MC833 - Relatório Final

Lucas Padilha - 119785
Marcos Kobuchi - 136823

Projeto 2: Detectar potencial colisão entre automóveis num cruzamento.

Suposições:

- As vias tem mão dupla;
- Todos os automóveis tem relógio sincronizado;

Carro

Um carro tem três camadas de aplicações, e elas serão explicadas nos tópicos posteriores. São elas: segurança, entretenimento e conforto/tráfego. Suas respectivas implementações podem ser definidas com qualquer conteúdo.

Para testar o carro, bem como todas suas aplicações, basta compilar o carro utilizando o seguinte comando:

```
gcc car.c -o car
```

E rodar o programa station utilizando:

```
./car
```

Para rodar o programa, deve **necessariamente** estar em execução os servidores relativos a camada de segurança e a camada de tráfego. O servidor relativo a camada de entretenimento não é essencial para o funcionamento da aplicação como um todo, como será explicado posteriormente.

Camada de Segurança

Foram assumidas as seguintes suposições:

- Colisões traseiras não ocorrem;
- Mudanças de velocidade são instantâneas
- Servidor emite ordens para mudar a velocidade.

A camada de segurança é um servidor que sabe onde estão todos os carros, e assim calcular se ocorrerá acidentes caso não interfira no fluxo dos carros.

O problema de controle de cruzamento consiste em prevenir que os carros colidam no cruzamento de ruas. Para isso, criamos a camada de segurança. Essa camada é um servidor que recebe de todos os carros conectados, suas informações básicas, como por exemplo: sua posição, sua velocidade, sua direção e seu tamanho. Todos esses parâmetros são importantes para o problema. E o tornam bastante desafiador.

O servidor funciona da seguinte forma, uma vez que o carro entre na pista, ele se cadastra em uma lista de carros. De tempos em tempos o carro envia para o servidor, suas informações básicas. Quando o servidor, recebe uma nova informação ele processa todas as informações que tem e executa o algoritmo de proteção.

O algoritmo consiste em uma solução gulosa, não-ótima para o problema. Uma vez que o servidor tem todos os dados dos carros, ele faz uma previsão de quando esses carros vão entrar no cruzamento (time_in) e quando esses carros vão sair do cruzamento (time_out) usando a posição e a velocidade do carro. Uma vez processado todas esses tempos, o servidor então, ordena os carros por tempo de entrada, para executar um algoritmo de FIFO (First In First Out). Então, percorre o vetor ordenado, e para cada carro:

Se o time_in do carro conflitar com o time_out_corrente. Manda uma mensagem de desacelerar o carro até não haver conflito. Atualiza o time_in e time_out do carro e o time_out_corrente será o time_out do carro.

Se não houver conflito, é mais simples, faz um cálculo de qual máximo que pode acelerar o carro para não ter conflito e manda uma mensagem para acelerar o carro. Atualiza o time_in e o time_out do carro e o time_out_corrente será o do time_out do carro.

Desta forma, todos os carros vão se encaixando no cruzamento, recebendo as mensagens de ordem para acelerar ou desacelerar o carro.

O maior impeditivo e o que ainda pode gerar colisão é a latência para recebimento dos pacotes, pode ser que as mensagens não cheguem a tempo para impedir o acidente. Mas na maioria dos casos, se a latência for pequena, esse algoritmo previne as colisões.

Para testar a aplicação de segurança, basta compilar a camada de segurança da estrada utilizando o seguinte comando:

```
gcc road-server.c -o server
```

E rodar o programa server utilizando:

```
./server
```

Esse programa, ao terminar de iniciar, ficará aguardando por conexões, e exibirá em seu *display* quando novos motoristas conectam ou desconectam dele.

```
---------------------------------
Waiting for new connection...
---------------------------------
Accepted new connection!
at address  : 0.0.0.0
at port     : 4391
---------------------------------
Client terminated!
at address  : 127.0.0.1
at port     : 8129
```

Um carro irá enviar ao servidor a cada intervalo de tempo suas informações, como velocidade, posição e tamanho.

```
---------------------------------
Received new message!
at address  : 127.0.0.1
at port     : 8129

Car info:
Speed       : 4
Position    : 80
Length      : 1
X           : -1
Y           : 0
```

Assim que o servidor recebe essa informação, ela envia a todos os carros uma ordem, dizendo para ele assumir uma determinada velocidade, otimizando assim o tempo para que todos possam passar pelo cruzamento.

Note que a segurança é um fator essencial para o bom funcionamento da aplicação. Portanto, se o carro não consegue conectar com o servidor de segurança, ele encerra.

Camada de Entretenimento

A camada de entretenimento se baseia em uma estação de rádio. Essa estação é um servidor (implementado em radio-station.c) que manda pacotes UDP para todos os clientes conectados a ele.

Uma estação de rádio abre um arquivo chamando entertainment.txt, que contém textos a serem transmitidos. Pode ser, por exemplo, uma música, propagandas, um programa de rádio, uma seita satânica, ou todos esses programas alternados. Esses áudios (representados na forma de texto) são transmitidos mesmo que não exista ninguém sintonizado -o que pode acontecer com estações de rádios reais. Um motorista, portanto, ao sintonizar, ele não escutará desde o início do programa, mas aquilo que está sendo transmitido no momento.

Para que um motorista sintonize nessa rádio, ele inicializa uma thread responsável para esse gerenciamento. No mundo real, essa thread representa num carro todo o seu sistema de rádio, composto por caixas de som, antenas, frequências (portas), etc.

No momento que essa thread é inicializada, ela busca se sintonizar (conectar) com a rádio. Caso não consiga, ela espera por alguns instantes para tentar novamente. Similarmente, se a rádio está sintonizada mas a estação de rádio sofre um corte de energia (cai), a thread espera por alguns instantes e tenta sintonizar novamente. A comparação no mundo real é que, enquanto o rádio não sintoniza, ele emite um chiado, até ter sucesso.

Temos dois fatos que merecem ser observados nessa abstração:

- A distância entre a estação de rádio e motorista interfere diretamente na latência. Essa distância é representada pelo caminho entre servidor e cliente, conectados através de roteadores. Porém, isso pouco importa, pois a camada de entretenimento não é um ponto crítico para o motorista, servindo a ele apenas como distração enquanto viaja ou está preso no trânsito.
- O sinal pode interferir na *confiabilidade* da informação. Rádios distantes, por exemplo, podem ter seu sinal enfraquecido se a distância for muito grande ou se houver obstáculos no caminho. Isso é representado diretamente pela perda de pacotes na rede. Como foi citado anteriormente, a comunicação nessa camada se dá através de sockets UDP. Foi escolhido essa modalidade pois é uma aplicação que exige pouca ou nenhuma confiabilidade na informação.

Para testar a aplicação de entretenimento, basta compilar a estação de rádio utilizando o seguinte comando:

```
gcc radio-station.c -o station
```

E rodar o programa station utilizando:

```
./station
```

Esse programa, ao terminar de iniciar, ficará aguardando por conexões, e exibirá em seu *display* quando novos motoristas conectam ou desconectam dele.

```
    ---------------------------------
    Waiting for new connection...
    ---------------------------------
    Accepted new connection!
    at address  : 0.0.0.0
    at port     : 4391
    ---------------------------------
    Client terminated!
    at address  : 127.0.0.1
    at port     : 8129
```

Similarmente, o cliente exibirá mensagens enquanto tenta se sintonizar na rádio, exibindo no seu display sua presente situação:

```
    ---------------------------------
    Starting Entertainment Layer...
    ---------------------------------
    Maybe Radio Station is off. Sorry!
    ---------------------------------
    Started Entertainment Layer!
    at address  : 127.0.0.1
    at port     : 8129
```

Quando conectado, exibirá no *display* sempre que receber informação da rádio.

```
    ---------------------------------
    Entertainment Layer:
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras id porta orci,
    eget pharetra tellus. P
```

Note que, para uma representação mais fiel da realidade, temos que diminuir o intervalo de disparo de uma informação para o mais próximo de zero. Em nossa abstração, optamos por enviar cinco caracteres a cada duzentos milisegundos.

Para não exibir apenas cinco caracteres sempre que recebe uma informação do servidor, armazenamos esses caracteres num buffer, e exibimos seu conteúdo sempre após um número definido de clocks ou quando o buffer está quase cheio. Após a exibição, limpamos o buffer. Observe que o conteúdo no buffer pode variar entre carros por diversos fatores, portanto cada motorista pode estar visualizando momentos distintos, porém muito próximos, da informação transmitida.

É interessante notar que durante os testes, invariavelmente, a informação recebida não era igual a mensagem enviada, ficando levemente distorcida. Ou seja, houve perdas de pacotes. Novamente, isso não é problema, visto que pequenas distorções não são problemas.

Para finalizar, concluímos essa seção com uma pequena observação, o fato de que a quantidade de informação transmitida aumenta linearmente com o número de motoristas sintonizados. Isso se deve pois, numa conexão multicast, o servidor deve enviar diretamente a informação para cada cliente. Isso não tem representação com o mundo real, visto que para uma pessoa sintonizar uma rádio, ele precisa apenas sintonizar a frequência, não acarretando em diferenças para a estação de rádio.

Camada de Tráfego*

A camada de tráfego é representada pelo farol do carro, que deve ser ligada ou desligada de acordo com as condições atuais de visibilidade. Num ambiente nublado, ou escuro devido a noite, o motorista pode (deve!) optar por ligar seus faróis.

Para que um motorista enxergue as condições de visibilidade, ele deve olhar para o lado de fora de seu carro, verificar o ambiente e em seguida tomar uma decisão de ligar ou não seus faróis. Em nossa abstração, o "ambiente" será representado por um servidor chamado "weather", o "olhar para fora" será um pacote enviado ao servidor, "verificar o ambiente" será um pacote recebido do servidor contendo as condições de visibilidade. Tendo as condições de visibilidade, podemos então tomar a decisão de ligar ou desligar os faróis (ou mantê-los no mesmo estado).

O ambiente, ao ser iniciado, determinará uma visibilidade inicial, sendo representada por um valor aleatório entre zero a dez, onde zero é visibilidade péssima e dez visibilidade ótima. No decorrer do tempo, essa visibilidade será melhorada ou piorada com iguais chances, e, claro, dentro de seus limites inferiores e superiores.

Para testar a aplicação de tráfego, basta compilar o ambiente utilizando o seguinte comando:

```
gcc weather.c -o weather
```

E rodar o programa weather utilizando:

```
./weather
```

Esse programa, ao terminar de iniciar, ficará aguardando por conexões, e exibirá em seu *display* quando novos motoristas conectam ou desconectam dele.

```
    ---------------------------------
    Waiting for new connection...
    ---------------------------------
    Accepted new connection!
    at address  : 0.0.0.0
    at port     : 4391
    ---------------------------------
    Client terminated!
    at address  : 127.0.0.1
    at port     : 8129
```

A cada determinado período de tempo, atualizará sua condição atual de visibilidade. Note que a próxima condição será no máximo um acima ou um abaixo da condição atual, portanto, não há variações bruscas na visibilidade. Por exemplo, não há chances da visibilidade estar num valor ótimo (dez) e no momento seguinte estar num valor péssimo (zero).

```
    ---------------------------------
    Weather Condition:
    ( 0 <-> 10 ) 8
```

Similarmente, o cliente exibirá mensagens quando estiver tentando se conectar ao serviço, exibindo no seu display sua presente situação:

```
    ---------------------------------
    Starting Traffic Layer...
    ---------------------------------
    Started Traffic Layer!
    ---------------------------------
    Maybe Traffic is off. Try again!
```

A visibilidade é um fator essencial para o bom funcionamento do automóvel. Portanto, caso não consiga se conectar, a aplicação encerra.

Ao verificar a visibilidade, o motorista pode tomar a decisão de acender ou desligar os faróis. Determinamos como boa visibilidade valores acima de três:

```
    ---------------------------------
    Traffic Layer:
    Conditions are good. Headlights are off.
    ---------------------------------
    Traffic Layer:
    Conditions are bad. Headlights are on.
```

Para a comunicação dessa camada, optamos por utilizar TCP, visto que um motorista, a não ser que tenha debilidade visual, conseguirá enxergar as condições do ambiente onde se encontra.

* Optamos por descartar a camada de conforto, devido a dificuldade de encontrar aplicação real utilizando redes. Pensamos em serviços, como falar ao celular, mas por causa do aumento de acidentes causado pelo seu uso, bem como o recente aumento no valor das multas, optamos por descartar essa opção.

Testes, Execução e Análise de Tráfego

Para a análise de tráfego, utilizamos o wireshark, ferramenta que permite monitorar o envio e recebimento de pacotes.

Para os testes, iniciamos as aplicações *server* e *weather*, as duas aplicações essenciais para o bom funcionamento do veículo, como explicado anteriormente. Em seguida, iniciamos uma primeira aplicação *car*, para que ele se conecte as duas aplicações, podendo localizar no gráfico 1 como sendo o instante inicial zero.

Podemos ver que o tráfego se mantém relativamente constante, até o tempo 50s, quando adicionamos outro carro no sistema. Podemos observar que há um pico no tráfego de pacotes TCP, correspondente a inicialização das conexões entre carro e aplicações. O mesmo se observa quando adicionamos um terceiro carro, no tempo 63s.

Vemos que, a partir da adição de novos carros, a troca de pacotes torna-se mais constante. Isso se deve ao fato do servidor precisar comunicar para mais carros suas novas velocidades.

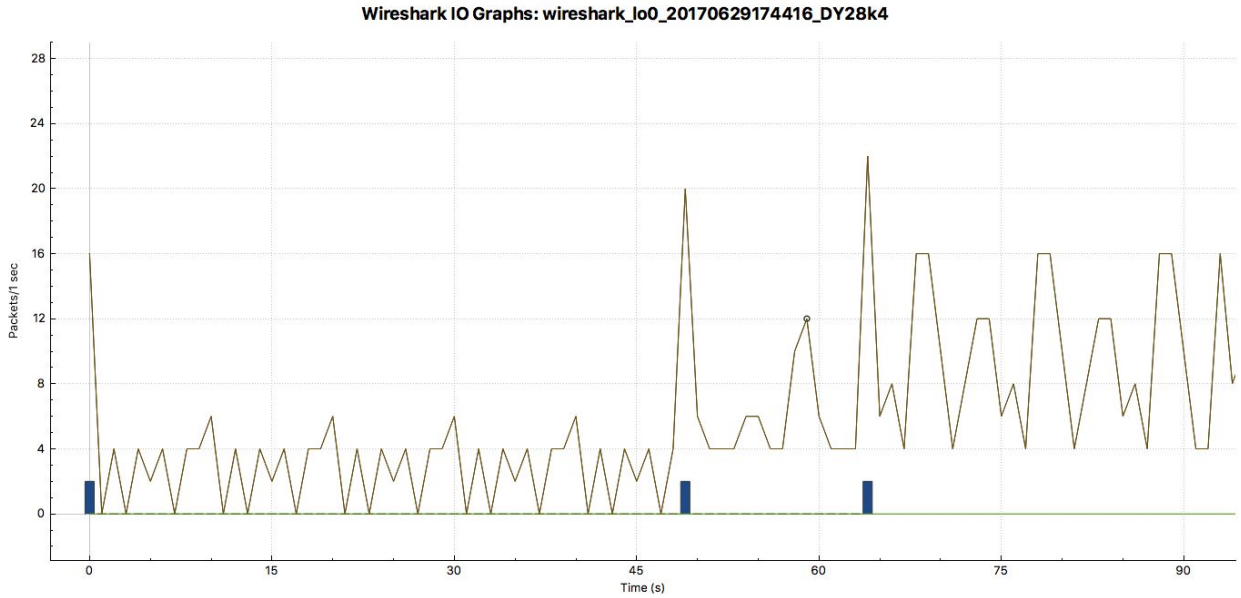

Gráfico 1. Camada de segurança e tráfego ativas. Teste com três carros.

Agora, vamos analisar quando temos apenas um carro, mas com o serviço de entretenimento ativo. No tempo 22s, ligamos a estação de rádio, e observamos que há um grande aumento no número de pacotes transportados, passando a triplicar o valor inicial. Isso era esperado, pois a estação de rádio envia pacotes de informação a cada 0.2 segundo.

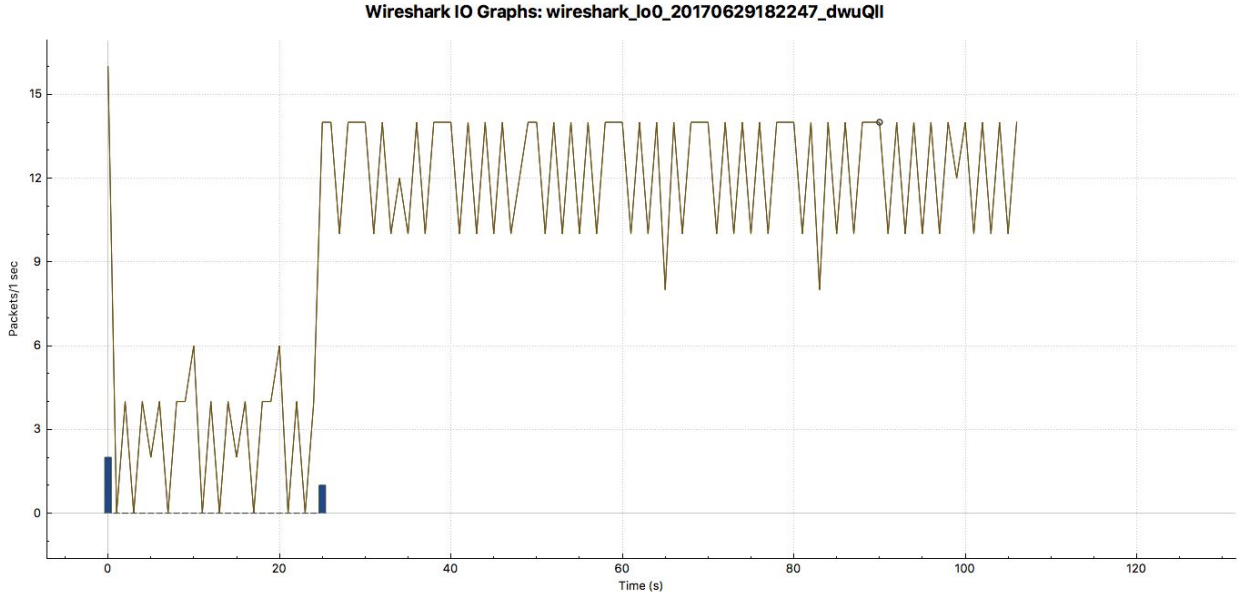

Gráfico 2. Camada de segurança, tráfego e entretenimento ativas. Teste com um carro.

Agora, vamos analisar o caso em que temos três carros, com a camada de entretenimento ativa. Podemos ver que o tráfego de pacotes é alto, tendo movimentado entre trinta e quarenta pacotes por segundo.

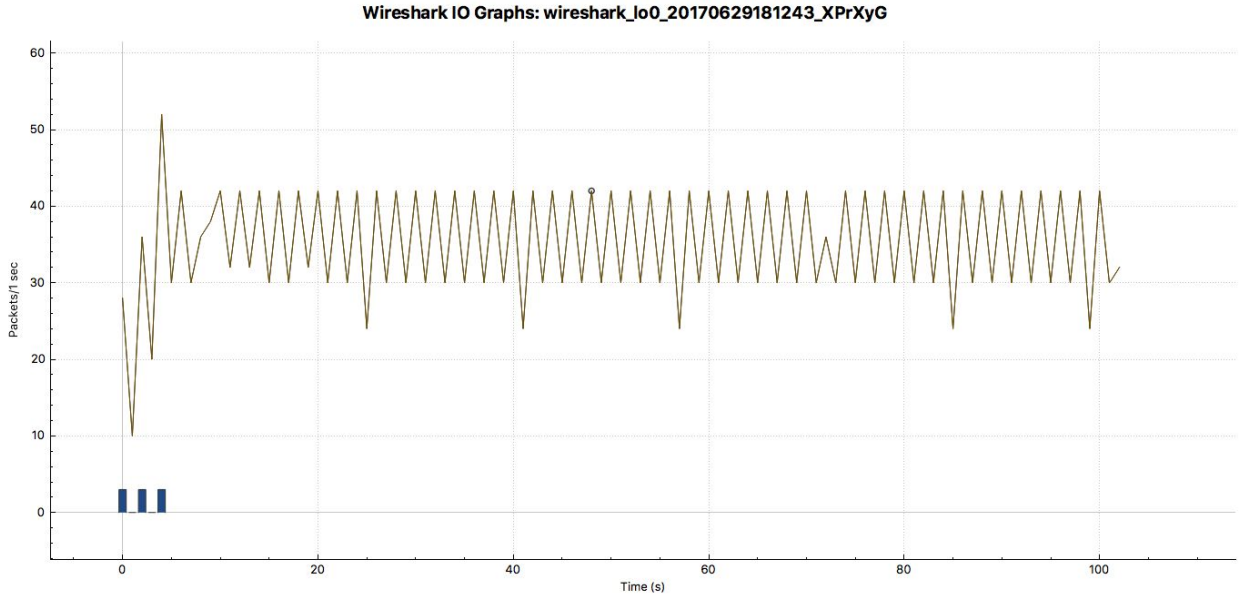

Gráfico 3. Camada de segurança, tráfego e entretenimento ativas. Teste com três carro.

Para o último teste, vamos observar o que acontece quando temos 6 carros, com todas as aplicações funcionando. Temos resultados bem próximos do que esperávamos, de acordo com os três testes anteriores.

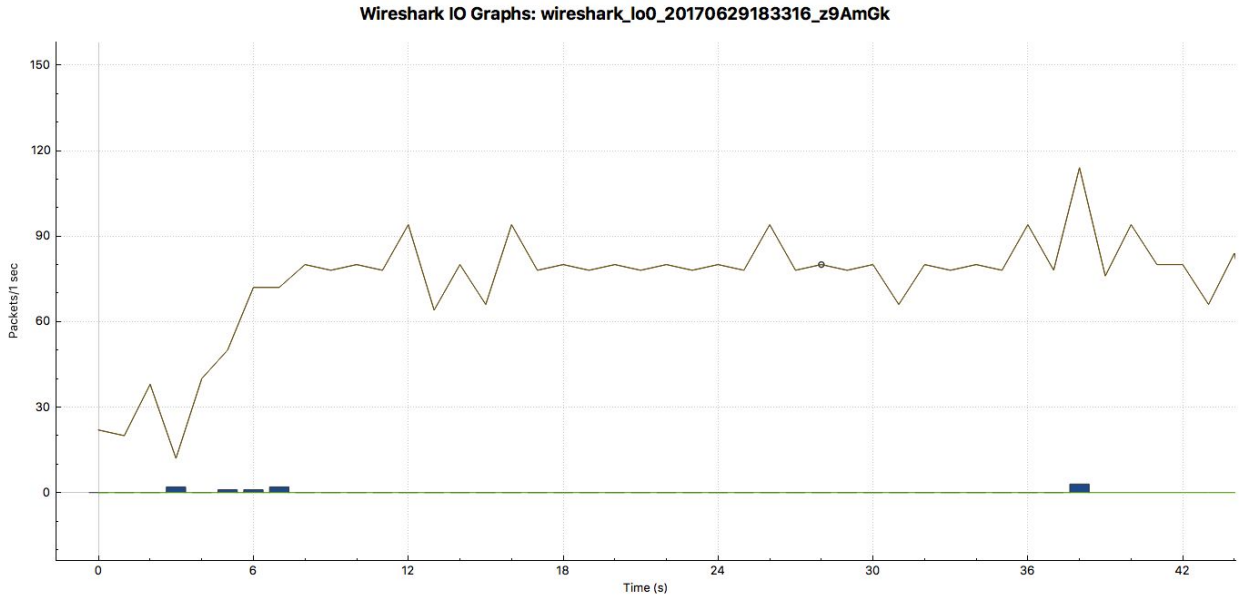

Gráfico 4. Camada de segurança, tráfego e entretenimento ativas. Teste com seis carro.

Conclusão

O projeto se mostrou desafiador, ao mesmo tempo em que permitiu aprender sobre comunicações em redes a partir do desenvolvimento do código e da observação do volume de tráfego que cada simulação gera na rede. Assim, é muito importante analisar e discutir quais são as opções de implementação disponíveis, verificar quais são suas vantagens e desvantagens, e assim decidir qual é a melhor a ser utilizada, de acordo com sua aplicação.