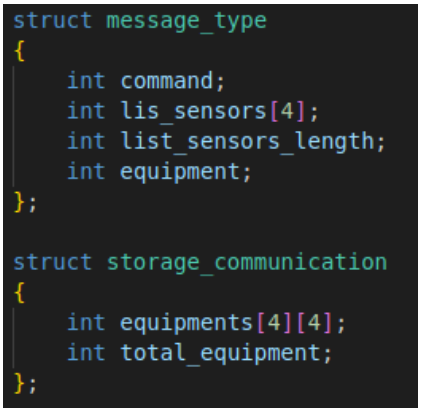
Gabriel Lopes Machado - 2018019516

ARQUITETURA DE COMUNICAÇÃO E ESTRUTURA DE DADOS

O desenvolvimento começa com o estudo do desenvolvimento em sockets em C, a partir do material disponibilizado foi possível dominar a arquitetura requisição resposta para um e para vários clientes (utilizando-se threads). Com o domínio da ferramenta a ser utilizada, o primeiro desafio foi adaptar o código obtido a partir do estudo em um formato que o cliente e o servidor ficassem em um loop infinito de troca de dados. Logo, com um esquema de troca constante de mensagens do cliente com o servidor, o próximo passo foi a definir como representar a troca de mensagens do cliente para o servidor e como armazenar o estado de cada sensor em cada equipamento



```
struct message_type
{
    int command;
    int lis_sensors[4];
    int list_sensors_length;
    int equipment;
};

struct storage_communication
{
    int equipments[4][4];
    int total_equipment;
};
```

Figura 1.

Na Figura 1. podemos observar as duas estruturas de dados utilizadas, são elas:

1. *message_type*: respresenta as informações necessárias para qualquer menssagen que é entregue para o servidor.
    a. *command*: inteiro que representa o comando para o servidor (0 - add; 1 - remove, 2 - lsit, 3 - read).
    b. *list_sensors*: nos comandos “add”, “remove” e “list” é possivel fazer uma consulta de mais de 1 sensor de um mesmo equipamento ao mesmo tempo, logo esse vetor vai conter os indices de cada sensor a se alterar em uma dada mensagem.
    c. *list_sensors_length*: a quantidade de sensores que vão ser consultados, ou seja, o tamanho de list_sensors.
    d. *equipment*: o índice que representa o equipamento a ser alterado.

2. storage_communication: o formato mais simples possível de se armazenar o estado de cada um dos 16 sensores possíveis e controlar a quantidade de sensores. (mesmo que existam 16 sensores distintos, apenas 15 podem ser instalados).
    a. *equipments*: vetor 2D onde a linha representa um equipamento e a coluna representa o sensor (preenchido com: 0 - não existe; 1 existe). Por exemplo, 0 índice [0][0] desse vetor representa um estado boleano do sensorId “01” do equipmentId “01”.
    b. *total_equipment*: A quantidade total de equipamentos já instalados.

VALIDAÇÃO DA MENSAGEM PELO SERVIDOR

O primeiro passo que o servidor deve tomar após receber uma mensagem qualquer de um cliente é verificar se ela se enquadra corretamente em um dos 4 formatos possíveis de mensagem. Para isso foi implementada a função *validate_and_create_message,* que tem como objetivo verificar se a string de entrada pode ser convertida para a estrutura *message_type* a partir da manipulação da mesma, onde seus parâmetros de entrada são:

- *char *buf*: string vinda do cliente.
- *struct message_type *message*: estrutura do mensagem a ser formada para futura manipulação do estado dos sensores.

Como o seu retorno é do tipo int, existem três valores distintos que a função retorna, são eles:

- -1: mensagem mal formada, a operação enviada pelo cliente não se enquadra no formato esperado.
- 0: a mensagem se enquadra em um dos quatro tipos de mensagem esperado
- 1: A operação é válida, porém ao menos um dos sensores não faz parte do conjunto especificado, nesse caso, o banco de sensores não é consultado e uma resposta do tipo **“invalid sensor”** é retornada.
- 1: A operação é válida, os sensores listados são válidos, porém o código do equipamento não é representado pelo sistema, nesse caso, o banco de sensores não é consultadoe uma resposta do tipo **“invalid equipment”** é retornada.

```
strcpy(aux, buf);
strtok(aux, " ");

if (0 == strcmp(aux, "add"))
{
    message->command = 0;
}
else if (0 == strcmp(aux, "remove"))
{
    message->command = 1;
}
else if (0 == strcmp(aux, "list"))
{
    message->command = 2;
}
else if (0 == strcmp(aux, "read"))
{
    message->command = 3;
}
else
{
    return -1;
}
```

Figura 2.

Na Figura 2. podemos observar um exemplo de manipulação da string de entrada *buf* utilizando-se de funções da biblioteca <string.h>. O primeiro passo é a cópia do conteúdo do buffer para uma variável auxiliar *aux*, logo em seguida, ela é separada pelo delimitador “ ” (espaço). Com a primeira *sobstring* obtida, basta comparar ela com um dos possíveis comandos, no caso de o

comando ser **“kill”** ou ser invalido o retorno de -1 faz com que a conexão com o servidor seja encerrada e o programa terminado.

Ainda existem outras manipulações bem mais complexas no código, mas que não vão ser tratadas neste texto por praticidade.

MANIPULAÇÃO DO BANCO DE SENSORES

Com a certeza de que a mensagem do cliente é válida e se encaixa de forma adequada em uma das operações possíveis, o servidor deve fazer a devida manipulação no estado dos sensores e retornar a mensagem adequada seguindo o tratamento de exceções estipulado nas especificações. Esse processo é implementado na função *update_bd_and_create_response* que faz todas essas operações baseada na mensagem recebida pelo servidor, ela tem como parâmetros de entrada:

- struct message_type *message: a mensagem determinada pela função anterior.
- struct storage_communication *storage: um ponteiro para o banco de dados que representa o estado de cada sensor.

O retorno da função é a string de resposta do servidor. Para termos uma noção desse processo observemos o seguinte trecho de código na Figura 3.:

```
response[0] = '\0';
switch (message->command)
{
case 0: // add
{
    if (storage->total_equipment + message->list_sensors_length >= 16)
    {
        strcat(response, "limit exceeded");
        break;
    }

    strcat(response, "sensor ");

    int number_already_exist = 0;
    int already_exists_vector[4] = {0, 0, 0, 0};
    for (int i = 0; i < message->list_sensors_length; i++)
    {
        if (storage->equipments[message->equipment][message->lis_sensors[i]])
        {
            already_exists_vector[i] = 1;
            number_already_exist++;
        }
        else
        {
            storage->equipments[message->equipment][message->lis_sensors[i]] = 1;
            storage->total_equipment++;
        }
    }
```

Figura 3.

Aqui temos o começo do código para executar o comando “add”. No primeiro *if* é validada a exceção da quantidade máxima de sensores possíveis. Já no *for* representado, para cada sensor passado pelo comando, é verificado se ele já não foi adicionado ao banco, nesse caso, o sensor

incorreto é anotado, caso contrário, o sensor é representado com um número 1 e o número total de equipamentos é atualizado.

DESAFIOS/DIFICULDADES/IMPREVISTOS

Uma diferença que pode ser notada entre a implementação adotada e a sugerida para ser incorporada em projetos do tipo é a representação do banco de sensores, que aqui foi uma *array* de duas dimensões estático. Por se tratar de uma aplicação industrial, que possivelmente vai rodar em um microcontrolador ou algum dispositivo de baixa capacidade de armazenamento, uma lista duplamente encadeada é mais recomendável, pois nesse caso, a memória ocupada é dinamicamente operada. Além disso, outro problema que a correção automática pode gerar é na execução do comando *"list",* pois, a partir dos exemplos disponibilizados, os sensores são listados na sequência que são adicionados ao equipamento, já com a implementação feita lista os sensores de forma ordenada com o Id.