TP2 Redes - Sistema de múltiplas conexões

Nome: Vinícius Assis Neves
Matrícula: 2019055036

1. Introdução

Este projeto foi desenvolvido com o propósito de simular o fluxo de troca de informações entre equipamentos na indústria 5.0. Existe um servidor que faz a conexão com cada cliente, armazena os dados e possibilita o comunicação entre clientes; e os clientes, que são os equipamentos. Cada cliente terá 3 comandos básicos: "list equipment", "request information from {id}" e "close connection"

O projeto foi desenvolvido na linguagem C, sendo 2 programas diferentes: um contendo o código do cliente e outro contendo o código do servidor. A comunicação é feita com o protocolo TCP por meio da biblioteca de sockets POSIX. Para suportar múltiplas conexões, são utilizadas threads, que serão melhor explicadas na seção da arquitetura e servidor.

2. Arquitetura

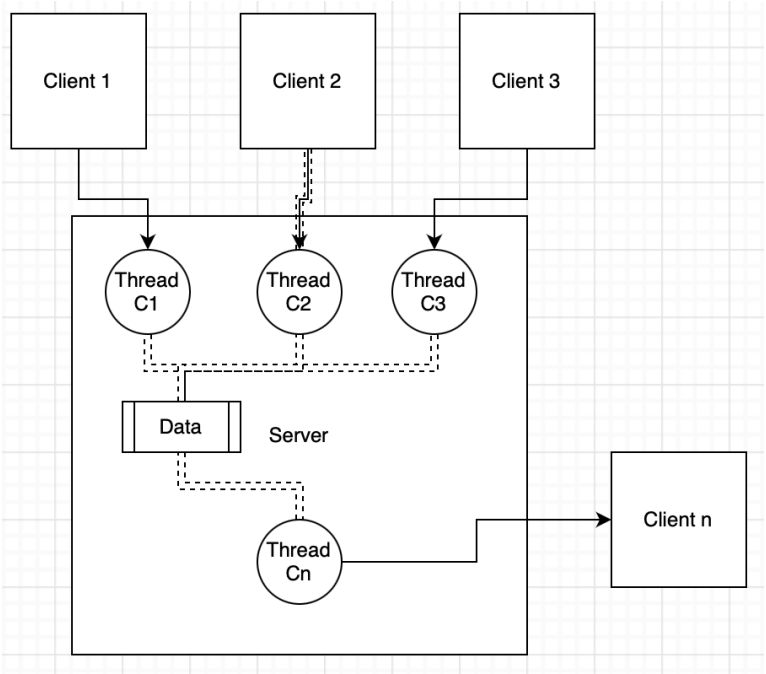

Figura 1. Arquitetura do projeto

O projeto possui uma arquitetura simples. Temos um servidor que, em seu loop principal, aguarda novas conexões. Ao estabelecer uma nova conexão, uma thread dedicada a receber mensagens do cliente que acabou de se conectar é criada. A thread possibilita que o servidor espere por mensagens de vários clientes de forma simultânea não bloqueate. Apesar de rodar em paralelo, todas as threads tem acesso à memória do servidor, ou seja, a quais equipamentos estão conectados e ao id da conexão de cada equipamento, o que possibilita que a thread do cliente 1 (Thread C1), por exemplo,

envie uma mensagem ao Cliente 2. Caso o servidor possuísse mais dados releventes em memória (como por exemplo um cache dos dados de cada equipamento), as threads também teriam acesso à estes. Quando a conexão é estabelecida, um equipamento é atribuído um ID, que é utilizado para identificá-lo na rede.

O projeto foi dividido em 3 arquivos de código:

1. server.c: contém o código do servidor
2. equipment.c: contém o código do cliente
3. common.h: possui algumas funções de uso geral

3. Servidor

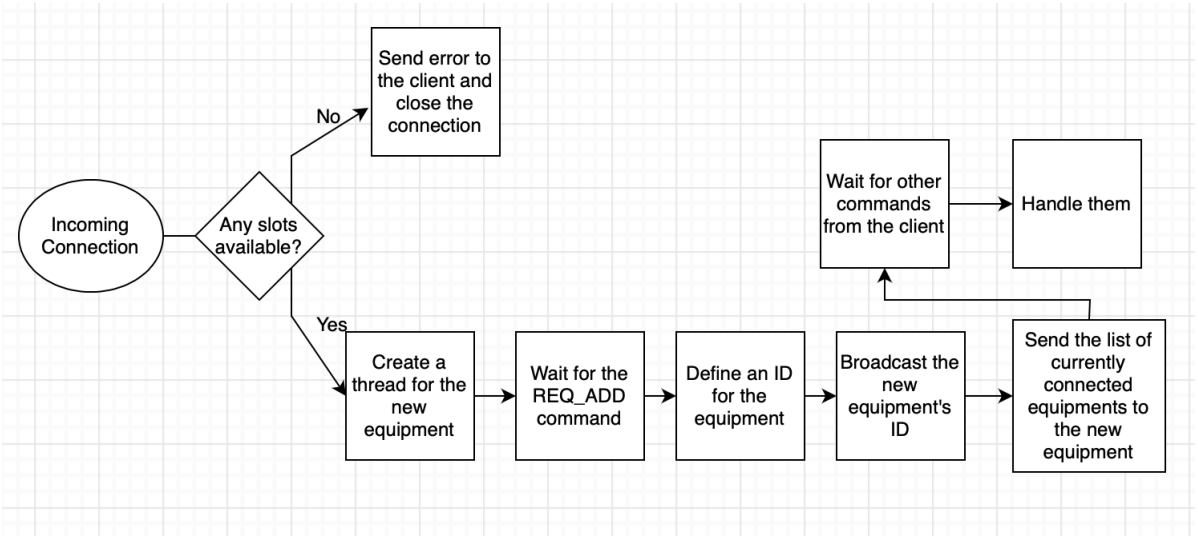

Figura 2. Fluxo de comunicação lado servidor

Após estabelecer a conexão, se há slots disponíveis, o servidor criará uma thread para o equipamento, que terá um id incremental, de 1 a 15. Tal id, pode ser reusado caso um equipamento se desconecte. Com a thread aberta, o servidor está pronto para receber comandos, sendo o primeiro deles o "REQ_ADD", que solicita o definição do id. O id do equipamento, é o mesmo da thread, e o broadcast deste será feito para informar todos os clientes (inclusive o solicitante), da adição do novo equipamento.

O servidor possui algumas estruturas de dados principais, sendo a maioria delas arrays, são elas:

- bool busyThreads[MAX_EQUIPMENTS + 1]: Do tipo booleano, guarda o status se a thread de id i está ocupada ou não. Ou seja, se busyThreads[i] for verdadeiro, o equipemento de id i já estabeleceu uma conexão com o servidor (mas isso não garante que a adição do equipamento já foi propriamente feita por meio do "REQ_ADD".
- bool equipments[MAX_EQUIPMENTS + 1]: Do tipo booleano, guarda o status de conexão do equipamento de id i. Se equipments[i] é verdadeiro, um equipamento está conectado e já solicitou a adição no sistema por meio do comando REQ_ADD. Se falso, ele pode estar conectado, mas não solicitou o REQ_ADD ainda.

- Int threadSocketsMap[MAX_EQUIPMENTS + 1]: Do tipo inteiro, esta array funciona como um dicionário que mapeia os ids dos equipamentos para os ids das conexões estabelecidas, para que o programa possa localizar um equipamento específico pelo seu id e enviar uma mensagem.
- pthread_t threads[MAX_EQUIPMENTS + 1]: Guarda as threads criadas para cada id.
- struct threadArgs: Utilizada para passagem de argumentos para a thread. Ela guarda o id do socket e o id definido para a thread/equipamenot.

Os ids são empre exibidos acrescidos de um 0 inicial, caso sejam menor que 10. Porém, por conveniência, utilizamos eles como inteiros, para poderem ser índices das arrays de forma simples. Como não existe um id 0, também por conveniência, o tamanho das arrays é MAX_EQUIPMENTS + 1, ou seja 16. Dessa forma, a posição 0 não é utilizada, mas sabemos que, por exemplo, equipments[15] se refere ao estado de conexão do equipamento de id 15.

3.1. Métodos principais

- int main(int argc, char const* argv[]): Configura o socket com IPV4 e protocolo TCP e espera conexões de clientes até que o programa seja encerrado.
- Bool _handleMessage(int equipId, char *message): Recebe o id do equipamento que enviou a mensagem e a mensagem em si. Divide a mensagem em tipo e argumentos, e direciona os argumentos para a função _handle correta que executará as açoes necessárias para o tipo de mensagem.
- *void _sendMessage(char* idMsg, int originEqId, int destinationEqId, char* payload, int destinationId*

  Este método basicamente recebe os valores a ser transmitidos na mensagem, combinam os valores relevantes para o idMsg (que pode ser dos tipos RES_ADD, RES_REM, REQ_INF, ERROR ou OR), os separando por espaço e terminando com uma quebra de linha "\n", e os envia para o *destinationId*. O parâmetro ***destinationId*** pode ser -1 (constante DESTINATION_EQ_ID), indicando que o destino é igual ao parâmetro *destinationEqId,* ou maior que 0, indicando o id da conexão para onde a mensagem será enviada. Note que *destinationId e destinationEqId* podem não se referirem ao mesmo equipamento (RES_INF, por exemplo, transmitirá para o *originEqId,* já que está respondendo ao REQ_INF de *originEqId.*

4. Equipamento

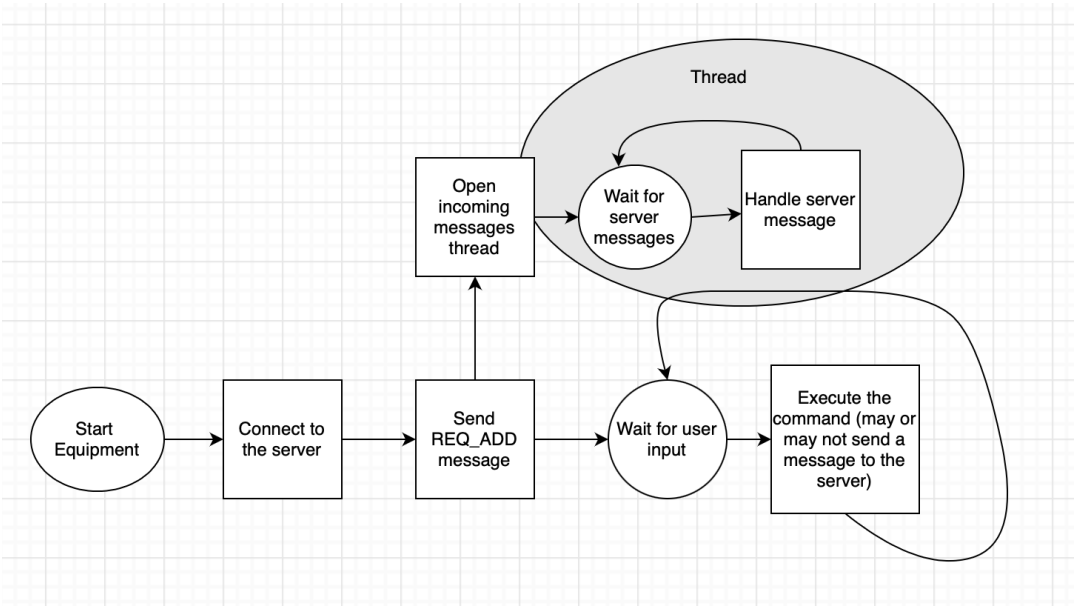

Figura 3. Fluxo de comunicação no lado do cliente

Assim que o equipamento é iniciado, ele automaticamente se conecta ao server e envia a mensagem solicitando a adição na rede (REQ_ADD). Apoós enviar a mensagem, ele abre uma thread, que ficará responsável por receber as mensagens do servidor a qualquer momento. A thread é importante, pois temos outro loop no código principal para receber o input do usuário, o que é uma ação bloqueante. Ter a thread nos permite receber mensagens do servidor ao mesmo tempo que o programa espera o input do usuário.

A primeira mensagem que o equipamento deve receber do servidor é a RES_ADD, informando o id atribuído a este, que será impresso na tela no formato "New ID {IDEq}". Logo em seguida o servidor irá transmitir o RES_LIST, informando o equipamento da lista de equipamentos que se encontram conectados no servidor, no momento. A especificação do projeto não especificou se o id do próprio equipamento será enviado. Por decisão de projeto, ele é enviado.

Após este ciclo inicial, o equipamento continuará com um loop no código principal esperando novos inputs do usuário e na thread, esperando mensagens do servidor.

O usuário pode realizar 4 ações:

1. Comando "list equipment": Este comando listará os equipamentos ordenados do menor id para o maior, sempre precedidos de um "0" caso menor que 10. Observação: o id do próprio equipamento não é impresso aqui
2. Comando "request information from {IDEq}": Este comando envia uma mensagem REQ_INF para o servidor, solicitando informação do equipamento com id idEq. O servidor recebe a mensagem, valida o id de origem e destino (e responde com um erro caso um deles sejam inválidos), encaminha a mensagem para o equipamento de id idEQ (que imprime "requested information"), recebe a resposta de idMsg RES_INF, e a transmite de volta para o equipamento solicitante, que imprime "Value from {IDEq}: {value}".
3. Comando "close connection": Envia uma mensagem REQ_REM(id do equipamento) para o servidor, que responde com o OK, e em seguida o cliente termina o programa.

4. Terminar o programa de forma forçada (desligar o computador ou encerrar o processo): nenhuma mensagem é enviada, mas o servidor detecta o encerramento da conexão e realiza a mesma ação de quando recebe o REQ_REM.

4.1. Métodos principais

- int main(int argc, char const* argv[]): Inicia a conexão, envia a mensagem REQ_ADD, cria a thread para receber mensagens e espera entradas do usuário para a realização de ações.
- void _executeCommand(char* command, size_t commandSize): delega a ação a realizada para a função certa, de acordo com o valor de "command"
- void _handleServerMessage(char* message): divide a mensagem em tipo de comando e argumentos. Logo em seguida, delete os argumentos para a função devida de acordo com o tipo de comando.
- void _sendMessage(char* idMsg, int originEqId, int destinationId, char* payload): recebe os parâmetros necessários para a formação da mensagem, concatena os relevantes para a mensagem de id idMsg, adiciona a quebra de linhas e envia a mensagem para o servidor.

5. Formato de Transmissão das mensagens

Mensagens de Controle					
Tipo	**Id Msg**	**Id origem**	**Id Destino**	**Payload**	**Descrição**
REQ_ADD	01	–	–	–	Mensagem de requisição de entrada de equipamento na rede
REQ_REM	02	$IdEQ_i$	–	–	Mensagem de requisição de saída de equipamento na rede, onde $IdEQ_i$ corresponde a identificação do equipamento solicitante.
RES_ADD	03	–	–	$IdEQ_i$	Mensagem de resposta com identificação $IdEQ_i$ do equipamento EQ_i
RES_LIST	04	–	–	$IdEQ_j$, $IdEQ_k$, ...	Mensagem com lista de identificação de equipamentos. Onde $IdEQ_j$, $IdEQ_k$, ... correspondem aos identificadores dos equipamentos transmitidos na mensagem.

Figura 4. Mensagens de controle
Fonte: Especificações do TP2

As mensagens a serem enviadas, tanto do cliente quanto do servidor, passam por uma função _sendMessage que recebe o id da mensagem, id de origem, id de destino, e payload. A figura 4 mostra algumas das mensagens que podem ser enviadas. Temos que os "_" indicam que o campo não é utilizado pela mensagem. O protocolo utilizado, separa os campos da mensagem por um espaço, e finaliza esta com uma quebra de linhas.

Exemplo: RES_ADD(04), seria codificada como:
"03 04\n".

Na mensagem **RES_LIST**, especificamente, os equipamentos presentes no servidor são enviados separados por uma vírgula, logo, um exemplo de comando aceito seria: "04 01,02,03,04,05,07". Já "04 01 02 03", por exemplo, sera um comando inválido.

6. Discussão

Este projeto teve objetivos educacionais, portanto, não foi desenvolvido para ser utilizado em cenários reais. Dito isso caso esta fosse a intenção, além do desenvolvimento de funcionalidades mais aplicáveis à indústria, algumas mudanças poderiam ser implementadas, como:

- Uso de um formato de transmissão de mensagem mais estruturado. Caso a mensagem fosse enviada em JSON, por exemplo, ela certamente seria maior, mas muito dificilmente o uso extra de recursos gerado por isso seria um problema nos dias de hoje. O envio da mensagem em JSON facilitaria o parsing das mensagens, já que a identificação dos atributos não dependeria da ordem e do id da mensagem.
- Verificações de segurança: O servidor aceita, por exemplo, que um equipamento solicite a remoção de outros.
- Verificações gerais: O servidor aceita, por exemplo, que um equipamento solicite informação do próprio equipamento. Apesar de não ser um grande problema, o usuário ou o código do equipamento certamente não teria tido aquela intenção, o retorno de um erro é mais apropriado.
- Uso de listas encadeadas para armazenar os equipamentos, de forma a poder remover o limite de 15, e deixar o limite ser a própria capacidade do servidor. Para isso, verificações dos recursos disponíveis deveriam ser feitas.

7. Conclusão

O desenvolvimento deste projeto foi um ótimo exemplo do porque que protocolos bem definidos são importantes para qualquer rede. Temos que as mensagens entre os equipamentos e servidor são codificadas e devem seguir um padrão rígido para que o sistema se comporte como esperado. Alguns detalhes na especificação do projeto, principalmente no que diz respeito ao formato de envio das mensagens, foram ambíguas (porém devidamente esclarecidas), e, caso não esclarecidas, poderiam levar a incompatibilidade de um cliente de um desenvolvedor funcionar com o servidor de outro desenvolvedor, e vice-versa.

Não houve dificuldades significativas no desenvolvimento do projeto. O maior desafio foi a compreensão integral das especificações. Fora isso, o projeto foi desenvolvido tomando como base um código já funcional de um sistema que funcionava com uma conexão única (TP1), a aprimorado para atender aos novos requisitos.