Relatório do projeto 2: Gerenciamento e Cruzamento

Isadora Sophia Garcia Rodopoulos *
Matheus Mortatti Diamantinos †
Luiz Fernando Bittencourt‡

Abstract

O objetivo do trabalho foi a implementação de um gerenciamento de trânsito e cruzamento, analisando o trade-off e a performance do algoritmo desenvolvido, além de seu impacto no mundo real.

Neste projeto, foi implementado um sistema cliente-servidor onde cada cliente representa um carro atravessando um cruzamento de mão dupla e o servidor fica responsável por processar três tipos de requisitos dos carros: **segurança**, **entretenimento** e **tráfego**.

Os pacotes de segurança são os responsáveis por pedir informações de trânsito, mandando a posição e velocidade do carro. O servidor, ao receber este pacote, calcula se uma colisão ocorrerá ou se já ocorreu, mandando um comando ao carro que fez a requisição com uma das seguintes ações:

- acelerar;
- freiar;
- chamar Ambulância.

1. Métodos

Os métodos utilizados neste projeto podem ser separados em duas categorias: **comunicação** e **detecção de colisão**. A seguir, cada categoria é detalhada.

1.1. Comunicação

O estabelecimento da comunicação entre cliente e servidor é feita utilizando o protocolo TCP, de modo que os pacotes cheguem garantidamente nas duas pontas da comunicação. Contudo, a troca de mensagens pode ter uma duração maior, fazendo com que os comandos de trânsito demorem mais para serem executados.

*RA 158018, Instituto de Computação, Universidade de Campinas, Unicamp. **Contact**: isadorasophiagr@gmail.com

†RA 156740, Instituto de Computação, Universidade de Campinas, Unicamp. **Contact**: matdiamantino@gmail.com

‡MC833, Instituto de Computação, Universidade de Campinas, Unicamp. **Contact**: bit@ic.unicamp.br

1.1.1 Cliente

O cliente realiza requisições ao servidor de dois tipos: Segurança e Outros, onde Segurança diz respeito a pacotes que pedem informações de trânsito e Outros são os pacotes de conforto e tráfego, com prioridade mais baixa. As prioridades são definidas de acordo com a latência de processamento e comunicação, sendo definidas como ¿ 100ms para as prioridades mais baixas e ¡ 10ms para as prioridades mais altas. Deste modo, é possível receber informações de trânsito com mais frequência e com menos atraso, obtendo uma confiabilidade maior nos carros autônomos.

Para simular tal prioridade, foi definida uma variável de tempo que nos diz o quanto tempo é necessário esperar antes de mandar outro pacote, dependendo do tipo deste:

Code 1. Demonstração do código para o envio de pacotes

```
if (time_passed(last_sent_security,
        my_car.cur_time) >= latency(SECURITY)) {
    /* send security packet */
}

if (time_passed(last_sent_other,
        my_car.cur_time) >= latency(OTHER)) {
    /* send any other packet */
}
```

1.1.2 Servidor

O servidor utiliza a estrutura `select` implementada na atividade **2.2** para se comunicar com diversos clientes ao mesmo tempo, de modo a calcular possíveis colisões para cada um dos clientes que estão trafegando pelo cruzamento. A estrutura funciona olhando para cada socket de comunicação de cada cliente ativo e verificando se alguma mensagem chega de algum deles. Se sim, verifica o tipo da mensagem. Se a mensagem for do tipo *segurança*, verifica colisão para algum dos clientes ativos e manda os comandos necessários aos carros que estão sujeitos a colisão. Se for do tipo *outros*, simula o processamento desta requisição processando uma função que roda por um tempo fixo sem realizar nenhuma atividade significativa, e então retorna um pacote sem informações úteis.

1.2. Detecção de colisão

Para realizar a detecção de colisão, primeiro discutimos a estrutura de dados utilizada para definir um carro.

Code 2. Estrutura utilizada para a descrição de um carro

```
typedef enum { LOW=9, HIGH=100 } Latency;
typedef enum { SECURITY=1, ENTERTAINMENT=2,
               COMFORT=3, OTHER=4 } Type;

typedef struct {
    struct timespec cur_time;
    int64_t break_time;
    int32_t command, size;
    int32_t x, y,
            vx, vy,
            dirx, diry;

    Type type;
} Car;
```

Nesta estrutura, temos o timestamp do carro (`cur_time`), o tempo em que o carro precisa ficar parado em caso de comando de freio (`break_time`), o comando que ele deve realizar (`command`) e variáveis que representam a posição (`x`, `y`), direção e velocidade (unidades por segundo) do carro.

Cada carro atualiza sua própria posição toda vez que o contador do respectivo cliente passa de 1 segundo. Então, a função `update_car()` é chamada.

Code 3. Código para a atualização de um carro

```
/* * update car for /n/ iterations */
void update_car_n(Car *car, int n) {
    car->x += n*car->vx;
    car->y += n*car->vy;
}
```

Cada cliente recebe um arquivo de entrada com as seguintes informações:

1. Quantidade de updates para as informações do carro;
2. Tamanho do carro;
3. Posição x;
4. Posição y;
5. Velocidade x;
6. Velocidade y;
7. Tempo de duração total.

E a cada intervalo de tempo definido nesta entrada, lê a próxima velocidade definida. Após os inputs terminarem, se necessário, uma velocidade padrão é definida para o carro.

No lado do servidor, então, ao receber um novo pacote de segurança do cliente, possíveis colisões com os diversos carros da pista são verificados a partir dos métodos a seguir.

Inicialmente, são atualizadas as posições dos carros que estão conectados a partir do timestamp da última mensagem recebida.

Code 4. Código para atualização dos carros

```
/* update cars */
for (i = 0; i < size; i++) {
    int64_t elapsed = since_now(cars[i].cur_time);
    clock_gettime(CLOCK_REALTIME, &cars[i].cur_time);
    update_car_n(&cars[i], elapsed);
}
```

Isto garantirá que o servidor saiba onde os outros carros estão mesmo se estes não mandaram mensagens a ele. Contudo, os carros podem ter mudado a velocidade sem o servidor saber e, portanto, isto é apenas uma previsão de onde os carros podem estar no tempo atual.

Então, é verificado para cada dupla de carros se os carros i e j já se colidiram ou se estão a colidir.

1.3. Colisão

Para detectar se a colisão ocorreu, observe a figura abaixo:

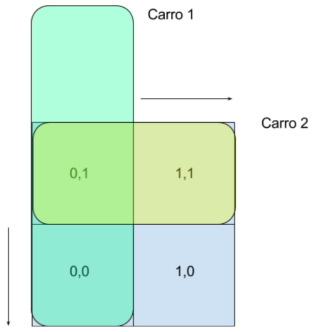

Figure 2. Modelo de visualização dos veículos

Podemos ver na figura que podemos verificar a colisão no ponto (0, 1) verificando se este ponto está dentro do intervalo de tamanho dos dois carros ao mesmo tempo. Para isso, podemos observar que o intervalo [(x1, y1), (x2, y2)] que representam os pontos da frente e da traseira do carro precisam conter o ponto (0, 1), conforme a imagem. Logo, generalizando, foi implementado a verificação abaixo:

Code 5. Algoritmo de detecção de colisão (1)

```
/* check if already collided! */
if (source.vy > 0)
    cond_s = source.y >= y
        && source.y <= y + source.size;
else
    cond_s = source.y <= y
        && source.y >= y - source.size;

if (target.vx > 0)
    cond_t = target.x >= x
        && target.x <= x + target.size;
else
    cond_t = target.x <= x
        && target.x >= x - target.size;

if (cond_s && cond_t) {
    *car1 = i;
```

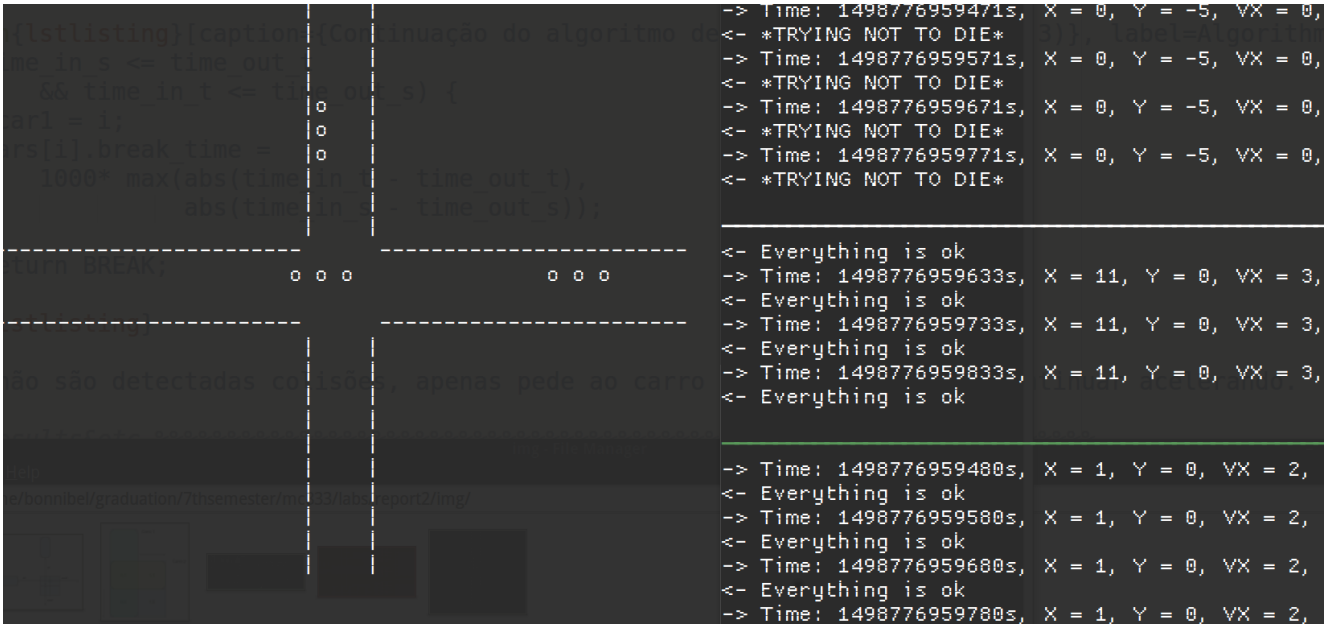

Figure 1. Exemplo de funcionamento do projeto. À esquerda, o servidor recebe o sinal de cada carro e, à direita, a resposta da comunicação dos clientes.

```
    *car2 = j;
    return AMBULANCE;
}
```

Observe que, dependendo da direção do carro, temos uma verificação diferente. Por exemplo, para o carro que está andando no eixo y com velocidade positiva, verificamos se a frente do carro já passou do ponto P e se a traseira ainda não passou. Assim, sabemos que o carro ainda está passando no cruzamento e podemos detectar se houve uma colisão entre dois carros. Se houver, mandamos o comando para chamar a ambulância aos dois carros que estão envolvidos no acidente.

Então, se não houve acidente, precisamos verificar se um acidente irá ocorrer, baseado no tempo que cada carro vai demorar para entrar e sair do cruzamento.

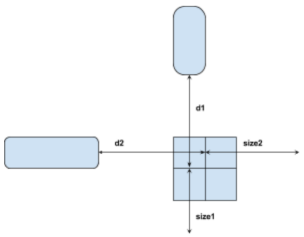

Figure 3. Modelo de cálculo de colisão dos veículos

Conforme a imagem acima, precisamos saber o tempo que demora para o carro 1 andar a distância d1 e o tempo que demora para ele andar a distância d1 + size1 (tamanho do carro). Deste modo, podemos verificar se os intervalos de tempo para o carro 1 e o 2 entrar e sair do cruzamento de intersectam e, então se os carros irão se colidir quando entrarem no cruzamento.

Code 6. Continuação do algoritmo de detecção de colisão (2)

```
/* source */
int dy = y - source.y;
int64_t time_in_s =
    DIVZERO(dy, source.vy);

dy = (y + sign(source.vy)*source.size)
        - source.y;
int64_t time_out_s =
    DIVZERO(dy, source.vy);

/* target */
int dx = x - target.x;
int64_t time_in_t =
    DIVZERO(dx, target.vx);

dx = (x + sign(target.vx)*target.size)
        - target.x;
int64_t time_out_t =
    DIVZERO(dx, target.vx);
```

Foi implementado o cálculo do tempo utilizando a equação:

$$\frac{\Delta S}{V} = \Delta T \tag{1}$$

...que nos retorna o tempo necessário para andar ΔS unidades andando com velocidade V.

Então, se os intervalos se intersectam, mandamos para um dos carros o comando de freio junto com um intervalo de tempo no qual ele deve ficar parado. Este intervalo é

calculado verificando o imaior intervalo de tempo entre o carro 1 e o 2 entrarem e sairem do cruzamento.

Code 7. Continuação do algoritmo de detecção de colisão (3)

```
if (time_in_s <= time_out_t
        && time_in_t <= time_out_s) {
    *car1 = i;
    cars[i].break_time =
        1000* max(abs(time_in_t - time_out_t),
                  abs(time_in_s - time_out_s));

    return BREAK;
}
```

Caso não são detectadas colisões, apenas pede ao carro que chama a função continuar acelerando.

2. Resultados e discussão

Para a validação da corretude do programa, foram executados diversos testes de fim a determinar a efetividade do nosso sistema de trânsito desenvolvido.

Para validar a etapa que detecta a colisão entre os carros - e, mais importante, evitar que eles se colidem - foi utilizado o recurso visual do servidor. O processo de teste baseou-se na instanciação de diferentes arquivos de entrada para cada carro e, a partir disso, foi realizada a observação visual do mapa do servidor. Logo, os testes permitiram que validássemos os algoritmos desenvolvidos ao longo do trabalho.

Por exemplo: foram executados testes para verificar se ocorria alguma colisão e, se ocorresse, se o servidor notificava corretamente. A partir desse caso, o cálculo para a sinalização de espera (freio) de um carro foi obtido a partir do experimento visual de diferentes resoluções do problema.

De uma forma geral, a etapa visual foi feita para validar a corretude do algoritmo e propor algumas otimizações ao sistema desenvolvido, dado diferentens parâmetros (como velocidade, cfrequê) para cada carro.

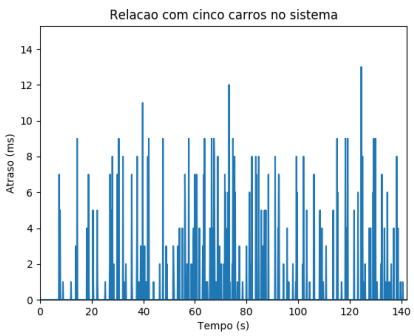

Figure 4. Gráfico de atraso para um sistema com cinco carros.

Em seguida, são avaliados aspectos técnicos do canal de comunicação entre o cliente e servidor. O primeiro caso de teste analisa a quantidade de tempo de *delay* que o servidor

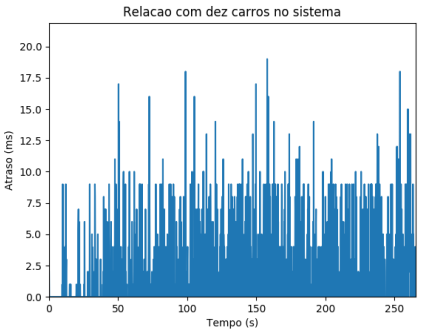

Figure 5. Gráfico de atraso para um sistema com dez carros.

demora para enviar a resposta, em relação do número de carros no sistema. É evidente que, quanto maior o número de carros, o servidor sofreria atraso na resposta - dado que todo o processamento do servidor ocorre de forma serial. Os gráficos acima ilustram o levantamento desses dados.

No caso de apenas um carro no sistema, o atraso em milissegundos era zero (ou seja, o atraso era desprezível). Logo, não foi disposto gráfico para esse caso.

O agravante dos valores dos atrasos dependia de fatores como velocidade dos carros e o clock do sistema - caso os carros se movimentassem lentamente, os valores de atraso não interferiam na prevenção de acidentes - isto é, os valores de atraso tornavam-se desprezíveis.

3. Ideias finais

De uma forma geral, o funcionamento do sistema foi relativamente estável, e evitou que acidentes ocorressem. Entretanto, se o algoritmo se escalasse a milhares de clientes, mudanças na infraestrutura teriam de ser feitas de modo a otimizar o processamento e evitar que ocorressem atrasos. Uma vez que atrasos acarretam em um número maior de acidentes.

Algoritmos de sincronização paralela seriam capazes de otimizar o funcionamento do servidor, por exemplo. Assim, no caso de um número de carros < 1000, os resultados foram satisfatórios.