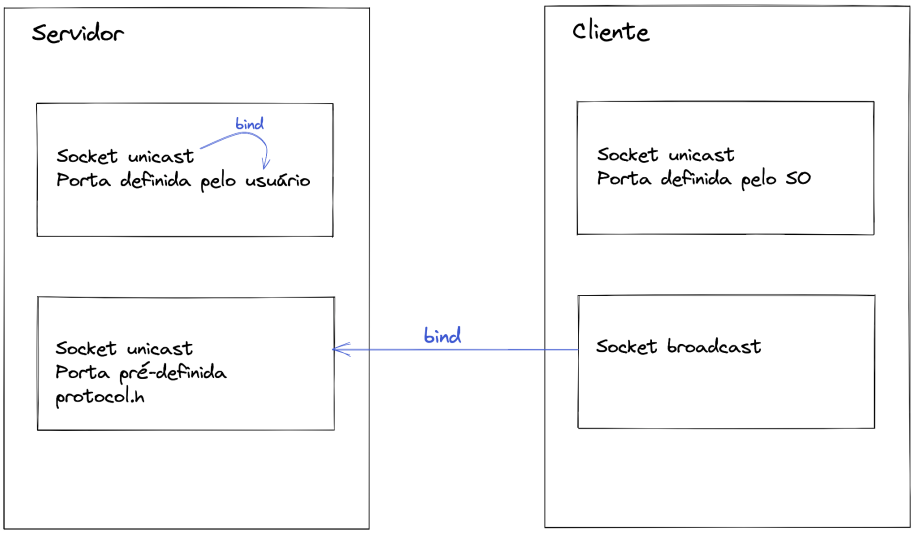
**Lívia Delgado de Almeida Carneiro - 2019054749**

## Introdução

Para implementação deste trabalho, utilizei **apenas** o protocolo **UDP** (por ele permitir transmissões broadcast).

Ambos os programas ***server*** e ***equipment*** utilizam 2 sockets, um utilizado para transmissões unicast e outro para transmissões broadcast. Essa solução visou solucionar problemas encontrados ao tentar utilizar ambas as formas de conexão em apenas uma porta.

Além disso, ambos os programas utilizam a abordagem de concorrência (multi-threaded) para conseguir lidar com múltiplas transmissões de diferentes tipos.

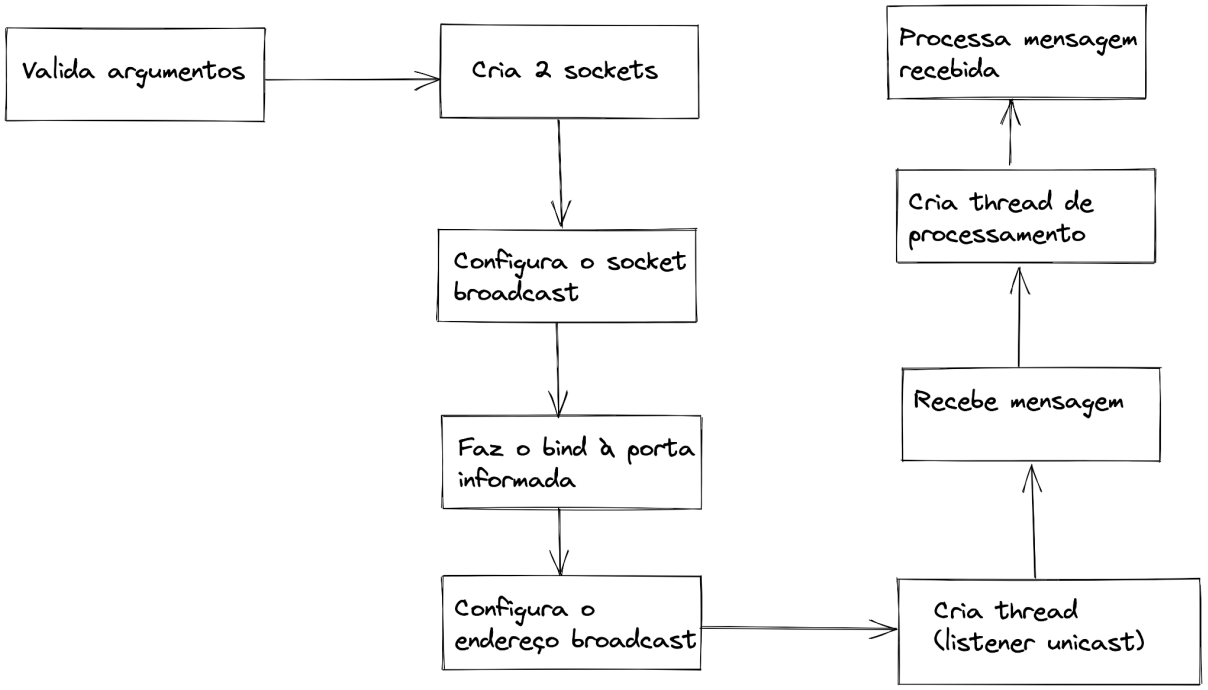



## Estruturação do código

- **Client.c:** arquivo responsável pela definição do Equipamento. Recebe instruções do terminal e mensagens enviadas pelo servidor (broadcast e unicast), processa mensagens e retorna os resultados.
- **Server.c:** arquivo responsável pela definição do Equipamento. Recebe mensagens enviadas pelos clientes (unicast), processa mensagens e retorna os resultados.
- **Control.c:** responsável pela estrutura de controle dos equipamentos pelo Servidor. Controle de equipamentos conectados e *source of truth* para orientações de conexão unicast (armazena os endereços de cada cliente conectado).
- **Protocol.c:** utilizado pelo Server e pelo Client, centraliza todos os métodos de conexão e interação utilizando o protocolo UDP.
- **Messaging.c:** utilizado pelo Server e pelo Client, centraliza as regras relacionadas às mensagens trocadas entre as aplicações.
- **Threads.c:** centraliza operações de criação de threads.
- **Utils.c:** funções e constantes utilitárias, utilizadas por todos os outros arquivos.

## Servidor

O servidor está organizado da seguinte maneira: valida argumentos, cria 2 sockets, configura o socket broadcast, faz o bind (do socket unicast) à porta informada pelo usuário, configura o endereço de broadcast para o qual enviará mensagens, cria uma thread UnicastListener (possibilita extensão para que ele inicie a comunicação por outros métodos), espera por uma mensagem, cria uma nova thread de processamento para cada mensagem recebida e processa a mensagem, vide diagrama abaixo.

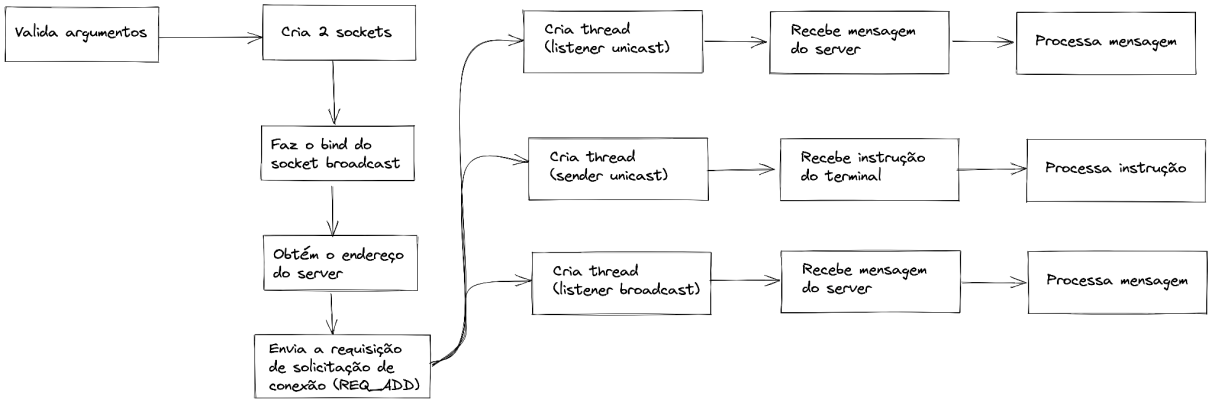



- Ele cria uma thread de processamento a cada mensagem recebida para que não seja interrompida a escuta por novas mensagens.
- Para cada mensagem processada, ele executa as ações conforme instruído na especificação.

**Equipamento**

O cliente está organizado da seguinte maneira: valida argumentos, cria 2 sockets, faz o bind do socket broadcast à porta adequada, obtém o endereço do servidor conforme parâmetros recebidos, envia a solicitação de conexão com o servidor, e cria 3 threads para processamento de mensagens e instruções.

Uma thread faz a função de Unicast Listener, em que ele reage a mensagens enviadas para a porta em que está conectado. Outra faz a função de Broadcast Listener, na qual ele ouve a porta em que as mensagens broadcast estão sendo enviadas. Por fim, a thread de Sender Unicast recebe comandos do terminal, os quais *podem* resultar em mensagens sendo enviadas por transmissão unicast para o servidor.



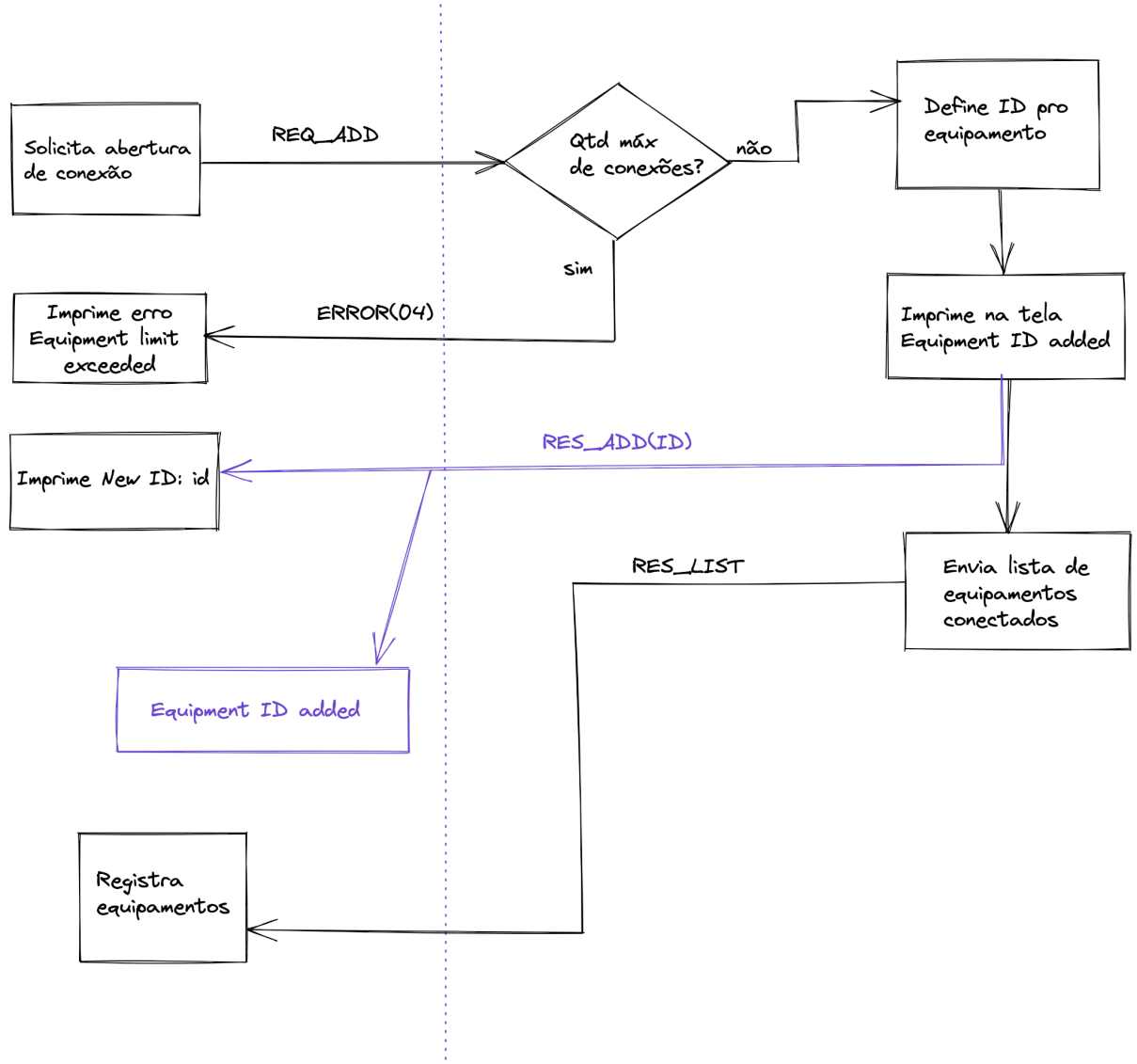

**Discussão**

Foram assumidos os seguintes pontos para desenvolvimento deste trabalho:

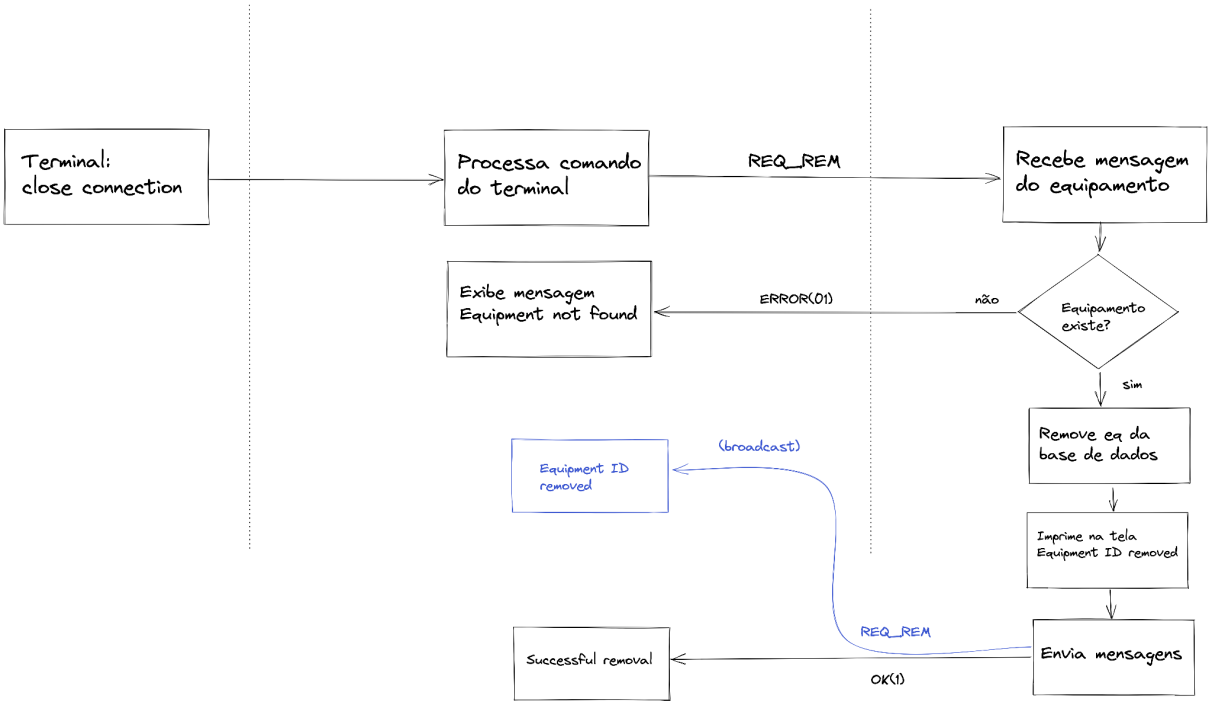
- Não é necessário fechar a conexão em qualquer outro caso que não seja o **close connection**
    - Resultados de erro, por exemplo, não encerram o programa.
- O comando **list equipment** lista todos os equipamentos atualmente conectados **com exceção do próprio id do equipamento**
    - Método *printListConnectedEquipments* do arquivo *client.c*
- Como estou utilizando **UDP**, que não é orientado a conexão, não consigo dizer se um cliente está desconectado (i.e. o processo do terminal foi encerrado) a menos que ele envie o comando de **close connection**. Ele continua registrado no servidor e em todos os outros equipamentos, contando também no número total de equipamentos.
- Não existe nenhum impeditivo para que, no fluxo de mensagens de dados, seja requisitado o dado do próprio equipamento por meio do terminal deste. Não foi implementada restrição no código, e ele funciona para esse cenário também.

**Fluxo das mensagens implementadas**

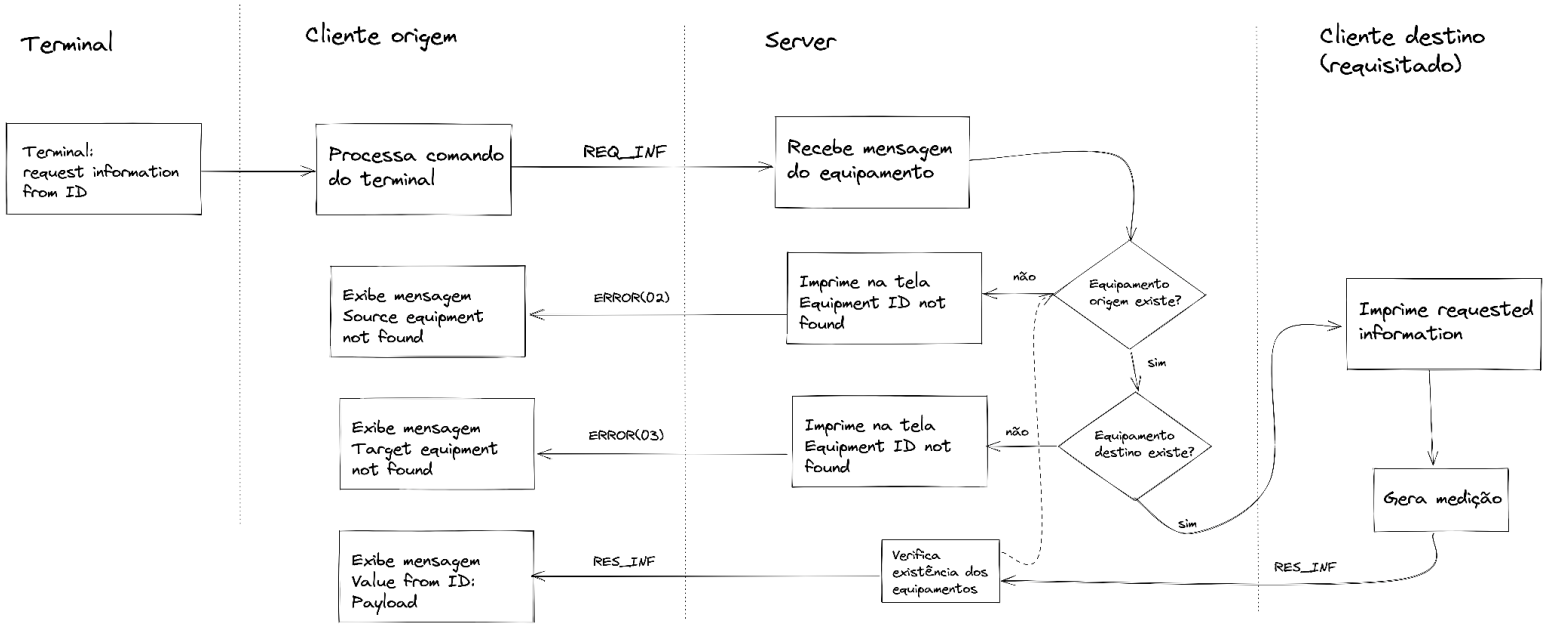
Abertura de conexão com o servidor



Fluxo de Fechamento de Conexão com o servidor




Fluxo das mensagens de dados



**Conclusão**

A principal dificuldade no desenvolvimento deste trabalho se deu pela transmissão broadcast e pela criação de threads para permitir o Non-blocking I/O.

Optei por utilizar **outra porta** para a comunicação broadcast após leitura de vários fóruns, materiais indicados e especialmente esses trechos do livro texto:

> *"Note also that a broadcast UDP datagram will actually be "heard" at a host **only if some program on that host is listening for datagrams on the port to which the datagram is addressed**."*

Tradução: "Note também que um datagrama UDP vai ser de fato "ouvido" por um host **somente se um programa naquele host estiver esperando por datagramas na porta para a qual o datagrama é endereçado**."

> *Note that a receiver program does not need to do anything special to receive broadcast datagrams **(except bind to the appropriate port)**.*

Tradução: "Note que um programa receptor não precisa fazer nada especial para receber datagramas broadcast (**exceto se conectar à porta apropriada**)".

Assim, entendi que, apesar de na especificação ser mencionada a criação de apenas um socket no cliente, não seria possível receber datagramas UDP por transmissão broadcast sem fazer o bind na porta adequada e tampouco receber por Unicast na porta broadcast.