Documentação TP2

Redes de Computadores

Nome: Lucca Silva Medeiros
Matrícula: 2019054773

1) Introdução

O trabalho prático consiste na implementação de uma solução que permita a instalação, desinstalação e monitoramento de equipamentos industriais em rede através de um software. Dado este cenário, foi desenvolvido um protocolo que suporta múltiplas conexões entre os equipamentos e o servidor, simulando a interação entre os agentes (equipamentos), responsáveis por fazer as solicitações de conexão, desconexão, listagem, e comunicação dos demais equipamentos também conectados na rede. E o controlador (servidor), responsável por centralizar e executar as mensagens solicitadas pelos agentes.

2) Arquitetura da comunicação

A implementação do trabalho se deu através da evolução do primeiro trabalho prático da disciplina, onde foi estabelecida a conexão de um só cliente com o servidor, desenvolvido através de POSIX sockets na linguagem C, utilizando a biblioteca padrão de sockets TCP. Foi reaproveitado do primeiro trabalho toda a infra-estrutura de estabelecimento da conexão:

- **Servidor**: Inicialização (socket), Abertura Passiva (bind, listen e accept), Comunicação (recv e send) e Finalização (close).

- **Clientes**: Inicialização (socket), Abertura Passiva (connect), Comunicação (send e recv) e Finalização (close).

Porém, agora o servidor deve suportar múltiplas conexões, sendo capaz de atender até 10 clientes simultaneamente, para isso foi implementado um servidor multi-threaded. A grande mudança feita por parte no servidor pode ser observada na seguinte estrutura:

```
int csock = accept(s, caddr, &caddrlen);
if (csock == -1)
{
    logexit("accept");
}

struct client_data *cdata = malloc(sizeof(*cdata));
if (!cdata)
{
    logexit("malloc");
}

cdata->csock = csock;
memcpy(&(cdata->storage), &storage, sizeof(storage));

pthread_t tid;
pthread_create(&tid, NULL, client_thread, cdata);
```

Após o estabelecimento da conexão entre o servidor e um equipamento (pós *accept* com atribuição de um socket de comunicação para o cliente), o servidor atribui para o cliente atual uma thread que processa em paralelo a comunicação estabelecida: < *pthread_create(&tid, NULL, client_thread, cdata);* >. Com isso o programa principal fica disponível novamente para esperar um novo *accept*, sendo possível estabelecer uma nova conexão com outro equipamento. Com isso tem-se um servidor multi-threaded funcional.

Agora o desafio é implementar a parte da comunicação por parte do equipamento. Haja vista as especificações do TP, o cliente deve ser capaz de receber mensagens em broadcast, ou seja, a qualquer momento o cliente deve ser capaz de receber e processar uma mensagem do servidor. Com isso a implementação feita no TP1 é insuficiente para atender essa nova demanda, uma vez que antes o cliente ficava preso na leitura do teclado e só aguardava uma resposta após envio de uma mensagem. Com isso foi implementado a seguinte mudança:

```c
// Inicializar Conexao
struct sockaddr *addr = (struct sockaddr *)(&storage);
if (0 != connect(s, addr, sizeof(storage)))
{
    logexit("connect");
}

char addrstr[BUFSZ];
addrtostr(addr, addrstr, BUFSZ);

fflush(stdin);

struct client_data *cdata = malloc(sizeof(*cdata));
if (!cdata)
{
    logexit("malloc");
}

cdata->s = s;

pthread_create(&tid_send, NULL, send_thread, cdata);
pthread_create(&tid_recv, NULL, recv_thread, cdata);

while (sendConnection && recvConnection)
{
    continue;
}
close(s);
exit(EXIT_SUCCESS);
```

Sempre que o equipamento estabelece uma conexão com o servidor ele cria 2 threads:

- **send_thread**: responsável pelo processamento e execução do envio das mensagens do equipamento para o servidor.

- **recv_thread**: responsável pelo processamento e execução do recebimento das mensagens enviadas do servidor para o equipamento.

Com isso, o equipamento agora é capaz de a qualquer momento receber ou enviar uma mensagem do servidor, pois ambos processos foram separados nas threads acima, que rodam em paralelo.

3) Implementação do Servidor

De acordo com as especificações do TP, o servidor deve ser capaz de centralizar e executar as mensagens solicitadas pelos agentes (Equipamentos). Para isso foi implementada a seguinte estrutura de dados:

```
struct Equip_s
{
   int ids[MAX_EQUIP];
   int csocks[MAX_EQUIP];
   int count;
} Equip_default = {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, 0};
typedef struct Equip_s Equip;

Equip server;
```

O servidor comporta dois arrays. *ids[MAX_EQUIP]* é responsável por armazenar os ids dos equipamentos instalados de forma booleana, ou seja: 1 significa que o ID em questão está instalado no servidor e 0 significa que o ID em questão não está instalado. Os ids são na verdade sua posição no array, e a representação gráfica para a impressão dos comandos é sempre id + 1, exemplo:

- **Servidor vazio sem nenhum equipamento instalado (condição inicial):**
  ids[MAX_EQUIP] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

- **Equipamento 01, 02 e 04 instalados:**
  ids[MAX_EQUIP] = [1, 1, 0, 1, 0, 0, 0, 0, 0, 0]

- **Equipamento 09 e 010 instalados:**
  ids[MAX_EQUIP] = [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]

Tabela de representação:

Posição no Array	Equipamento
ids[0]	01
ids[1]	02
ids[2]	03
ids[3]	04
ids[4]	05
ids[5]	06
ids[6]	07
ids[7]	08
ids[8]	09
ids[9]	010

Já o array *csocks[MAX_EQUIP]* é responsável por armazenar os sockets de comunicação de cada equipamento. O valor = 0 significa que ainda não foi registrado um socket para aquele equipamento, já qualquer outro valor inteiro representa o socket do equipamento em questão, exemplo:

- **Socket do equipamento 01 é 4:**
  csocks[0] = 4

A função deste array é principalmente armazenar as informações necessárias para se fazer um “broadcast” quando necessário, pois basta iterar sobre cada item em *csocks[MAX_EQUIP]* cujo valor for diferente de 0, e enviar uma mensagem unicast para cada equipamento, desta forma é possível enviar a mesma mensagem para todos os equipamentos conectados na rede, produzindo o mesmo efeito de uma mensagem broadcast.

Por último, o int *count* representa um contador de equipamentos atualmente instalados no servidor. Sua função é facilitar quaisquer implementação onde se deve levar em consideração o número atual de equipamentos na rede, como por exemplo ao adicionar um novo equipamento deve-se verificar se o limite de 10 equipamentos foi atingido.

4) Implementação do Equipamento

De acordo com as especificações do TP, o equipamento deve ser capaz de fazer as solicitações de conexão, desconexão, listagem, e também solicitar informações dos demais equipamentos também conectados na rede. Para isso foi implementada a seguinte estrutura de dados:

```
struct Equip_s
{
   int ids[MAX_EQUIP];
   int myId;
} Equip_default = {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, 0};
typedef struct Equip_s Equip;
```

De forma semelhante à implementação do servidor, ids[MAX_EQUIP] é responsável por armazenar os ids dos equipamentos instalados: 1 significa que o ID em questão está instalado na rede, 2 significa que é o próprio ID do equipamento e 0 significa que o ID em questão não está instalado. Além disso, o int *myId* também armazena o valor do ID do equipamento em questão, porém na representação gráfica para a impressão dos comandos, ou seja, id + 1, exemplo:

- **Rede com equipamentos 01 e 02 instalados:**
  - **Struct *Equip*_s do equipamento 01:**
    ids[MAX_EQUIP] = [2, 1, 0, 0, 0, 0, 0, 0, 0, 0]
    myId = 1
  - **Struct *Equip*_s do equipamento 02:**
    ids[MAX_EQUIP] = [1, 2, 0, 0, 0, 0, 0, 0, 0, 0]
    myId = 2

5) Discussão

A implementação deste trabalho foi de fato desafiadora, principalmente no que diz respeito a parte da comunicação. A implementação do multi-thread no servidor foi de fácil adaptação do servidor já desenvolvido no TP1, principalmente por conta do material de apoio sugerido na documentação. Porém a parte da comunicação do equipamento exigiu uma pesquisa e um estudo mais profundo para sua implementação.

Com a comunicação implementada o trabalho seguiu bem, principalmente por conta das descrições detalhadas presentes na documentação de cada comando que deveria ser executado. Todas as mensagens descritas foram implementadas exatamente iguais, salvo as seguintes exceções:

Em um cenário onde somente os equipamentos 01 e 02 estivessem instalados, a documentação descreve o comando RES_LIST como: RES_LIST(01 02) porém implementei da seguinte forma: RES_LIST(1, 1, 0, 0, 0, 0, 0, 0, 0, 0).

Optei por enviar o meu array de IDs pois dessa forma fica mais fácil iterar sobre a string de resposta e facilmente identificar os IDs instalados ou não.

Outra mudança foi a respeito do equipamento de ID = 10. As mensagens que envolvem esse ID são identificadas como ID = 0, para facilitar a leitura da string no receptor (haja vista que as strings terão o mesmo tamanho, pois o 10 adicionaria um caractere a mais). Porém na linha de comando o equipamento ainda é tratado como 010. Exemplo de comunicação:

- **Terminal do equipamento 09**: request information from 010
- **Servidor recebe**: REQ_INF(09 00)
- **Servidor envia para equipamento 010**: REQ_INF(09 00)
- **Equipamento 010 envia para servidor:** RES_INF(00 09 payload)
- **Servidor envia para equipamento 09:** RES_INF(00 09 payload)
- **Terminal do equipamento 09**: value from 010: payload

6) Conclusão

O desenvolvimento deste trabalho possibilitou a aplicação prática de diversos conceitos estudados durante a disciplina de redes de computadores. Com a implementação das etapas descritas acima foi possível implementar o protocolo de comunicação proposto e assim entender melhor como funciona a comunicação na prática via sockets no formato cliente-servidor, em um contexto multi-thread.