- Guilherme Lucas da Silva RA: 155618
- Felipe Dal Mas Eulalio RA: 155299

Projeto 2 - Monitoramento de Tráfego

1. Introdução e Objetivos

Após realizar uma série de experimentos ao longo da matéria, notamos que é extremamente simples realizar a comunição entre dois processos presentes em máquinas diferentes, por meio de protocolos de rede muito bem estabelecidos ao longo dos anos. Usando esses padrões, tivemos a responsabilidade de implementar um sistema que monitora um cruzamento, notando possíveis colisões e tentando evitá-las, por meio de instruções dada aos automóveis, que eram clientes. Essas instruções eram como "acelere" ou "freie". Caso a colisão não consiga ser evitada, então será enviada uma mensagem chamando uma ambulância para o local do acidente.

2. Resumo das Aplicações

Nesta seção explicaremos resumidamente o funcionamento das aplicações envolvidas no projeto.

2.1 Detalhes gerais

Algumas coisas foram assumidas para a implementação do projeto, dentre elas está o fato de que as acelerações são instântaneas e que as velocidades dos veículos são constantes, exceto se o servidor envia alguma ordem para o veículo alterar sua velocidade. Além disso, foi considerado que os véiculos possuiam posições que variam de **-50 a 50** em duas possíveis direções, vertical e horizontal, as velocidades de **-5 a 5** e o tamanho de **1 a 5**.

2.2 Cliente

Nesse projeto, adotamos cada cliente como um automóvel, que possui as seguintes características: **id, tamanho, velocidade, posição, timestamp e direção**. Assim, obtidas essas informações via linha de comando, é montada uma mensagem com essas características e então essas informações são enviadas para o servidos decidir se vai acontecer alguma colisão e tomar alguma decisão sobre isso. Após realizar o envio da mensagem, o cliente espera para receber as instruções do servidos, sobre se deve **acelerar ou parar**. Caso a mensagem diga para o automóvel parar, ele para e epsera por 2 segundos parado, até tentar voltar a velocidade anterior. Vale ressaltar também, que no primeiro recebimento de mensagem pelo cliente, este define seu **id**, baseado na informação dada pelo servidor.

Além disso, o cliente atualiza sempre a posição do veículo e verifica quando o mesmo sai do quadro de simulação, encerrando a simulação. Outro caso no qual a simulação é encerrada é quando o veículo se envolve em uma colisão.

2.3 Servidor

No caso do servidor, uma thread principal fica esperando as conexões, utilizando um **select** para gerenciar as conexões com os clientes. Além disso, o servidor possuia um *array* no qual guardava as últimas informações sobre cada veículo, como **id, tamanho, velocidade, posição, timestamp e direção**. Além disso, a cada nova mensagem de segurança, são verificadas as possíveis colisões que o veículo que enviou os dados possa ter e, caso achada alguma, ambos os carros eram avisados da colisão, sempre dando um *delay* de 10 ms para enviar as resposta. O servidor também recebe mensagens de conforto/tráfego e entreterimento, para as quais dá apenas um *delay* de 100 ms e faz um *echo* na mensagem recebida.

2.4 Script de Teste

Script desenvolvido em Python é extremamente simples e realiza a simples tarefa de executar o servidor e alguns clientes com características aleatórias baseadas nos limites que definimos no trabalho. Além disso, tem a função de matar todos os processos que esse mesmo script iniciou.

3. Implementação

3.1 Cliente

Para a implementação do código do cliente, foram necessárias algumas definições feitas no header `definitions.h` . A estrutura mais importante dessa parte do código é a **ClientMessage** estrutura feita para enviar as informações do servidor ao cliente. Esse possui os seguintes atributos:

- type: tipo da mensagem (conforto, segurança, entretenimento)
- id: identificador único para o automóvel que está enviando a mensagem
- size: tamanho do automóvel em questão
- speed: velocidade do automóvel
- position: posição que o automóvel se encontra no mapa
- direction: informação para saber se o carro está se movendo na vertical ou horizontal
- message: mensagem que está enviando

Após obtidos esses campos pela linha de comando, onde o usuário passa os valores como argumentos do **cliente** a estrutura de mensagem é povoado com essas informação e enviada para o servidor, bloqueando até receber a mensagem do mesmo. Para esse processo são usados as seguintes funções:

- socket: função necessária para a criação de um socket, onde serão enviadas as mensagens

- gethostbyname: obtem o endereço do servidor a partir do nome
- connect: estabelece conexão entre o cliente e servidor
- send: envia mensagem para o servidor
- read: recebe mensagem do servidor

3.2 Servidor

Assim como para o cliente, o server fez uso de várias estruturas definidas no `definitions.h`. Algumas foram muito utilizadas, como a **ClientMessage**, que recebia a mensagem do cliente e que já foi esmiuçada anteriormente, a **Car**, que guarda as informações de um veículo, tais como:

- type: tipo da mensagem (conforto, segurança, entretenimento)
- id: identificador único para o automóvel que está enviando a mensagem
- size: tamanho do automóvel em questão
- speed: velocidade do automóvel
- position: posição que o automóvel se encontra no mapa
- direction: informação para saber se o carro está se movendo na vertical ou horizontal

A estrutura **IndexMessage** é utilizada para passar o **id** de um cliente e a mensagem recebida do servidor para a thread que irá responder a mensagem.

Dessa forma, o servidor foi implementado de uma forma que existe uma thread principal que fica escutando todas as conexões e, a cada nova mensagem recebida, cria uma nova thread que irá verificar o tipo da mensagem, fazer as operações necessárias e responder ao cliente. Para controlar o acesso ao vetor de carros, foi utilizado um lock, a fim de que nenhuma informação fosse acessada ou escrita ao mesmo tempo.

Além disso, foram utilizadas as seguintes funções para fazer a comunicação com os clientes:

- socket: função necessária para a criação de um socket, onde serão enviadas as mensagens
- bind: atribui um endereço ao socket, ao qual ele responderá quando chegar dados para esse endereço
- listen: permite ao processo escutar em um dado socket, ligando o processo ao socket
- accept: aceira uma nova conexão
- read: recebe uma mensagem do cliente
- send: envia uma mensagem para o cliente
- select: permite ao programa monitorar diversos sockets Retorna o número de sockets que estão sendo monitorados. Se ocorrer um erro, retorna -1
- FD_SET: adiciona um socket a lista de sockets a serem monitorados
- FD_CLR: remove um socket da lista de sockets a serem monitorados
- FD_ISSET: checa se algo ocorreu no socket master. Se sim, é um novo cliente se conectando

Com relação à detecção de colisões, foi feito um algoritmo no qual era calculado quanto tempo cada veículo demoraria para chegar ao cruzamento. A esse valor era descontado o tempo desde a última

mensagem enviada. Feito isso, era comparado se o veículo chegaria ao cruzamento enquanto o outro ainda estaria cruzando, ou seja, considerando o tamanho do veículo, se houvesse tempo hábil para evitar a colisão, os carros envolvidos eram avisados. Caso ambos os veículos estivessem chegando ao centro, essa seria uma colisão inevitável, que acionaria a mensagem de *ambulância*. Toda essa lógica se encontra no arquivo `collisionChecker.c`.

4. Testes

Criamos um simples método de testar o software de maneira automatizada. O `teste_script.py` lança o servidor e os clientes com valores aleatórios para testar o projeto. Para testar, basta dar o comando `make`. Feito isso, serão exibidas na saída principal a mensagem recebida pelo servidor do cliente e mensagens em caso de possíveis colisões ou colisões inevitáveis.

5. Resultados e Conclusões

Para observar o tráfego na rede, foi utilizado o Wireshark. Com o auxílio da ferramenta, foi possível observar os pacotes e também o atraso no envio dos mesmos. Dessa forma, abaixo estão os gráficos obtidos para testes realizados com 5, 10 e 20 clientes.

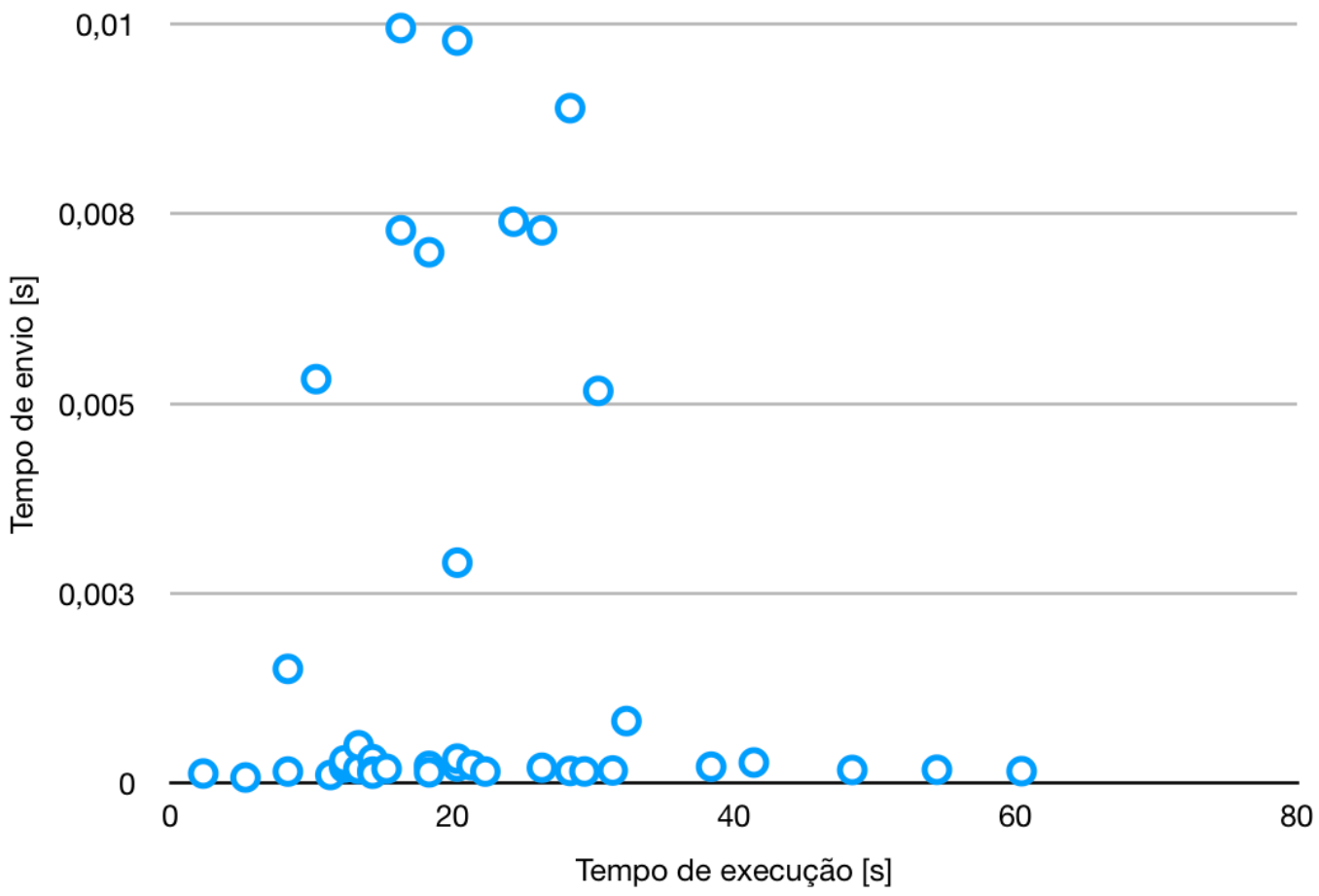

Figura 1: 5 clientes

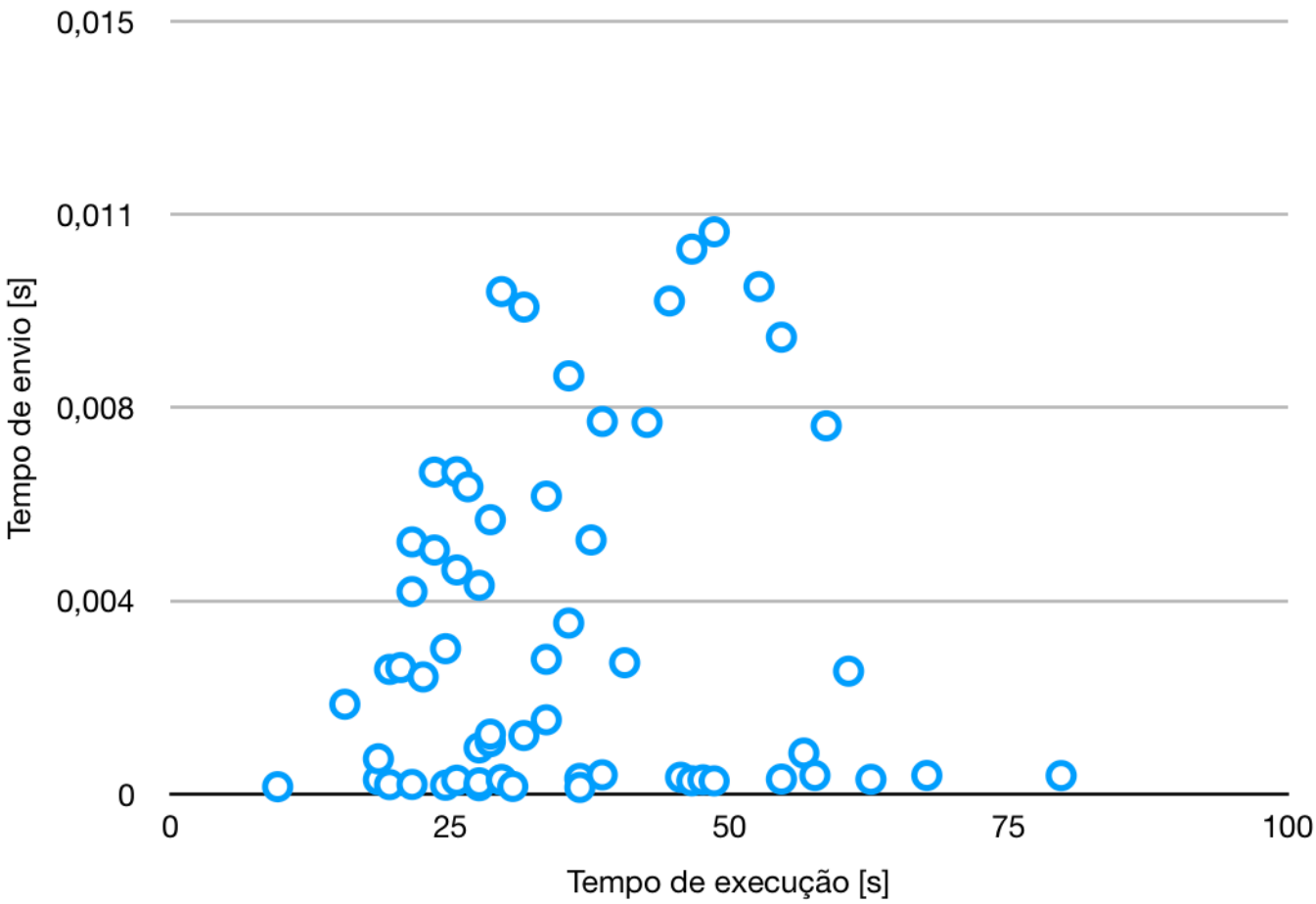

Figura 2: 10 clientes

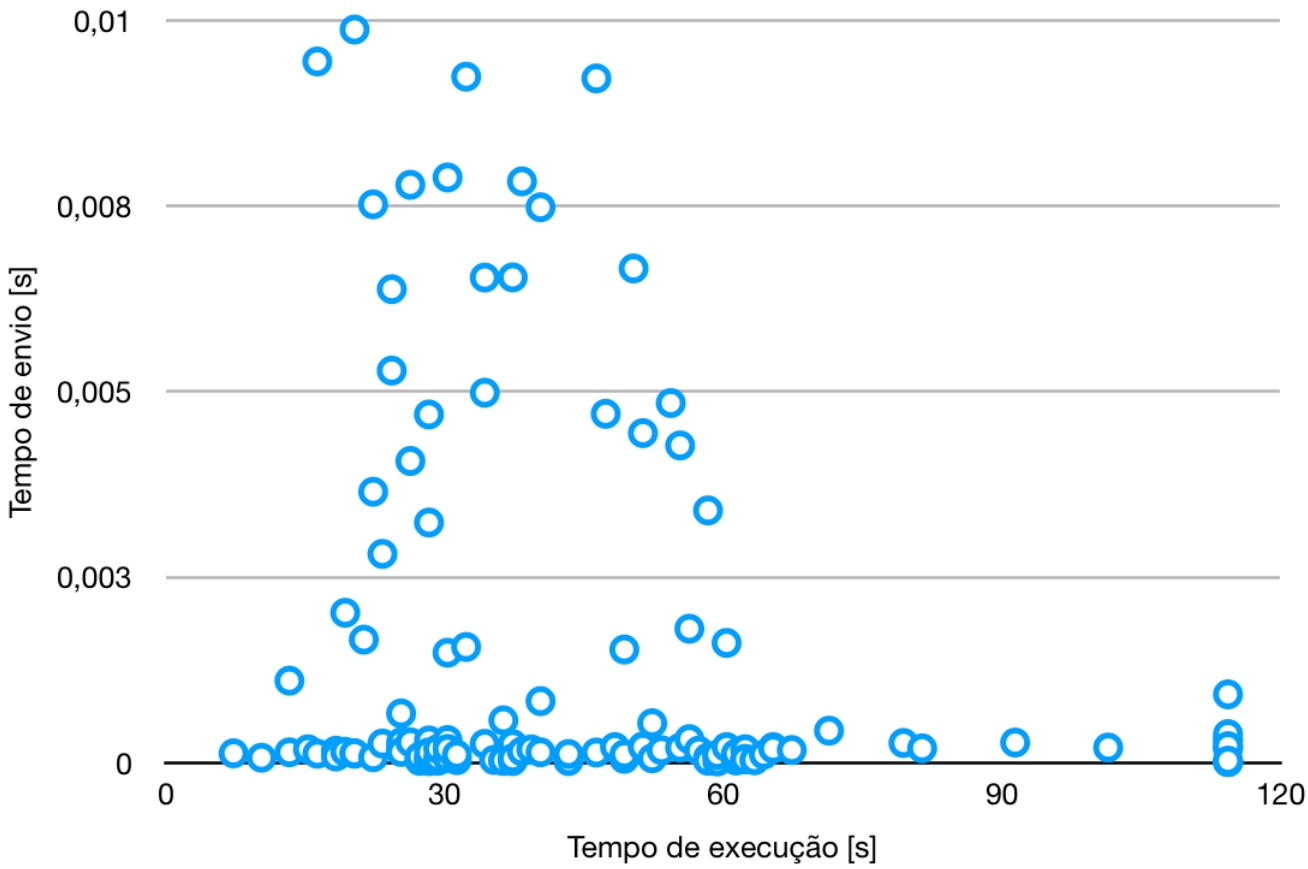

Figura 3: 20 clientes

É possível observar que os atrasos são maiores no meio da execução, quando existem mais clientes conectados e, consequentemente, mais pacotes sendo trocados com o servidor.

Com relação ao preditor de colisões, pode-se dizer que houve um resultado até que satisfatório. Em algumas situações, apesar de prever as colisões, os carros não conseguiam receber a mensagem de colisão, talvez por alguma perda, e depois de um tempo acabavam colidindo. Mas fora essas raras exceções, as colisões foram bem encontradas e evitadas quando possível.