Maria Júlia Berriel de Sousa **RA 117964**
Vinícius Andrade Frederico **RA 139229**

Projeto 02 - MC833

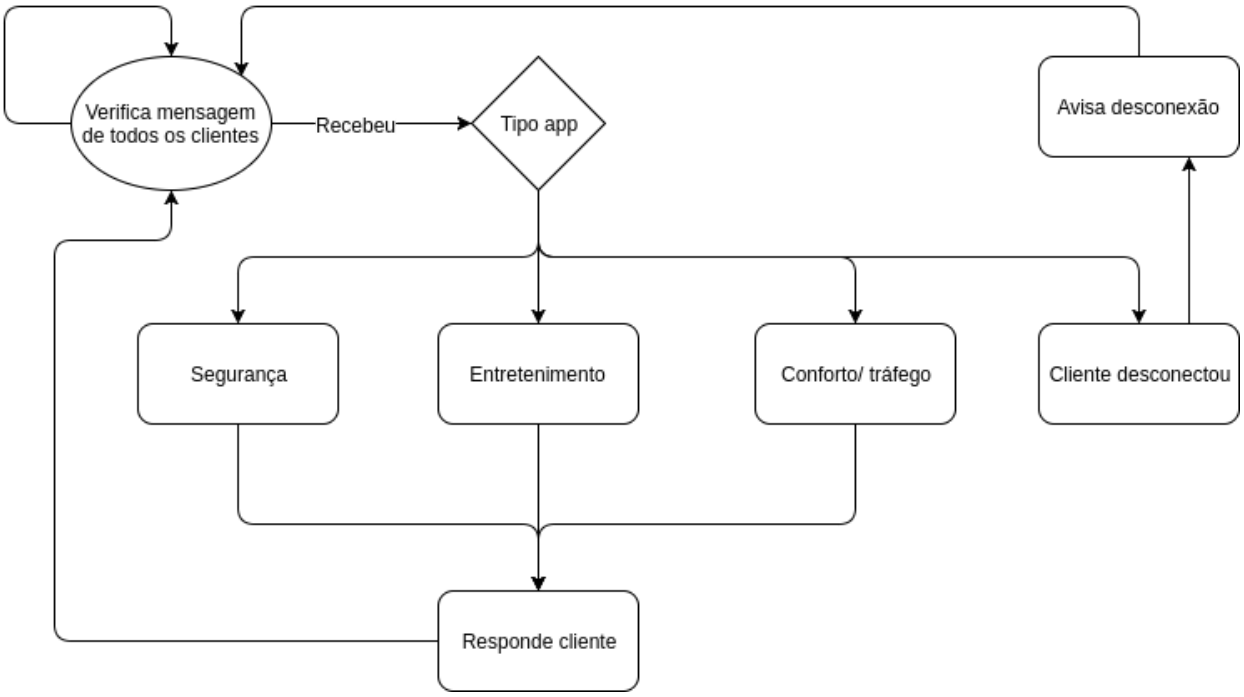
Implementação

Para a implementação desse projeto, foi priorizada a simplicidade para se ganhar na performance, especialmente na aplicação de segurança. Seguindo as especificações, o servidor pode atender diversos clientes e lida com três tipos de aplicações, segurança, entretenimento e tráfego. A aplicação de segurança tem prioridade sobre as outras duas, por isso é executada no servidor, enquanto as outras são executadas em um servidor remoto.

Servidor

O servidor foi implementado usando o protocolo UDP, assim é possível atender diversos clientes simultaneamente e rapidamente, como não há a necessidade de se criar threads ou processos para cada a cliente, a resposta pode ser enviada rapidamente. Isso é importante para a aplicação de segurança, que deve responder rápido para evitar acidentes.

O funcionamento do servidor pode ser visto no diagrama abaixo:



Como o servidor é para apenas um cruzamento, o número máximo de clientes simultâneos é pequeno, portanto não é preciso um servidor muito complexo que garanta que pacotes não serão perdidos por conta de algum overflow, portanto o protocolo UDP é uma boa escolha, pois diminui o número total de pacotes que trafegam pelo servidor.

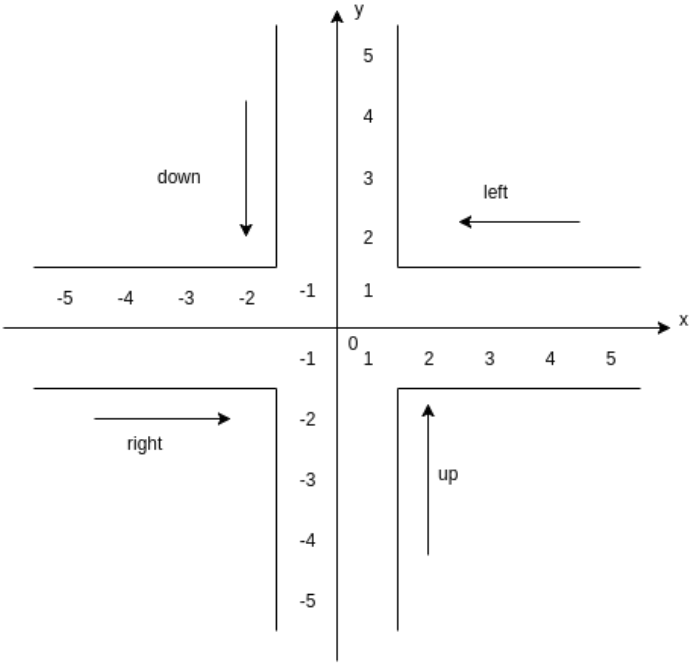
Para o caso de pacotes perdidos por má conexão, o cliente envia requisições em pequenos intervalos de tempo para que algum pacote perdido não tenha influência no comportamento geral do conjunto.

O funcionamento das aplicações de entretenimento e tráfego são simulados pelo servidor como se fosse a comunicação com outro servidor mais longe, por isso há um delay para a resposta para esses tipos de pacote.

Aplicação de Segurança

Esta aplicação é a que detecta que ação o carro que enviou uma requisição deve tomar, ou se houve uma colisão. Para isso o servidor armazena todos os carros perto do cruzamento, e faz um cálculo para saber quando cada carro estará no cruzamento. Se dois carros forem ocupar o mesmo espaço no cruzamento ao mesmo tempo, o servidor responde com a ação apropriada, diminuir a velocidade ou acelerar.

O cruzamento é representado internamente pelo servidor da seguinte maneira:



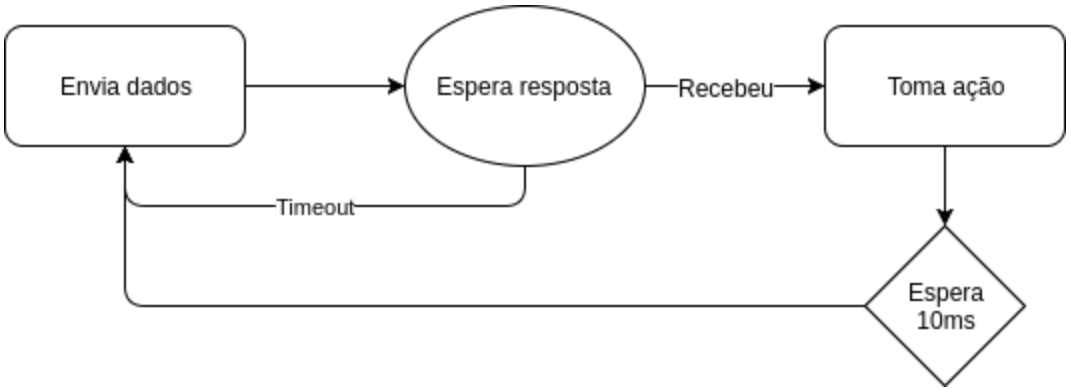
Os espaços no grid são discretos,sendo que as posições com *x=0* ou *y=0* nunca são usadas.

A resposta da aplicação é uma das seguintes mensagens, cada uma representada por um inteiro:

```
// Tipos de mensagens de resposta
#define MSG_HELLO               0
#define MSG_OK                  1
#define MSG_SLOW                2
#define MSG_ACC                 3
#define MSG_AMBULANCE           4
```

Cliente (Carro)

Quando um carro chega perto o suficiente do cruzamento, ele começa a se comunicar com o servidor, enviando um pacote inicial. Em seguida, procede enviando pacotes de segurança o sequencialmente, e de outros tipos quando necessário. O diagrama abaixo demonstra o funcionamento do cliente para a aplicação de segurança, já que é a única que implica na tomada de alguma ação pelo carro.



Os pacotes enviados pelo cliente tem sempre o mesmo formato, com números indicando cada campo. Para campos não numéricos, a conversão é feita durante o processamento.O uso de números ao invés de strings deixa o pacote mais enxuto para ser transmitido.

O formato do pacote é o seguinte:

```c
typedef struct car_packet
{
    int type;           //tipo de pacote
    unsigned int ts;    //timestamp, inicia em 0
    int pos_x;          //posição x no grid
    int pos_y;          //posição y no grid
    int vel;            //velocidade
    direction dir;      //direção
    int size;           //tamanho do carro
} car_packet_t;
```

O tipo de pacote pode ser um dos seguintes:

```c
// Tipos de pacotes
#define PACKET_T_HELLO          0
#define PACKET_T_SECURITY       1
#define PACKET_T_ENTERTAINMENT  2
#define PACKET_T_TRAFFIC        3
```

O timestamp é sempre um número inteiro, as posições seguem o mesmo padrão do grid, a velocidade é dada por unidades no espaço por unidades de tempo, e o tamanho é quantas posições no grid o carro ocupa.

Assume-se que a velocidade é alterada instantaneamente. O servidor assume que a velocidade de cliente foi alterada conforme instruído.

Respostas do servidor:

```c
// Tipos de mensagens de resposta
#define MSG_HELLO               0
#define MSG_OK                  1
#define MSG_SLOW                2
#define MSG_ACC                 3
#define MSG_AMBULANCE           4
```

Se um carro é instruído à frear, sua velocidade é reduzida para metade se velocidade > 1. Caso contrário, velocidade inicial = 1, velocidade final = 0.
Analogamente, acelerar significa dobrar a velocidade (ou setar como 1 caso seja 0), porém essa instrução só é dada caso o carro tenha reduzido a velocidade no passado.

Testes

Foram testados 3 casos: (cada linha indica início de um carro, índice dos carros começando de 0)

1) o caso simples, de carros com velocidade e tamanho 1, com colisão, onde o carro 1 é instruído a ficar parado, até que possa voltar a sua velocidade sem causar colisão.
   Entrada (cars.in):
   #tipo | timestamp | posição X | posição Y | velocidade | direção | tamanho
   1 0 1 -7 1 0 1
   1 0 -6 1 1 3 1

   Saída em `cars.out`

2) Carros com velocidade 2 e tamanho 3, com colisão, onde como há forma de reduzir a velocidade, no instante 1, o carro 1 é instruído a reduzir a velocidade, para evitar a colisão, e, no instante 6, como não há mais risco de colisão, é instruído a acelerar.
   Entrada (cars2.in):
   #tipo | timestamp | posição X | posição Y | velocidade | direção | tamanho
   1 0 1 -20 2 0 3
   1 0 -16 -1 2 3 3

   Saída em `cars2.out`

3) Carros com velocidade e tamanho diferente
   Entrada (cars3.in):
   #tipo | timestamp | posição X | posição Y | velocidade | direção | tamanho
   1 0 1 -20 2 0 3
   1 0 1 -20 2 0 5
   1 0 -16 -1 2 3 3
   1 0 -17 -1 2 3 1

   Saída em `cars3.out`

Nos 3 casos o sistema foi capaz de evitar a colisão através da manipulação de velocidade.

Compilação:

Para compilar o servidor, rodar `make`.
Para compilar o cliente, rodar `gcc cliente.c utils.c -I . -o cliente`.

Execução:

Para rodar o servidor, rodar `./server`.
Para compilar o cliente, rodar `./cliente -h <host> -i <arquivo_entrada>`.