Trabalho Prático - Redes de Computadores

Turma: TM

Professor: Aldri Luiz dos Santos

Aluno/Nº Matrícula: Gustavo Santana Cavassani / 2021070845

1. Introdução

Como enfatizado na proposta do trabalho, a comunicação e colaboração entre equipamentos e sistemas é de suma importância para otimização de processos e melhora da tomada de decisões. Paralelo a isso, o protocolo TCP é uma grande ferramenta nesse processo, garantindo a segurança e a conexão entre os envolvidos na comunicação.

Assim, a fim da correta implementação do cliente-servidor e troca de informações/mensagens, foi implementado uma comunicação por protocolo TCP em meu sistema, onde, para cada comando (ou implementação), foi feita a troca de "confirmações" ou "requisições, ou seja, o servidor recebia primeiro um pedido do equipamento, checava em seu banco de dados pelo que foi requisitado, e em caso positivo, retornava para o cliente o que foi requisitado; Dessa forma, observa-se a correta implementação do protocolo TCP no cliente-servidor.

Observação: Infelizmente, não consegui realizar a implementação do peer-to-peer por estar encontrando erros demais na lógica, logo, deixei nos códigos apenas a implementação do cliente-servidor.

2. Mensagens

Para realização das mensagens no cliente-servidor, decidi implementar o uso de strings compostas do ID da mensagem, do ID do equipamento (caso fosse necessário) tanto de origem, quanto de destino, e no final da string, a informação útil, ou payload (caso fosse necessário).

Dessa forma, cada mensagem ficava da seguinte forma:

- **REQ_ADD**: "05"
- **REQ_REM**: "06 IdEquip"
- **RES_ADD**(Para recebimento do New ID): "07 NewId"
- **RES_ADD**(Para adicionar novo ID ao banco de dados): "07 IdEquip"
- **RES_LIST**: "08 IdEquip1\n08 IdEquip2\n.." (Fui concatenando cada RES_LIST)

Obs.: Por conta de problemas para diferenciar os ID's recebidos na string do RES_LIST, fiz uma lógica "extensa" que tratava cada ID separadamente, foi a única implementação que funcionou corretamente.

- **REQ_INF**: "09 orig: IdOrig dest: IdDest"

Obs.: Para conseguir diferenciar qual ID de destino e qual o ID de origem, coloquei os termos 'orig' e 'dest' antes de cada ID.

- **RES_INF**: "code: 10 orig: IdOrig dest: IdDest payload: Payload"

Obs.: Para não conflitar o ID da mensagem com o ID do equipamento 10 (caso o mesmo fosse criado), coloquei o termo 'code' antes do ID da mensagem; O ID de origem e destino tem a mesma implementação que a mensagem acima, e para a carga útil foi colocado 'payload' antes da informação.

- **ERROR**: "11(0X)"

Obs.: Para cada error, era enviado e recebido apenas o ID da mensagem e o payload, que era o código do erro.

- **OK**: "12(01)"

Obs.: Similar ao error, mas apenas ocorre para código 01, quando um equipamento é removido com sucesso.

Assim, para mensagens que não necessitavam do ID do equipamento como REQ_ADD e ERROR(04), foi usado apenas uma comparação de strings (strcmp) entre o que estava chegando no buffer e o código em questão. Já para mensagens como

RES_ADD, RES_LIST,..., que tem junto do seu ID de mensagem, o ID do equipamento ou payload, foi usada uma lógica onde, eu armazenava em uma variável um ponteiro para o ID da mensagem, e casa esse fosse diferente de nulo, armazena em outra variável (ou outras variáveis) o ID do equipamento em questão, e caso houvesse, o payload da mensagem.

```c
if (strcmp(errorMsg, "11(04)") == 0)
{
  printf("Equipment limit exceeded\n");
```

Figura 1 – Exemplo Comparação de Strings

```c
char *RES_ADD = strstr(buffer, "07");    // Pega o ID da mensagem RES_ADD
int IDEquip = atoi(strchr(buffer, ' ')); // Pega o ID do equipamento

if (RES_ADD != NULL)
{
```

Figura 2 – Exemplo Uso de Ponteiros

3. Arquitetura

A arquitetura da comunicação cliente-servidor do sistema ficou composta de dois arquivos principais:

- **Server.c:** Responsável por implementar o código do servidor. Recebe e trata as mensagens enviadas pelos equipamentos, faz a diferenciação das mensagens de controle, retorna os error's e confirmações para cada equipamento corretamente, e faz o controle das conexões, limitando a 15 equipamentos conectados simultaneamente, fechando a conexão quando necessário.
- **Equipment.c:** Responsável por implementar o código do equipamento. Faz o envio de requests via terminal ao servidor, recebe e trata mensagens vindas do servidor, atualizando seu banco de dados quando necessário.

Nas seções seguintes, cada parte da arquitetura será explicada detalhadamente.

4. Servidor

Para uma sequência no detalhamento do server.c e equipment.c, será explicado em ordem das linhas, ou seja, da primeira à última linha do código.

Primeiramente, foi feito os includes de bibliotecas padrões e outras bibliotecas necessárias para a programação com sockets, exceto uma biblioteca especial que usei para gerar um delay, time.h. Após os includes, foi feito a definição de valores fixos, como o máximo de equipamentos (MAX_EQUIPS) e tamanho máximo do buffer (BUFFER_SIZE), e a declaração de uma struct que usei para guardar cada equipamento criado, separando seus dados entre o ID do equipamento e o socket onde esse equipamento se encontra conectado, e um número de equipamentos atualmente conectados na rede, para fazer o controle de conexões.

```c
// Struct para armazenar a lista de conexões
typedef struct
{
  int ID;
  int socket;
} id_s;

id_s connectionList[MAX_EQUIPS];
int numEquips = 0;
```

Figura 3 – Struct para ID's e Equipamentos Presentes

Logo abaixo, foram declaradas as funções que utilizei para tratar erros no programa; 'usage' foi usado para tratar o caso em que o usuário passava incorretamente os argumentos para criar um servidor no terminal; 'logexit' foi uma das funções mais usadas no código, pois foi usada para tratar qualquer tipo de erro em ações como bind(), listen(), accept(), dentre outros, e por último, a função 'handleError' que criei inicialmente para tratar todos os error's, mas acabou que foi utilizada apenas para o ERROR(04) por problemas na implementação com outros error's, os quais implementei direto na lógica do programa.

```c
void handleError(char *errorMsg, int socket)
{
  if (strcmp(errorMsg, "11(04)") == 0)
  {
    send(socket, "11(04)", strlen("11(04)"), 0);
    // printf("maximum of 15 equipments connected reached\n");
    numEquips--;
    close(socket);
  }
}
```

Figura 4 – Função handleError

Entrando agora no main, foi feito o teste dos argumentos por meio do 'usage', o armazenamento da porta usada para a comunicação cliente-servidor (a do p2p ficou comentada, pois não foi utilizada) e do endereço IP do servidor. Após esses

procedimentos iniciais, até entrar no loop de comunicação entre servidor e clientes, foram feitos os seguintes procedimentos, em sequência: a criação do socket cliente-servidor para comunicação; a definição de opções do socket para habilitar o reuso (pois estava acontecendo um erro onde, quando eu fechava um servidor e tentava abrir outro logo depois, acusava erro de endereço já em uso); a criação e inicialização da estrutura do servidor; o bind do socket ao endereço; a realização do listen, pondo o socket para escutar por novas conexões; a inicialização com 0 da lista de conexões declarada anteriormente (connectionList); a definição do conjunto de descritores de arquivos atuais 'sockets' e do conjunto de cópia 'socketsReady', pois o select é destrutivo, então é usada uma cópia do conjunto principal para manipulação, e a definição do socket máximo (o 'último socket' do conjunto); a inicialização do conjunto principal, passagem do socket cliente-servidor para o conjunto de descritores e armazenamento do socket cliente-servidor como 'socket máximo', e por fim, a declaração do buffer que é usado o tempo inteiro dentro do loop para envio e recebimento de mensagens.

```
// Definindo a reutilização do endereço para corrigir erro de "Adress already in use"
int reuse = 1;
if (setsockopt(client_server, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse)) < 0)
{
  logexit("setsockopt() failed\n");
}
```

Figura 5 – Definição de Opções do Sock

Agora entrando no while, a principal parte do código onde ocorre toda comunicação e tratamento de mensagens. Primeiramente, é inicializada a cópia do conjunto de sockets e o endereço do equipamento; logo após, é feito o select passando o socket máximo e a cópia do conjunto de descritores como parâmetro. Feitas as inicializações e o select, foi criado um outro loop para iterar no conjunto de descritores de arquivos e o iterador desse loop define se o socket sendo analisado é o socket principal cliente-servidor ou outros sockets adjacentes criados após os equipamentos se conectarem. É feito o teste se o socket atual está 'setado', ou seja, se há dados chegando nele, e caso esteja, entra na lógica principal do código.

```
for (int sockIt = 0; sockIt <= maxSocket; sockIt++)
{
  if (FD_ISSET(sockIt, &socketsReady))
  {
```

Figura 5 – Iterador dos Sockets

```
while (1)
{
  socketsReady = sockets;
  struct sockaddr_in clientAddr;              // Definindo o ende
  socklen_t clntAddrLen = sizeof(clientAddr); // Definindo o tama

  if (select(maxSocket + 1, &socketsReady, NULL, NULL, NULL) < 0)
  {
    logexit("select() failed\n");
  }
```

Figura 6 – Inicialização e Select

Dentro do primeiro if (iterador de sockets é igual ao socket principal), é feito a aceitação do novo equipamento e o primeiro teste de **REQ_ADD**; caso o número de equipamentos não tenha sido atingido e o equipamento tenha enviado corretamente a mensagem de ID 05, o servidor retorna ao equipamento o **RES_ADD** com seu novo ID e adiciona ao seu banco de dados o ID do novo equipamento, junto do seu respectivo socket, além de enviar também o **RES_LIST**, atualizando o banco de dados do equipamento com todos equipamentos presentes atualmente no cluster; Nessa parte, foi colocado um comando 'sleep' que gera um pequeno delay, pois estava havendo um conflito entre o RES_ADD e o RES_LIST, onde em alguns momentos o RES_LIST não era enviado, mas gerando esse atraso dá tempo ao equipamento de receber seu ID e logo depois, receber e armazenar os ID's no cluster.

```
for (int i = 0; i < numEquips; i++)
{
  sleep(0.8); // Esse sleep foi colocado apenas para gerar um del
```

Figura 7 – Delay (sleep)

Caso esse socket já tenha sido identificado e recebido corretamente a lista de equipamentos, as outras condições são tratadas, como **REQ_REM**, **REQ_INF** e **RES_INF.** Como grande parte das mensagens de controle foram tratadas de forma similar, não irei explicar todas separadamente. Como introduzido na seção de mensagens, o método que utilizei foi utilizar um ponteiro que procura dentro de uma string (nesse caso, o buffer) por uma string passada como referência que no caso é o ID da mensagem; se esse ponteiro é diferente de nulo, ou seja, o ID da mensagem foi encontrado, a condição do if é atendida, entra-se no bloco da mensagem de controle atual, e dentro desse bloco é tratada mensagem respectiva, onde vou dar como exemplo o REQ_REM. Quando o bloco do REQ_REM é executado, armazena-se em uma variável o ID do equipamento por meio do 'strchr' (por isso foi implementado um espaço entre o ID da mensagem de controle e o ID do equipamento, pois o strchr pega o que vem após esse espaço) e do 'atoi' (que converte de string para inteiro); logo após, esse ID é procurado no banco de dados do servidor e caso seja encontrado, ele é removido do banco de dados e o seu socket recebe a mensagem OK(01) para o fechamento do equipamento; caso não seja encontrado, a variável 'remove' recebe 1 e aciona uma condição que envia o ERROR(01), sinalizando que o equipamento não foi encontrado no banco de dados.

Acho importante citar, na implementação do RES_INF, por conta do ID da mensagem ser 10, e isso poder conflitar com um equipamento de ID 10 (caso o mesmo seja criado), eu alterei a mensagem para ter o termo 'code: ' antes do ID da mensagem, para diferenciar o código da mensagem do ID do equipamento.

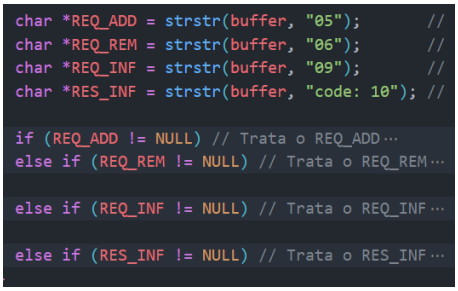

Figura 8 – Tratamento das Mensagens de Controle no Socket Princial

Figura 9 – Tratamento do REQ_REM

Figura 10 – RES_INF com code

Saindo da condição de que o iterador de sockets é igual ao cliente-servidor principal, e entrando no else, ou seja, o socket sendo tratado agora é um socket adjacente de um equipamento já conectado ao servidor. A ideia do que foi feito é similar ao que foi retratado acima; o servidor testa o recebimento de uma mensagem pelo equipamento, e caso esse seja igual a zero, significa que o equipamento foi fechado, então o servidor remove o socket do seu conjunto de descritores de arquivos; caso o teste de recebimento seja maior que 0, acontece o mesmo que foi feito acima, é armazenado em uma variável um ponteiro para o ID da mensagem de controle, e caso esse ponteiro seja diferente de nulo, a condição é atendida e a mensagem de controle respectiva é tratada. Isso resume o funcionamento do código server.c.

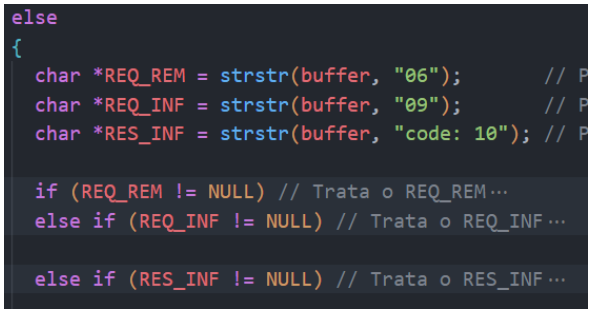

Figura 11 – Tratamento das Mensagens de Controle nos Sockets Adjacentes

5. Equipamento

De primeiro momento, o código equipment.c inicializa de forma similar ao server.c, com os mesmos define's, definições de valores máximos e funções utilizadas (usage, logexit e handleError), a única diferença é que a lista utilizada como banco de dados para armazenas os ID's é um array de inteiros. Entrando no main, ocorrem os seguintes procedimentos: declaração do array dos ID's e inicialização desse array, e declaração do buffer usado para troca de mensagens; teste dos argumentos passados para criação do equipamento; armazenamento do endereço do servidor e da porta usada para o socket; criação do socket do equipamento; declaração e inicialização do conjunto de descritores de arquivos do equipamento (diferente do servidor, no equipamento é usado apenas um conjunto), e do socket máximo como o socket criado para o equipamento; criação da estrutura do endereço do servidor e conversão do endereço IPV4 para string (ou binário) e armazenando na porta da struct criada.

Após esses procedimentos iniciais, antes de entrar no loop de comunicação entre equipamento e servidor, o equipamento faz o connect com o servidor, e ao fazer esse connect, é feito o envio do **REQ_ADD** ao servidor, e esse faz o tratamento, caso o número de equipamentos não tenha sido excedido, o equipamento recebe o a mensagem contendo **RES_ADD** e o seu novo ID, entrando na condição else, onde acontecerá o mesmo que foi feito no servidor, um ponteiro para o ID da mensagem e um inteiro que armazena o ID do equipamento.

Figura 12 – Primeiro REQ_ADD para Novo ID

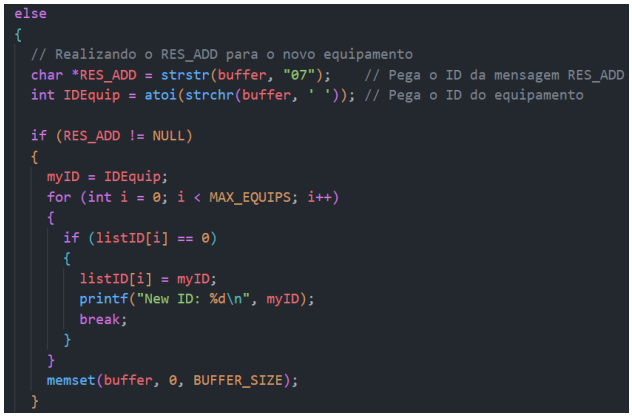

Figura 13 – Tratamento do RES_ADD

Após o RES_ADD ter sido feito com sucesso, entra-se na parte principal do código, o loop while. Dentro do loop, o conjunto de descritores de arquivos são inicializados e 'setados' toda vez (stdinfileno é setado também para receber dados do terminal); após a inicialização, é feito o select, similar ao servidor e caso esse seja feito com sucesso, entra na parte principal do código. Após o select, são feitos dois ifs: o primeiro testa se há dados no socket do equipamento, caso haja, o equipamento recebe a mensagem e o processo é o mesmo do servidor, um ponteiro recebe o ID da mensagem e caso esse ponteiro seja diferente de nulo, a condição da mensagem de controle é atendida e o controle dela é realizado.

Figura 14 – Tratamento das Mensagens de Controle

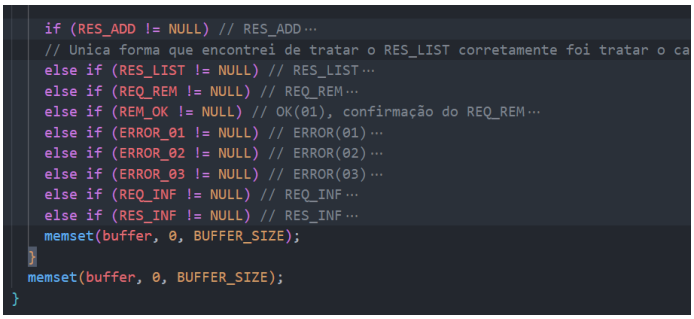

Figura 15 – Tratamento das Mensagens de Controle

Por conta de um problema de conflito ao tratar o RES_LIST, onde os ID's conectados na rede estavam sendo concatenados de forma errada, acabei tendo que fazer uma implementação mais extensa que checa cada ID separadamente, do 1 ao 15, mas que felizmente funcionou.

Figura 16 – RES_LIST

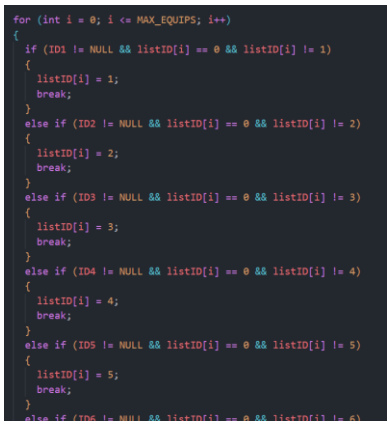

Figura 17 – RES_LIST

Agora indo pro segundo if, que testa se há dados vindos do terminal, apenas três condições são testadas, que são a de ./close connection, que envia o código 06 para remoção do equipamento; a de ./list equipment, que faz a listagem dos equipamentos conectados no cluster atualmente printando todos os ID's presentes no array de ID's, e a de ./request information from ID, que faz envio do REQ_INF, passando seu ID como ID de origem e o ID desejado como o ID de destino, juntos do ID 09. O último else não é necessário, mas deixei apenas para caso queira enviar uma mensagem ao servidor, tem essa opção. Isso resumo o funcionamento do código equipment.c.

Figura 18 – Tratamento de Dados do Terminal

6. Discussão

Observa-se então que foram encontrados diversos desafios durante a implementação dos códigos. A forma como enviar os ID's das mensagens de controle juntamente dos ID's dos equipamentos em uma string, e separar os mesmos no outro extremo que recebe eles concatenados, foi o principal problema enfrentado, além de outros problemas como a forma de armazenamento dos ID's no servidor, que foi solucionado pela criação de uma struct, dentre outros diversos problemas e questões.

7. Conclusão

Por fim, observa-se a utilização do protocolo TCP para a comunicação entre equipamentos de uma indústria por meio de um servidor que conecta todos em um cluster, onde esse servidor faz todo o controle e gerenciamento necessário para o funcionamento correto do processo, promovendo a troca de informações em tempo real para otimizar produção em geral.