Documentação TP2 - Sistema de múltiplas conexões

Aluno: Bernardo Gomes Nunes
Matrícula: 2019054374

O Trabalho Prático 2 da disciplina de Redes de Computadores consistiu em construir um sistema de comunicação centralizado, responsável por coordenar múltiplas conexões simultâneas entre equipamentos e permitir a comunicação entre eles. Esse sistema é composto por dois programas:

1. **Server (Servidor):** conecta múltiplos equipamentos e funciona como "intermediador" da comunicação .Ele garante que as solicitações fazem sentido e repassa as mensagens para os equipamentos de destino.
2. **Equipment (Equipamento):** conecta-se ao servidor e pode solicitar informações de outros equipamentos da rede. Além disso, ele também deve responder às solicitações que chegam do servidor.

O primeiro desafio consistiu em programar o servidor para lidar com múltiplas conexões. Para tanto, utilizei a biblioteca *pthread*, seguindo os exemplos presentes na seção 6.4.2 do livro "TCP IP Sockets in C, Second Edition Practical Guide for Programmers". Tendo como base o código desenvolvido no Trabalho Prático 1, foram necessárias poucas adaptações: criei uma função principal para as *threads*, que consiste basicamente em um *loop* que escuta as mensagens de um determinado cliente (ver Imagem 1); criei uma *struct* que é passada como argumento para a função recém criada, ela contém apenas o descritor de arquivo do cliente (ver imagem 2)

```c
// Programa principal de cada Thread
void *ThreadMain(void *args) {
    // Garante que os recursos da thread sejam desalocados no final
    pthread_detach(pthread_self());

    // Recupera o identificador socket dos parâmetros
    int clntSocket = ((ThreadArgs *) args)->sock;

    // Escuta solicitações do cliente
    handleTCPClient(clntSocket);

    free(args);
    return NULL;
}
```

Imagem 1- *Thread* que roda no servidor e escuta um determinado cliente

```c
typedef struct ThreadArgs {
    int sock;
} ThreadArgs;
```

Imagem 2 - Estrutura passada como argumento para as *Threads*

Para armazenar os equipamentos conectados ao servidor, utilizei uma matriz de inteiros 15x2. Cada linha da matriz representa um equipamento. São 15 linhas porque essa foi a quantidade máxima de conexões simultâneas especificada. A primeira coluna da matriz representa o identificador do cliente, que é atribuído pelo sistema no momento da conexão. A segunda coluna armazena o descritor de arquivo do cliente, que é necessário para fazer a comunicação.

De acordo com a especificação do Trabalho Prático, os clientes (equipamentos) também possuem uma lista dos demais clientes conectados. Sendo assim, criei um vetor de inteiros no programa “Equipamento” para armazenar os identificadores dos demais equipamentos da rede. Diferentemente do servidor, não houve necessidade de criar uma matriz, já que um cliente não precisa ver o descritor de arquivo dos outros clientes.

Os identificadores dos equipamentos foram implementados como números inteiros positivos sequenciais. Foi criada uma variável de controle que armazena o próximo identificador (que será atribuído ao próximo equipamento que se comunicar). Ela é incrementada a cada conexão. Quando um cliente é desconectado, seu identificador fica “aposentado”, pois quebraria a sequência. Tendo em vista o escopo deste trabalho, acredito que não seja um problema pois a quantidade de inteiros que podem ser representados é relativamente grande. No entanto, em um cenário real, poderia ser necessário que esses identificadores ficassem disponíveis para futuras conexões.

Criou-se uma função utilitária para formatar um identificador numérico e convertê-lo em uma sequência de caracteres. Essa formatação consiste em inserir o caractere “0” à esquerda do número, caso ele seja menor que dez. Dessa forma, a saída impressa no terminal estará de acordo com os exemplos que foram apresentados na especificação.

Para representar as mensagens, criou-se uma estrutura como a mostrada na imagem abaixo:

```c
typedef struct Message {
    int id;
    int originId;
    int destinationId;
    char payload[100];
} Message;
```

Imagem 3 - Estrutura das mensagens

A *struct* descrita na imagem acima descreve as mensagens exatamente da mesma forma que a especificação:

- ***id**:* representa o tipo da mensagem;
- ***originId**:* identificador do equipamento de origem;
- ***destinationId***: identificador do equipamento de destino;
- ***payload***: corpo da requisição que varia de acordo com o tipo da mensagem.

Escolheu-se o tamanho do campo *payload* baseando-se na quantidade máxima de conexões e nas características de cada tipo de mensagem. Caso esse limite seja expandido ou uma nova instrução for adicionada ao protocolo, esse tamanho deverá ser revisto.

Para enviar as mensagens na rede, foi necessário converter a *struct* em uma *string.* Para isso, criei uma função comum utilizada tanto no servidor quanto nos clientes. Da mesma forma, criei uma função inversa para converter *string* em *struct* após os dados serem recebidos. Foram utilizadas constantes para definir os delimitadores de campo (**‘|’**) e final da mensagem (**‘\n’**). Quando um campo não é necessário para determinado tipo de mensagem, seu valor é zero ou uma string vazia (quando se trata do *payload*)

Os identificadores de tipo da mensagem e código de erro são constantes definidas conforme a imagem abaixo:

```
// IDs das mensagens
#define REQ_ADD  1
#define REQ_REM  2
#define RES_ADD  3
#define RES_LIST 4
#define REQ_INF  5
#define RES_INF  6
#define MSG_ERR  7
#define MSG_OK   8

// Possíveis erros
#define ERR_EQUIP_NOT_FOUND      1
#define ERR_SRC_EQUIP_NOT_FOUND  2
#define ERR_TAR_EQUIP_NOT_FOUND  3
#define ERR_EQUIP_LIMIT_EXCEEDED 4
```

Imagem 4 - Definição de constantes

De acordo com as instruções do trabalho, os equipamentos devem ser capazes de enviar mensagens para o servidor quando receberem instruções via linha de comando (exceto “list equipment”). Além disso, eles também devem estar sempre preparados para receber mensagens do servidor. Para conseguir implementar os dois comportamentos, utilizei *threads* também nos clientes.

Para cada equipamento, foi criada uma *thread* responsável por escutar e processar as entradas realizadas por linha de comando. Enquanto isso, o programa principal fica aguardando as mensagens do servidor e realiza as ações necessárias quando as recebe.

O sistema funciona corretamente para os casos que foram apresentados nas instruções, mas foi necessário definir comportamentos para casos excepcionais.

1. **Instruções via linha de comando**: quando o usuário envia uma instrução que não é identificada pelo sistema, nada acontece. Optei por manter o cliente conectado e não imprimir mensagem de erro, por receio de impactar em possíveis testes automatizados.
2. **Remoção de equipamentos**: nada impede que um equipamento *x* tente remover um equipamento *y.* Parti da premissa de que um equipamento só solicitará a remoção dele próprio.
3. **Encerramento do programa de equipamento**: o programa do equipamento foi programado para encerrar em dois momentos: após receber a confirmação do “REQ_REM” e ao receber o erro “Equipment limit exceeded”. Caso ocorra um encerramento forçado (Ex.: Ctrl+C) os demais programas continuarão executando, mas não serão notificados que o equipamento foi removido.