loop infinito (while(1)) que representa o funcionamento corrente do cliente até que uma condição de saída aconteça. Note que duas mensagens de log são impressas no terminal do servidor, uma indicando que o servidor foi inicializado e está esperando conexões no *accept*, e outra confirmando do sucesso da conexão de um cliente, que também é impressa no terminal do cliente, conforme a figura abaixo:

```
araujo@LAPTOP-TRRBOCNO:/mnt/c/Users/arauf/Documents/UFMG/2023-1/Redes/TPs/TP1$ ./client ::1 51511
Connected to IPv6 ::1 51511

araujo@LAPTOP-TRRBOCNO:/mnt/c/Users/arauf/Documents/UFMG/2023-1/Redes/TPs/TP1/serv$ ./server v6 51511
[log] Bound to IPv6 :: 51511, waiting connections
[log] Waiting for new client
[log] connected from IPv6 ::1 47850
```

O servidor possui dois loops principais, aquele que espera por uma conexão de um cliente e outro que trata aquele cliente recebido. O servidor fecha a conexão do cliente após ocorrer uma condição que exija isso, por exemplo, o cliente digita um comando inválido.

Nesse sentido, após a inicialização do cliente, esperamos o input do usuário através do *fgets* de C, escrevendo a string da entrada em *buffer* e, então, começamos a validação da entrada:

1. Caso os primeiros 12 caracteres do buffer sejam exatamente "select file "
 - a. Lemos o buffer com o *strtok* de C até encontrar o nome do arquivo, usando "<espaço>" como *token*. Note que, comandos do tipo "select filearquivo.txt", "select file" ou "select file teste.java*" (onde teste.java é um arquivo válido e * é qualquer sequência de caracteres) não passam na validação do comando e será tratado como comando inválido. Um tratamento particular feito aqui foi especificamente se o buffer seria exatamente "select file ", daí consideramos que o usuário não selecionou nenhum arquivo e printamos no terminal do cliente "no file selected!\n" e espera um próximo comando. Note que, caso o comando seja "select file<espaço><tab>", o cliente tenta acessar um arquivo de nome <tab> e, claro, falha.
 - b. Por meio da função *access* de C, verificamos se o arquivo está no mesmo diretório em que o executável do cliente está. Caso positivo, continuamos, caso contrário, imprimimos no terminal do cliente a mensagem especificada: "file <nome> does not exist\n" e espera o próximo comando.
 - c. Verificada a existência do arquivo, avaliamos sua extensão. Nesse sentido, comparamos a extensão do arquivo em si com cada entrada de um array de strings que contém as extensões válidas. Caso uma comparação seja bem sucedida, a extensão é válida e o arquivo foi selecionado com sucesso, caso contrário, a extensão é inválida (ou o arquivo não possui extensão) e o terminal do cliente imprime "file <nome> is not valid!\n" e espera o próximo comando.
2. Caso o buffer seja "send file\n"

- a. Caso nenhum arquivo tenha sido selecionado ainda, o terminal do cliente imprime “no file selected!\n” e espera o próximo comando.
 - b. Caso contrário, o cliente monta a mensagem de envio da forma “<nomearquivo><conteudodoarquivo>\lend”, em no máximo 500 bytes. Então, caso o cliente tenha uma mensagem de arquivo montada, ela é prontamente enviada para o servidor. Note que, uma vez enviado um arquivo, ele continua selecionado, de modo que um novo “send file” mandaria o mesmo arquivo novamente até que um novo arquivo válido seja selecionado, isto é, ele existe no diretório do cliente e sua extensão é válida.
 - c. Note que, caso o usuário apague o arquivo entre o select file e o send file, o servidor apenas imprime “<nomearquivo> does not exist\n”, **deseleciona o arquivo anterior**, e espera um novo comando.
3. Caso buffer seja “exit!\n”.
- a. “exit!\lend” é enviado para o servidor, que é processado em separado para desconectar ambos cliente e servidor, terminando suas execuções e imprime “connection closed” (string enviada do servidor para o cliente é “connection closed!\lend”) em seus terminais.

Caso nenhuma dessas validações sejam verdadeiras, o buffer contém um comando inválido. Nesse contexto, enviamos “invalid command!\lend” para o servidor, que processa especificamente essa entrada e prontamente responde “disconnect” para o cliente, que imprime no seu terminal: “disconnected due to incorrect command”, de modo que essa troca de mensagem específica desconecta apenas o cliente atual e o servidor continua funcionando, no aguardo de um novo cliente, como mostrado abaixo.



```

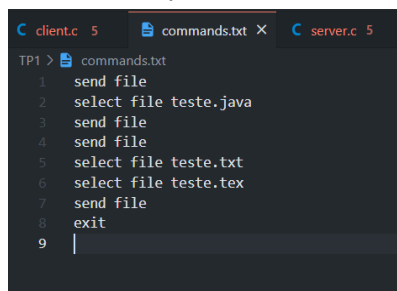
araujo@LAPTOP-TRRBOCNO: /mnt/c/Users/arauj/Documents/UFMG/2023-1/Redes/TPs/TP1$ ./client ::1 51511
Connected to IPv6 ::1 51511
comando invalido
disconnected due to incorrect command
araujo@LAPTOP-TRRBOCNO: /mnt/c/Users/arauj/Documents/UFMG/2023-1/Redes/TPs/TP1$

araujo@LAPTOP-TRRBOCNO: /mnt/c/Users/arauj/Documents/UFMG/2023-1/Redes/TPs/TP1/serv$ ./server v6 51511
[log] Bound to IPv6 :: 51511, waiting connections
[log] Waiting for new client
[log] connected from IPv6 ::1 53964
[log] Waiting for new client

```

Observação sobre a entrada: Este TP funciona como especificado tanto passando os comandos um a um via terminal (stdin) tanto quanto usando um arquivo txt que contém comandos separados por quebras de linhas e passando-o para a stdin usando o operador “<” (exemplo: ./client ::1 51511 < commands.txt), desde que sejam no padrão Linux de quebras de linhas (padrão LF: Line Feed), ou seja, somente o caractere ‘\n’ (decimal = 10 na tabela ASCII). Desse modo, o arquivo txt de comandos reflete com fidelidade o processo de digitar comandos um a um no terminal apertando enter para enviá-los, que adiciona um ‘\n’ ao buffer. Se o padrão CRLF (Carriage Return, Line Feed) for usado nos arquivos txt de comandos (ou caso os testes não incluam o char ‘\n’ ao fim do buffer), o comportamento será errático, uma vez que esse padrão, comumente associado ao Windows, adiciona um caractere ‘\r’ (decimal 13 na tabela ASCII, o carriage return) antes do caractere ‘\n’.

Exemplo de txt com comandos:



```

client.c 5  commands.txt X  server.c 5
TP1 > commands.txt
1 send file
2 select file teste.java
3 send file
4 send file
5 select file teste.txt
6 select file teste.tex
7 send file
8 exit
9

```

Na perspectiva do servidor, ele espera receber os bytes do cliente e realiza as possíveis validações antes de entrar no campo onde o arquivo (se houver) é processado. Primeiro trata se os bytes recebidos foram maiores que 0: se for -1, houve falha no *recv()* e ambas as conexões são

terminadas, se for 0, então somente o cliente é desconectado e o servidor aguarda por uma nova conexão.

Como já explicado anteriormente, se não for a mensagem de arquivo correta, o servidor somente receberá "exit\\end", que termina ambas as conexões/execuções, ou receberá "invalid command\\end", que corresponde à uma resposta do tratamento de comando inválido do cliente, cujo resultado é somente desconectar o cliente.

Caso seja a mensagem correta, o servidor faz o *parse* da mensagem para recuperar o nome do arquivo e sua extensão. Então, ele lê até o "." e compara os primeiros caracteres do resto da mensagem (após o ".") com a mesma lista de extensões válidas citadas acima, redeclarada no `server.c`. Nesse sentido, se a string da extensão tem *n* caracteres, iremos olhar apenas para os *n* primeiros caracteres do resto da mensagem (usando `strncmp()` de `string.h`) e montar o nome do arquivo com a extensão cuja comparação foi bem sucedida. Além de montar o nome do arquivo, montamos também seu conteúdo em outra string apenas movendo o ponteiro em *n* posições e retirando o "\\end" ao final.

Observação Importante: Note que, em `server.c`, essa mesma lista de extensões válidas contém ".cpp" antes de ".c", uma vez que a conferência de extensões acima é feita sequencialmente, caso ".c" fosse analisado antes de ".cpp", os arquivos ".cpp" sempre seriam ignorados, de modo que seriam considerados arquivos ".c" com dois caracteres "p" no início. Note que essa ambiguidade poderia ser evitada, bem como o parsing seria mais facilitado, caso houvesse um espaço entre o nome do arquivo e seu conteúdo na mensagem recebida pelo servidor.

Em posse do nome do arquivo e de seu conteúdo, caso ele já exista no diretório em que o servidor se encontra, o arquivo é sobrescrito com o conteúdo passado pela mensagem e uma mensagem "file <nomearquivo> overwritten\\n\\end" é enviada para o cliente, que então imprime ela em seu terminal (sem o "\\end" ao final). Caso o arquivo não esteja presente no diretório do servidor (`servidor.out`) um novo arquivo é criado com o nome recebido, ele é escrito com o conteúdo recebido pela mensagem e, por fim, uma mensagem "file <nomearquivo> received\\n\\end" é enviada para o cliente que prontamente a imprime. Em ambos esses casos (arquivo escrito ou sobrescrito) nenhuma conexão/execução é encerrada e o cliente espera por um novo comando.

Dificuldades, desafios e imprevistos

As dificuldades que enfrentei ao longo da codificação deste TP foram algumas questões de implementação na linguagem C e confusões acerca da especificação postada. Nesse sentido, muitos Segmentation Faults, erros de alocação de memória e buffers com textos errados ocorreram devido ao aspectos de envio de strings via socket, os quais eu não estava acostumado. Acerca das confusões da especificação, várias vezes tive que alterar detalhes de implementação devido às dúvidas que surgiram (tanto por minha parte quanto por parte dos alunos) por conta de aspectos não bem definidos na especificação, tanto quanto aspectos contraditórios entre o fórum de dúvidas e a especificação. Por exemplo, não ficou claro para mim o comportamento do protocolo para comandos errados do cliente. Na especificação é dito que o servidor deve desconectar o cliente (sem mencionar que o servidor deve ser desconectado, mas em <https://virtual.ufmg.br/20231/mod/forum/discuss.php?d=25713#p52267>, é afirmado que o servidor também deve ser desconectado em caso de comando errado. Por fim, adotei uma solução que acredito estar mais próxima do especificado: por meio de uma troca de mensagens específica, o servidor desconecta o cliente (que por sua vez tem sua execução terminada) e volta para o loop mais externo para esperar uma nova conexão de um novo cliente. Essa escolha foi feita pois não julguei adequado o servidor terminar sua execução por conta de um comando errado do cliente, a menos que o comando seja o exit correto.