COMPUTAÇÃO ESCALÁVEL
Relatório A1

Autores: Ana Carolina Erthal, Bernardo Vargas, Cristiano Larréa, Felipe Lamarca, Guilherme de Melo e Paloma Borges
Docente: Thiago Pinheiro de Araújo.

Escola de Matemática Aplicada
FGV EMAp

Rio de Janeiro
2023.1

Sumário

+

1 Introdução

O presente projeto baseia-se na elaboração de um sistema de ETL (*Extract-Transform-Load*) de um sistema de monitoramento de rodovias. Para tanto, foi necessário o desenvolvimento dos seguintes programas:

- Um mock que possa simular a entrega de dados ao sistema por uma rodovia, desenvolvido em *Python*;
- Um programa que realize o ETL dos dados, desenvolvido em *C++*.

1.1 Considerações

Para o entendimento do projeto, o grupo definiu as seguintes considerações além das definidas nos requisitos do trabalho:

- Uma instância do mock (isto é, um programa do mock executando no terminal) simula uma rodovia com n pistas, com $\frac{n}{2}$ pistas de ida e $\frac{n}{2}$ pistas de volta. Por exemplo, uma instância do mock simularia a entrega de dados da rodovia BR-101.
- Cada instância do mock devolve seus dados dentro de uma pasta que possui o nome da rodovia (indicado ao iniciar a execução do mock). Essa pasta encontra-se dentro de outra pasta do repositório, a pasta `data`.
- O ETL fica em modo de escuta por novos arquivos dentro dessas pastas que estão dentro de `data`, sempre escolhendo o mais recente para o processamento (esse processo será melhor detalhado na seção 3).
- Durante o processo de escuta, novas instâncias do mock podem ser instanciadas (simulando novas rodovias entregando dados ao sistema de monitoramento). O ETL atualizará o número de rodovias presentes na simulação e os demais dados.
- Durante o processo de escuta, instâncias do mock que já estão rodando podem ser derrubadas (simulando uma rodovia que parou de entregar dados ao ETL), de forma que sua pasta ficará vazia. Entretanto, essa rodovia ainda será contada como uma rodovia presente na simulação (mas que não está gerando dados). Essa escolha será justificada na seção 4.1.1.

2 Mock

O processo inicial envolve a implementação de um ecossistema que gere problemas de concorrência e paralelismo — um *mock*, simulador que opera de forma mais ou menos semelhante ao cotidiano do negócio que se pretende estudar. Em particular, um *mock* de um sistema de monitoramento de rodovias, como é o caso deste trabalho, deve dar conta, principalmente, de simular uma série de instâncias de carros que interagem entre si em uma determinada rodovia a partir de determinados parâmetros, como as velocidades máxima e mínima de uma rodovia, a mudança de pistas, eventuais acidentes etc.

O *mock* implementado para os objetivos deste trabalho gera novos arquivos `.txt`, um para cada rodovia mapeada, a cada ciclo de monitoramento. Os arquivos de uma rodovia são nomeados declarando o ciclo em questão (para armazenarmos o tempo), e identificam quais carros estão naquela rodovia em cada ciclo e a posição em que se encontram. Cada carro é identificado por sua respectiva placa (gerada aleatoriamente por uma função) e a posição é dada por uma tupla que informa, no índice 0, a distância percorrida pelo carro desde que entrou na pista, e no índice 1 a pista onde se encontra.

Ao nível de implementação, cada rodovia é identificada por uma matriz de tamanho Tamanho da rodovia × Número de pistas. Rodovias possuem tanto pistas de ida quanto de volta.

Conforme as especificações indicadas, as rodovias possuem alguns atributos importantes. São eles: `name`, `lanes` (número de pistas), `size`, `speed_limit` (definição de um limite físico de velocidade para os veículos), `prob_vehicle_surge` (probabilidade de entrar um novo veículo na pista), `prob_lane_change` (probabilidade de um veículo tentar trocar de pista), `max_speed` (velocidade máxima da rodovia), `min_speed` (velocidade mínima dos carros em movimento na rodovia), `cicles_to_remove_collision` (dado que houve um acidente na pista, o número de ciclos que ocorrerão até que os carros envolvidos no acidente sejam removidos da pista), `collision_risk` (probabilidade de colisão), `max_acceleration` e `max_decceleration`.

Para cada instância do *mock* (isto é, em cada rodovia), atualizamos as posições dos carros respeitando as especificações de negócio. Segue uma explicação mais detalhada:

Veículos aceleram/desaceleram: a cada ciclo de geração de novo arquivo, a velocidade dos carros é atualizada somando o valor da aceleração à velocidade. São ajustados alguns casos particulares: (i) a velocidade do carro é mantida sempre acima do limite mínimo de velocidade na pista; e (ii) um carro não pode ultrapassar seu limite físico de velocidade (i.e., se o limite de velocidade de um carro é 200 km/h, ele não pode alcançar uma velocidade de 220 km/h).

Aqui tomamos a decisão de considerar que a aceleração deve variar no intervalo [`-max_acceleration`, `max_acceleration`], para que os veículos na rodovia possam trafegar aumentando e diminuindo sua velocidade em valores razoáveis. O

`max_decceleration` é um valor muito mais considerável, já que deve ser usado quando o carro precisa diminuir a velocidade bruscamente para evitar colisão, e na vida real os carros não realizariam reduções de velocidade dessa magnitude sem algum imprevisto.

Naturalmente, um veículo pode se encontrar abaixo da velocidade mínima, mas somente quanto está envolvido em algum acidente (colisão). A velocidade máxima pode ser ultrapassada a qualquer momento, afinal, uma das especificações do trabalho é que sejam mapeados os veículos que infrinjam a marca de velocidade máxima da rodovia. O limite, claro, é o físico do próprio carro, conforme já comentado. A decisão de implementar esse limite foi tomada em busca de tentar modelar de forma mais parecida possível com a realidade, restringindo superiormente a velocidade na medida em que, na vida real, há um máximo em que conseguem chegar.

Veículos tentam trocar de pista: de acordo com a probabilidade de um veículo trocar de pista, determinamos se o veículo trocará de faixa a partir da comparação com um número gerado aleatoriamente. Caso o veículo troque de faixa de fato, a faixa de destino é escolhida aleatoriamente entre as faixas adjacentes, a menos que o carro esteja na faixa mais à esquerda (`car.y == 0`) ou na faixa mais à direita (`car.y == road.lanes-1`). Se o carro estiver na faixa mais à esquerda, ele mudará para a faixa imediatamente à direita (`car.y += 1`); se o carro estiver na faixa mais à direita, ele mudará para a faixa imediatamente à esquerda (`car.y -= 1`). Se o carro estiver em qualquer outra faixa, ele terá uma chance equivalente de mudar para a faixa imediatamente à esquerda ou à direita (`car.y = random.choice([car.y+1,car.y-1])`).

Veículos agem conforme a possibilidade de colisão: dada a velocidade i alcançada por um veículo em um determinado ciclo, espera-se que ele tenha avançado i entradas a frente na matriz quando computarmos o ciclo a seguir. Portanto, caso haja algum outro veículo nesse range à frente, é possível que haja colisão. Assim, a variável `trigger_colision`, setada inicialmente como `False`, é atualizada para `True`, informando que há risco de acidente. Naturalmente, caso não haja carros no intervalo a frente, simplesmente atualizamos a posição do carro.

No caso geral, a partir do momento em que detectamos a possibilidade de colisão, o veículo deve avaliar se é possível evitar o acidente trocando de pista. Para isso, avaliamos primeiramente se há algum veículo na pista imediatamente à esquerda. Caso não haja, o veículo muda de pista e evita a colisão; caso haja, ele faz a mesma avaliação à direita. No caso de não ser possível se deslocar para a pista da esquerda ou da direita, a ação tomada é diminuir a velocidade mais drasticamente.

Inserção dos carros em uma nova matriz: como tecnicamente todos os carros andam ao mesmo tempo, precisamos garantir que, no cômputo da matriz que mapeia

um novo ciclo, um carro não "atravesse" outro, o que maquiaria casos de colisão. A solução para isso foi ordenar a lista de carros pela velocidade e inserir ordenadamente os carros na nova matriz, do mais lento para o mais veloz. Como avaliamos a `trigger_collision`, garantimos que o carro tentará trocar de pista ou diminuirá a velocidade para não colidir.

Colisões: para fins de teste e garantir que haja alguma colisão na execução do *mock*, há casos onde o acidente é forçado. Para isso, quando há chance de colisão (`trigger_collision == True`), geramos um número aleatório e avaliamos se ele é menor que a probabilidade de colisão. Em caso positivo, o acidente ocorre.

Quando, apesar dos esforços do motorista, ainda ocorre colisão, então setamos `car.colision = True` e encontramos o carro que corresponde àquele que foi atingido na colisão. Quando encontramos esse caso, informamos que esse outro carro colidiu.

A partir do momento que os carros colidem, suas velocidades são setadas para 0. Pelas características do código já explicadas, os carros que vierem atrás da colisão perceberão carros mais a frente e notarão possibilidade de colisão. Com isso, tentarão evitar acidente trocando de pista ou diminuindo a velocidade. O trânsito, portanto, será gerado naturalmente.

Os carros que colidiram são removidos da pista após um determinado número de ciclos.

3 Extract

3.1 Escrita e leitura dos dados gerados pelo mock

A escrita e leitura dos dados foi modelada de forma a resolver os seguintes problemas:

1. Acesso para leitura enquanto o mock ainda está escrevendo, lendo assim dados corrompidos;

2. Leitura de outros dados, e não o mais recente, ao mock gerar mais de um conjunto de dados no tempo entre uma leitura do ETL e outra.

De modo a criar uma solução para esses problemas, o grupo adaptou a abordagem do problema *produtor e consumidor*, conforme figura abaixo.

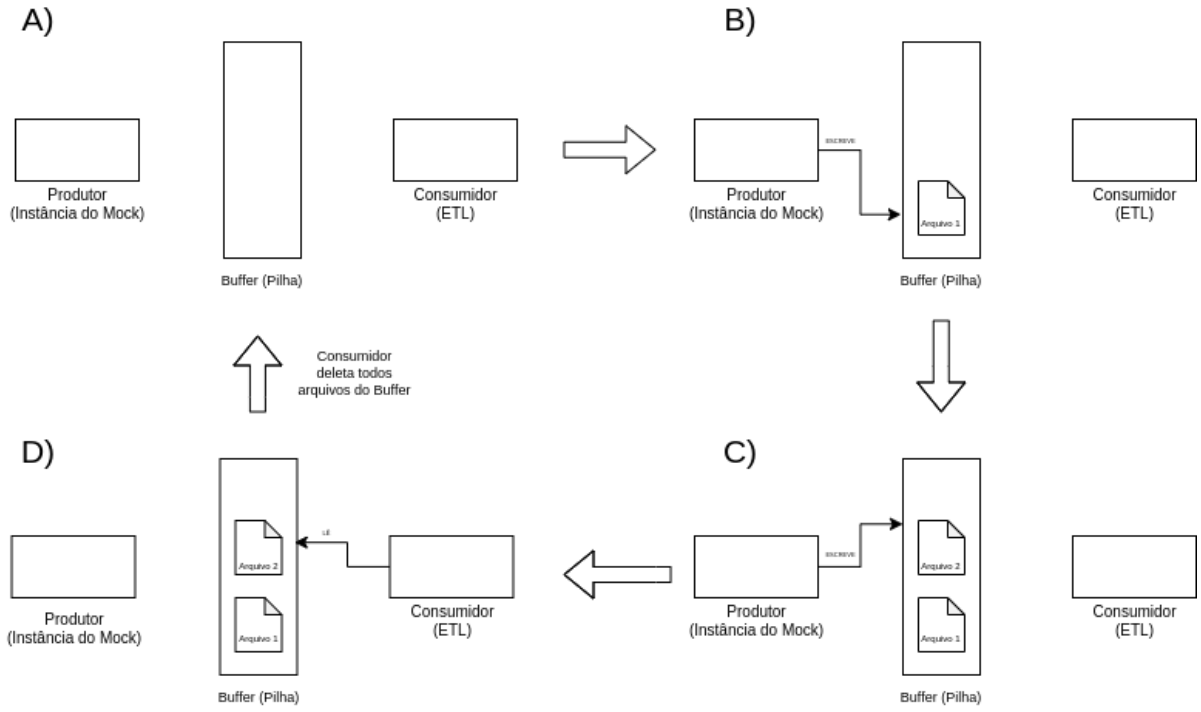

Figura 1 – Processo de extração dos dados do Mock pelo ETL (no caso do problema 3). A etapa A) é o estado inicial, onde não há dados no buffer, o produtor ainda está produzindo dados e o consumidor está processando os dados da última leitura. Nas etapas B) e C), o produtor já passou por dois ciclos de geração de dados enquanto o consumidor ainda está processando. Na etapa D), o consumidor terminou o processamento e consome o arquivo mais recente. Por fim, o consumidor apaga todos os dados do buffer e o sistema volta ao estado A).

A solução consiste em um sistema de produtor-consumidor onde o buffer é uma pilha, isto é, uma fila com política FILO (first-in-last-out), de modo que o ETL sempre consuma o último arquivo gerado (ou seja, colete os dados mais recentes). Além disso,

após consumir o arquivo do topo da pilha, o sistema apaga todos arquivos existentes na pilha (para que, na próxima leitura, ele não leia um arquivo antigo). Esse fluxograma é apresentado na Figura 1.

Perceba que, com essa lógica, evitamos os problemas acima: a geração de diversos arquivos evita o problema (1), de forma que o mock não ficará abrindo e escrevendo sobre um mesmo arquivo a cada iteração; e a utilização de uma pilha resolve o problema (2), pois sempre pega o arquivo mais recente.

É importante frisar que não temos uma estrutura de dados de pilha "de verdade", mas sim uma abstração dela através dos nomes dos arquivos, já que estamos realizando a comunicação entre aplicações diferentes. Por isso, nosso buffer "não tem limite"(exceto pela memória do computador onde o sistema estará sendo executado), e, dessa forma, nunca ficará esperando para inserir dados no buffer. Por fim, é importante ressaltar que a operação de I/O de deleção dos arquivos do buffer é rodada em uma thread separada da principal, para otimização do sistema.

3.2 Lendo de múltiplas rodovias

Vamos aplicar a solução acima em um contexto onde temos múltiplos produtores. Isso pode gerar um problema: se o produtor 1 gerar seus dados mais rapidamente que todos outros produtores, será muito provável que o ETL leia muitos arquivos do produtor 1 e poucos arquivos dos outros. Isso é um problema pois temos a prioridade de exibir os carros com risco de colisão de cada rodovia e, portanto, precisamos de uma solução para que tenhamos o risco de colisão de todos as rodovias atualizados o mais rápido possível. Explicaremos essa solução detalhadamente posteriormente. Para contornar esse problema, o ETL foi implementado de forma que lê um arquivo de cada rodovia a cada iteração. Assim, temos uma solução na qual o ETL lê o arquivo mais recente de n rodovias em n iterações, conforme a representação visual na Figura 2.

Apesar disso, é importante ressaltar que essa busca por arquivos foi implementada de forma que, caso não haja um arquivo no buffer de uma rodovia (pois ela pode ter parado de gerar dados, por exemplo), nosso ETL não ficará esperando por esse arquivo, mas sim continuará extraindo o arquivo das próximas rodovias. Isso foi implementando através de um `for` por todas as rodovias onde, caso o arquivo buscado não existe, então implementamos um `continue`.

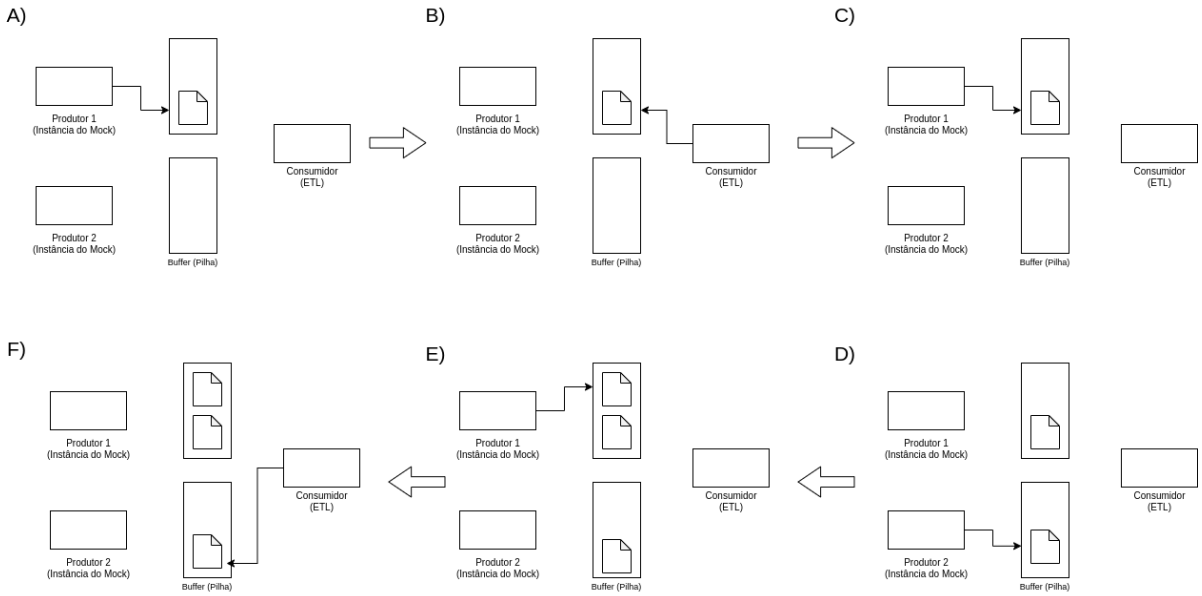

Figura 2 – Fluxograma exemplificando a ordem de leitura dos dados em caso do problema descrito acima.

4 Transform

Antes de proceder ao detalhamento da implementação, é importante informar que utilizamos dois `typedef` para facilitar a leitura do código. O primeiro deles é a `Lane`, um vetor de tuplas que representa uma **pista** de uma rodovia, onde o primeiro elemento da tupla é a placa do carro e o outro é um inteiro, que varia como sendo a velocidade, a aceleração ou o risco de colisão de cada carro, dependendo do caso. O segundo `typedef`, `Road`, é um vetor de `Lane`, representando um conjunto de pistas, ou seja, uma rodovia.

4.1 Modelagem da *main* do programa

Nessa seção, vamos apresentar a modelagem geral da *main* do programa. Para fins de visualização e explicação, seu fluxograma será dividido em 2 partes, que serão explicadas abaixo. Após, explicaremos com mais detalhes as funções utilizadas e outros aspectos do programa.

Note que os blocos em vermelho — no caso, a apresentação dos dados no dashboard —, são protegidos por um mutex para garantir que todas as informações serão apresentadas.

4.1.1 Inicialização e atualização dos repositórios de dados

Nossa análise inicia-se com a criação das estruturas de dados que serão os nossos "repositório de dados", necessários para as análises subsequentes. Essas estruturas são `vectors` de `Road`. Cada posição do `vector` se refere aos dados de uma rodovia, isto

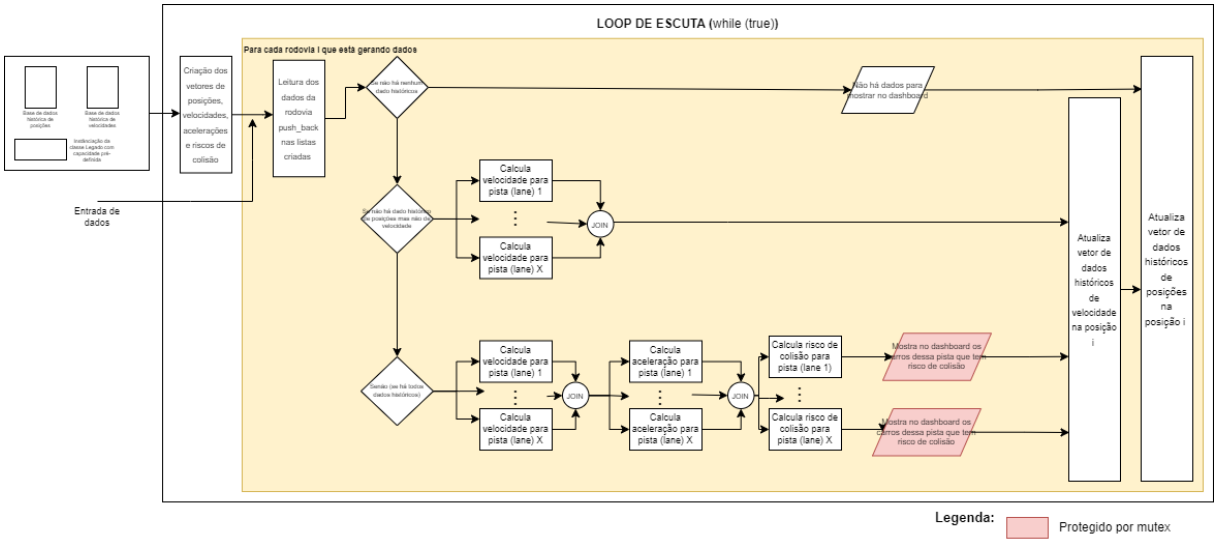

Figura 3 – Funcionamento do início da *main*. Para melhor visualização, a imagem pode ser visitada nesse link.

é, uma instância do *mock*. Essa inicialização é realizada fora da escuta (isto é, fora do `while True`). Entretanto, após o início da escuta, verificamos a cada iteração se o número de rodovias aumentou (isto é, se uma instância do mock foi levantada enquanto o ETL executava). Em caso positivo, aumentamos o tamanho desse vetor. Essa escolha também justifica uma das considerações que já relatamos na seção 1 desse relatório: a de que uma rodovia que para de gerar dados não é considerada como uma rodovia que saiu da simulação — ora, se ainda possuímos seus dados históricos no nosso repositório de dados, então essa rodovia ainda é uma instância no nosso programa.

4.1.2 Cálculo da quantidade de rodovias

Cada instância do *mock* (isto é, cada rodovia) gera os dados em uma pasta dentro da pasta *data* do projeto. Assim, a lógica para cálculo da quantidade de rodovias foi contar o número de *sub-pastas* dentro da pasta *data* do projeto. Essa contagem foi realizada através do pacote *dirent.h*. Subtraiu-se duas unidades do valor retornado, referentes às pastas `.` (diretório atual) e `..` (diretório pai).

4.1.3 Prioridade da thread

Conforme os requisitos do projeto, a análise que verifica o risco de colisão deve ter prioridade em relação às outras. Para solucionar esse problema, a ideia foi calcular o risco de colisão o mais cedo possível no programa e exibi-lo assim que calculado (isto é, exibir os dados mesmo que o programa tenha outras tarefas para realizar). Para que isso seja possível, foi necessário calcular a velocidade dos carros, as acelerações (duas

informações necessárias para cálculo do risco de colisão) e, após o cálculo, sua exibição direta no dashboard (no nosso contexto, o console).

Seguimos, então, para os cálculos. A entrada que o sistema de monitoramento recebe é um arquivo onde cada linha representa um carro — mais especificamente, sua placa e posição x (distância percorrida na rodovia) e y (pista). Portanto, utilizamos essas informações para calcular as velocidades, acelerações e riscos de colisão. A motivação é a seguinte: dados os veículos em uma determinada pista da rodovia, os adicionamos a uma lista ordenada pela posição x na pista. Com isso, passamos a realizar a avaliação dos pares de carros — cada veículo em relação àquele imediatamente à frente. Consideramos, conforme modelado no *mock*, que dado um ciclo $i + 1$, a posição$_{i+1}$ = posição$_i$ + velocidade$_i$ + aceleração$_i$. Explicamos o processo mais detalhadamente a seguir.

4.1.4 Cálculo da velocidade

A função `calc_speed_thread( )` é utilizada em cada *thread* de cálculo de velocidade, isto é, é chamada dentro da função `calc_speed( )` e passada para cada *thread* criada. Essa função recebe:

`positions:` vetor de vetores, um para cada pista, de tuplas com identificação de placa e posição de cada carro. Esse vetor maior, portanto, representa as posições na rodovia inteira;

`*old_position:` ponteiro para `old_position`, de mesmo tipo, que são dados históricos das posições do último ciclo;

`*calculated_speed:` ponteiro para `calculated_speed`, que é onde inserimos as tuplas de (placa, velocidade) que calculamos;

`i e j:` inteiros que representam respectivamente a pista e o carro em questão;

`ref(mtx):` mutex utilizado na região crítica para escrever em `calculated_speed`, já que teremos múltiplas *threads* realizando esta mesma ação ao mesmo simultaneamente.

Essa função é chamada em cada *thread* de cálculo de velocidade. Ela procura a posição anterior do mesmo carro fazendo comparações pela placa em `old_position`, e calcula a velocidade atual dada pela `posição atual - posição anterior`. Em seguida, cria uma tupla (`placa, velocidade`) e a insere no vetor de velocidades calculadas (`calculated_speed`). Trata-se de uma região crítica protegida por um por mutex para evitar que múltiplas *threads* escrevam no vetor simultaneamente.

A função `calc_speed( )` é a responsável por distribuir os cálculos de velocidade em *threads*. Ela recebe como argumentos o mesmo vetor de vetores de tuplas, `positions`, e um ponteiro para `old_positions`.

A função cria um vetor (rodovia inteira) onde irá armazenar as velocidades calculadas e adiciona vetores (pistas) a ele. Para cada pista, cria-se uma *thread* na qual se chama a função `calc_speed_thread( )` para calcular a velocidade de cada carro naquela pista. Essa implementação de pseudoparalelismo em threads foi modelada de forma a entregar o resultado o mais rápido possível. A contagem do número de *threads* indica o número total de pistas da rodovia que está sendo analisada.

4.1.5 Cálculo da aceleração

O processo para calcular a aceleração é semelhante. Entretanto, o cálculo da aceleração para um carro depende do cálculo da velocidade desse mesmo carro. Por isso, para uma mesma pista, esse cálculo só pode ocorrer após termos calculado as velocidades desses carros. Por isso, não é possível implementar essa função para ocorrer em paralelo com o cálculo da velocidade.

Nesse caso, a função `calc_accel` recebe um vetor de tuplas com as velocidades atuais, e outro com as velocidades do ciclo anterior. Novamente, a tupla armazena a placa do carro no índice 0.

A função `calc_accel( )` cria *threads* para cada pista e chama a função `calc_accel_thread` em cada uma delas, que recebe:

`speed:` vetor de vetores (um para cada pista) de tuplas com argumentos (`placa, velocidade`) para cada carro (isto é, um vetor de `Lanes`), representando as velocidades de todos os carros na rodovia;

`*old_speeds:` ponteiro para `old_speeds`, de mesmo tipo, que são dados históricos das velocidades do último ciclo;

`*calculated_accel:` ponteiro para `calculated_accel`, que é onde inserimos as tuplas de (placa, aceleração) que calculamos;

`i e j:` inteiros que representam, respectivamente, a pista e o carro em questão;

`ref(mtx):` mutex utilizado na região crítica de escrever em `calculated_accel`, dado que teremos múltiplas *threads* fazendo o mesmo.

Primeiramente, a função `calc_accel( )` inicializa o vetor `calculated_accel` com o tamanho do vetor `speed` e, em seguida, itera por cada velocidade atual e por cada carro dentro dessa velocidade.

Para cada pista, uma *thread* é criada e a função `calc_accel_thread( )` é utilizada. Para cada carro, ela procura a velocidade do mesmo carro no ciclo anterior em `old_speeds` e calcula a aceleração a partir da diferença entre as velocidades. Em seguida, cria uma tupla (`placa, aceleração`) e a insere no vetor de acelerações calculadas

(`calculated_accel`), protegendo por mutex para garantir que apenas uma *thread* adicione uma tupla ao vetor de acelerações por vez.

A função retorna um vetor chamado `calculated_accel`, do tipo `Road`, contendo as acelerações calculadas e placa para cada carro.

Note que a chamada da função `join( )` deve ocorrer antes de chamar a função que determina o risco de colisão. Isso é necessário porque o risco de colisão é definido a partir da comparação de um carro com o outro imediatamente à frente, ou seja, é necessário que todos os carros estejam com as suas posições devidamente atualizadas.

Repare que em ambas as funções (de cálculo de velocidades e de acelerações) optou-se por criar as threads por pistas que estão presente em uma rodovia, ao invés da criação por carros. A partir das aulas, sabemos que após um certo número X de threads executando em pseudoparalelismo, atingimos o limite do *hardware*, além de o custo de troca de contexto começar a "pesar" no tempo. Com isso, a otimização de tempo se torna limitada - este acaba se mantendo praticamente constante mesmo com o aumento do número de threads. Dado isso, escolhemos criar as threads por pistas de uma rodovia, já que a diferença de tempo entre a criação por pistas e por carros seria muito pequena.

Perceba que, aqui, poderíamos ter optado por outra modelagem: ao invés de rodar o cálculo das velocidades em paralelo, dar *join()*, rodar o cálculo das acelerações em paralelo, dar *join()* poderíamos dividir as threads de forma que uma mesma thread executaria esses dois cálculos e só posteriormente dar *join()*. Essa maneira seria levemente mais rápida, visto que eliminaríamos um *join()* do código. Entretanto, isso acarretaria complicações desnecessárias no código por pouco ganho em tempo. Portanto, decidimos por implementar da maneira descrita nesse relatório.

4.1.6 Cálculo do risco de colisão

O risco de colisão é uma variável binária que assume valor 0 quando não há risco, e 1 quando há risco de acidente. Para determinar se os veículos potencialmente se envolverão em um acidente, implementamos uma função `calc_collision_risk( )` que recebe ponteiros para os vetores criados com o cálculo de velocidade e aceleração, assim como um ponteiro para o vetor de posições (todos do tipo `Lane` — vetores de tuplas com a placa na posição 0 e o valor em questão na posição 1).

Inicialmente, ordenamos o vetor de posições em ordem crescente. Em seguida, iteramos pelo vetor em busca da sua velocidade e aceleração. Com esses valores, calculamos a posição que esperamos que o carro assuma no próximo ciclo. Isto é, recebemos posição_i, velocidade_i e aceleração_i, e calculamos:

$$\text{posição}_{i+1} = \text{posição}_i + \text{velocidade}_i + \text{aceleração}_i$$

Substituímos os valores no vetor de posições pelas posição_{i+1}, mas mantendo a ordenação. Assim, podemos iterar pelo vetor checando, para cada carro, se há risco de

colisão com o próximo veículo. Com isso, adicionamos as novas tuplas (placa, risco de colisão) a um vetor `collision_risk`.

Ao iterar pelo vetor de posições, checamos, para cada carro, se a sua nova posição ultrapassa a nova posição do carro à frente. Em caso positivo, entendemos que há risco de colisão. Adicionamos ambos os carros envolvidos em `collision_risk` com a placa dos carros e valor 1. Se não identificarmos risco de colisão (isto é, os carros estiverem ordenados de forma crescente), adicionamos com a placa do carro e valor 0. Note que, como adicionamos ambos quando detectamos colisão, não podemos adicionar o carro da frente novamente quando checamos se há risco de colisão dele com o próximo: usamos uma variável booleana auxiliar `previous_collision` para evitar que isso aconteça, checando sempre se houve colisão agora e anteriormente.

Na `main( )`, passamos a função `calc_collision_risk( )` para `threads` em um vetor de `threads`, calculando paralelamente o risco de colisão dos carros em diferentes `Lanes`.

Por fim, apresentamos esses resultados já no dashboard (console) e, em seguida, atualizamos os vetores dos dados históricos. Depois disso, a *main* segue com sua *thread* principal para a segunda parte, apresentada abaixo.

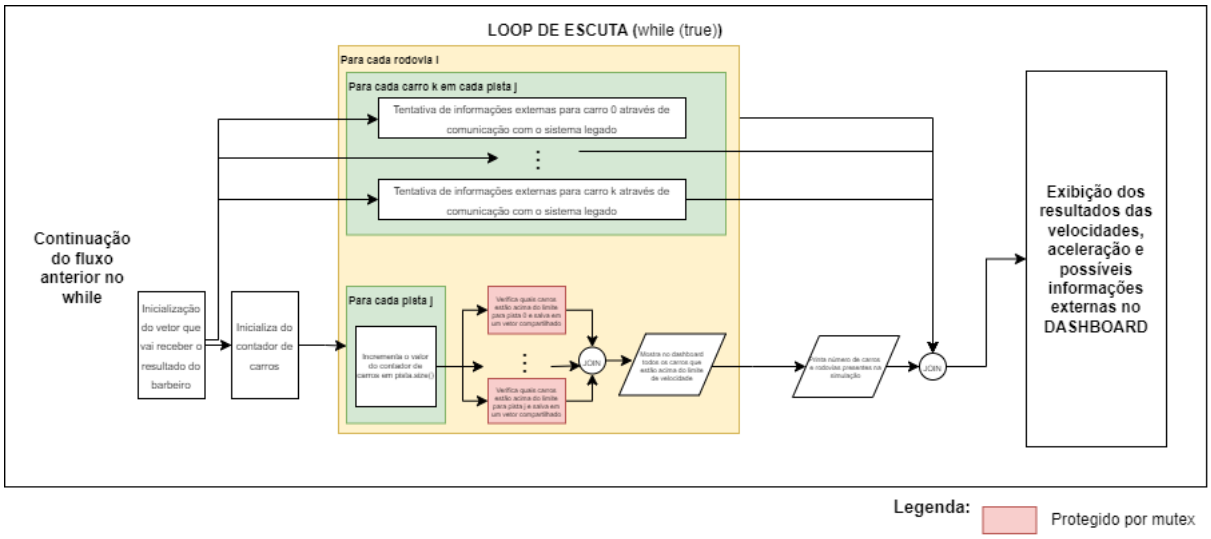

Figura 4 – Funcionamento da segunda parte da *main*. Para melhor visualização, a imagem pode ser visitada nesse link.

As análises da segunda parte começam com a inicialização do vetor em que vamos receber o resultado da consulta de informações no sistema externo (esse item será explorado e destrinchado mais a frente, quando tratarmos da solução desenvolvida para o sistema Legado). Em seguida, já podemos partir para as tentativas de buscar informações de proprietário, modelo e ano através desse sistema Legado, realizando essas consultas para cada carro, em cada pista, em cada rodovia. Entretanto, paralelamente, inicializamos

a análise do número de veículos e dos carros acima da velocidade.

4.1.7 Cálculo de número de veículos

Para realizar o cálculo da quantidade de veículos presentes na simulação, utilizamos um vetor de `Road`, que armazena todas as rodovias da simulação (`positions_list`). Realizamos um `for` que passa por cada rodovia, e cada pista e ao percorrer o vetor, o contador `n_carros_simulacao` é incrementado pelo quantidade de carros em cada pista, contando assim todos os carros na rodovia. Optamos por não usar threads para esse cálculo pois simultaneamente temos múltiplas threads responsáveis pelas consultas na api legado em paralelo, e como a consulta a api legado é nosso limite inferior de tempo (última operação a ser resolvida), para nossa modelagem não havia ganho em paralelizar.

4.1.8 Cálculo de veículos acima da velocidade

A função `speed_limits( )` é responsável por checar o limite de velocidade de cada rodovia e armazená-los em um vetor de inteiros. Ela recebe um vetor de strings com o nome das pastas e retorna um vetor de inteiros com os limites de velocidade de cada rodovia, extraídos a partir do nome de cada pasta.

Para cada rodovia, utilizamos a função `cars_above_limit( )`, que recebe a matriz de velocidades de cada carro (`matrix_speeds`) e um inteiro (`limit`) representando o limite de velocidade dessa rodovia, extraído do vetor `speed_limits`, retornado pela função `speed_limits( )`.

Então, para cada pista na matriz de velocidades, é criada uma *thread* que executa a função `is_above_limit( )` para calcular se cada carro na pista em questão está acima do limite de velocidade. Essa função recebe como argumentos:

`lane_to_calc` ponteiro para uma pista - vetor de tuplas do tipo `Lane`;

`limit` inteiro que representa o limite de velocidade da pista em questão;

`answer` ponteiro para um vetor de vetores de tuplas que armazena os resultados booleanos para cada carro;

`ref(mtx)` mutex utilizado para proteger o acesso ao vetor `answer`.

A função percorre a pista, checa se cada carro está acima do limite de velocidade e armazena o resultado em um vetor de tuplas de (`string, booleano`). Esse vetor é então adicionado à matriz de resultados `answer`, que será retornada pela função.

Por fim, os resultados dessa análise são enviados ao dashboard (console). Em seguida, os resultados do número de carros e número de rodovias também são enviados.

4.1.9 Comunicação com sistema legado

Para retornar informações de nome do proprietário, ano e modelo de um carro (através de sua placa), o ETL consulta um sistema legado (no nosso projeto implementado através da classe `Legado`).

Para a maior eficiência possível, ao receber uma lista `(placa, posição)`, criamos uma *thread* para cada placa para a realização da consulta à base de dados externa (no nosso projeto, representada no arquivo `legado.csv`). A consulta é feita através da função `query_vehicle` da classe `Legado`. Entretanto, como todas *threads* chamam essa função da classe em pseudoparalelismo, essa classe possui uma capacidade de requisições para processar, além de processar apenas uma por vez. Assim, temos um problema de comunicação com o sistema legado a resolver, para que não tenhamos nenhuma espécie de cruzamento de dados (como uma *thread* receber informações de outra placa) ou até mesmo uma *thread* ficar esperando para sempre para ser processada.

Para o problema de comunicação com o sistema legado, a solução foi a implementação do problema clássico de barbeiro adormecido, mas com algumas adaptações. Para a nossa aplicação, temos os seguintes papéis:

- O barbeiro adormecido é a função da classe `Legado` que faz a consulta aos dados referentes à uma placa;
- Um cliente é uma requisição do ETL para o sistema legado;
- A quantidade de "cadeiras disponíveis"é a capacidade da fila indicada no construtor da classe do sistema legado.

Repare que temos apenas um "barbeiro", já que possuímos apenas uma instância da classe `Legado` no código. Essa escolha de abordagem foi feita devido aos requisitos do trabalho, principalmente ao fato de que se a fila estiver cheia, então o processo cliente (nesse caso, a requisição do ETL para o sistema legado) deverá seguir a execução sem obter a resposta.

A análise começa, conforme já explicado acima, com a distribuição das N placas (existentes no arquivo) em N *threads*. Após, todas essas *threads* tentam acessar a classe Legado.

Essa tentativa começa com a avaliação de se ainda há espaço disponível na fila de processamento. Caso haja, a análise entra para a fila. Caso não, a análise segue adiante e essa *thread* não obterá as informações adicionais. Repare que nesse momento podemos ter o problema de duas threads fazendo essa avaliação ao mesmo tempo (por exemplo: temos apenas uma vaga na fila, mas duas threads fazem a avaliação da condicional do `if` ao mesmo tempo. As duas vão retornar `true` e incrementar o contador de *requests* na fila ao mesmo tempo, enquanto na verdade só temos capacidade para uma). Para contornar

esse problema, foi um implementado um *mutex* `queue_mutex` na hora de avaliação da capacidade da fila.

Caso haja vaga, a análise entra para ser processada. Nesse momento, ela chama a função que realiza a consulta e guarda nos campos da classe. Depois, esses campos da classe são acessados através de funções `get` e o valor retornado é salvo em um `vector` referente a essa placa. Nessa etapa, também há a implementação de um *mutex* `isExecutingMutex` a fim de garantir que não haja mais de uma *thread* solicitando a avaliação ao mesmo tempo.

Finalmente, o contador da fila é decrementado após a avaliação, realizando o *lock* e *unlock* no `queueMutex` novamente, antes e depois da operação.

Repare que diferentemente do código padrão estudado em aula, temos um *mutex* a menos no código. Isso acontece pois nosso "barbeiro"não está acordado sempre, isto é, não possui um `while True`. Ele só é "acordado"quando uma *thread* chama sua função de análise.

Logo no início das criação das threads, é criado um vetor de tamanho n, sendo n o número de placas (e, consequentemente, o número de threads) dessa iteração. Chamaremos esse vetor de "vetor compartilhado".

Além disso, cada *thread* possui um vetor referente a placa que está sendo analisada (chamaremos esse vetor de "vetor local"). Esse vetor tem tamanho 4, onde na primeira posição temos a placa que está sendo analisada e nas três restantes temos

- Em caso de acesso à base de dados externa: as outras posições serão modelo, ano e nome do proprietário do carro;
- Em caso de não conseguir acesso à base de dados, então essas posições receberão um espaço vazio.

Após a avaliação (ou não) de cada placa, as threads alocam seus "vetores locais"na posição i do vetor criado antes da criação das threads, ou seja, a thread i aloca o vetor local referente à sua placa na posição i do vetor compartilhado. Essa abordagem foi utilizada para evitar que haja problemas de concorrência na hora do salvamento dessas informações, devido a utilização de uma "memória compartilhada"pelas threads. Com essa abordagem, cada thread acessa apenas um espaço de memória do vetor e, assim, evitamos possíveis problemas.

5 Load

Ao executar o projeto, as seguintes análises são apresentadas no dashboard exibido no terminal:

- Número de carros presentes na simulação;
- Número de rodovias presentes na simulação;
- Veículos acima do limite de velocidade da rodovia;
- Informações adicionais de cada veículo, obtidas a partir do Legado;
- Velocidade e aceleração de cada veículo;
- Veículos que apresentam risco de colisão;
- Tempo de análise.

Como comentado anteriormente, a análise de risco de colisão possui prioridade sobre as outras análises, por isso é exibida primeiro a cada iteração do ETL. As outras análises são feitas conforme explicado nos tópicos anteriores, calculadas através da `main`.

A escolha de apresentar o dashboard no terminal foi pensada em exibir as informações o mais rápido possível, em vez de utilizar alguma ferramenta externa de BI que aumentaria consideravelmente esse tempo.

Por fim, apresentamos abaixo um pseudo-código da parte final da *main*, onde exibimos os resultados no dashboard (console).

```
Para cada rodovia:
    Para cada pista:
        Para cada carro:
            Mostra placa e posicao
            Procura carro com essa placa no vetor de velocidades:
                Se acha:
                    Mostra velocidade
                   Sai do loop;
                Senao:
                    Mostra que nao possui;
            Procura carro com essa placa no vetor de aceleracao:
                Se acha:
                    Mostra acelara  o
                    Sai do loop;
                Senao
                   Mostra que n  o possui;
            Procura carro com essa placa no vetor de risco de colisao:
                    Se acha:
                        Mostra risco = 1;
```

```
20                          Sai do loop;
21                      Senao
22                          Mostra risco = 0;
23              Procura carro com essa placa no vetor de informacoes
    adicionais:
24                  Mostra as informacoes adicionais;
```

Das linhas 1-3, percorremos até cada carro. Depois, para cada carro mostramos sua placa e sua posição atual (linha 4). Após, percorremos o vetor de velocidades/aceleração/risco de colisão para achar as informações desse carro (linha 5/11/17). Aqui, é importante ressaltar que devemos avaliar se esse carro existe (linha 6/13/18) ou não (linha 9/15/21), afinal, se for um carro que entrou agora na rodovia (sua primeira/segunda iteração), não teremos dados históricos para calcular velocidade/aceleração. Se existir, exibimos a informação (linha 7/13/19) e sai do loop (linha 8/14/20) para evitar que o código continue correndo. Se não, apenas mostra que não possui (linha 10/16/22). Por fim, exibimos as informações adicionais (linha 23 e 24), que podem existir (caso a thread tenha conseguido acessar a classe Legado) ou serem vazias (caso não tenha).

5.1 Tempo de análise

Para calcular o tempo de análise, realizamos a diferença de tempo entre o momento em que os dados são gerados no `mock` e o momento em que a análise é concluída. O momento em que os dados são emitidos é extraído do nome de cada arquivo - o método `time()`, da biblioteca `time`, foi utilizada no *mock* na geração dos dados. No final, após todos os resultados serem exibidos no dashboard, acessamos o tempo atual, ou seja, o momento de conclusão da análise, utilizando a biblioteca `chrono`. A partir disso, calculamos a diferença e exibimos, em milisegundos, o tempo de cada análise.

5.2 Exemplo de output para um carro

```
FC731 --> Possui Risco de Colis o
Placa: FC731
Velocidade acima do limite.

Placa: FC731
Posi   o: 8917
Velocidade: 130
Acelera  o: 10
Risco de colis o: 1
Modelo: Taurus
Ano: 2004
Propriet rio: Betty Gray
```