Fundação Getulio Vargas - Escola de Matemática Aplicada
Curso: Ciência de Dados e Inteligência Artificial
Disciplina: Computação Escalável
Professor: Thiago de Pinheiro Araujo
Alunos: Gianlucca Devigili e Maisa Fraiz

Projeto de AS

1 Introdução

Este documento é um relatório do projeto de paralelismo e concorrência em uma simulação de trânsito, realizado como Avaliação Substitutiva para a matéria de Computação Escalável.

Com o objetivo de representar ruas, cruzamentos, carros e outros aspectos do tráfego, e explorar os assuntos de computação abordados na disciplina, a seguir apresentaremos a modelagem do nosso trabalho, algumas decisões importantes do nosso projeto e como resolvemos e contornamos alguns de nossos desafios.

2 Modelagem geral

2.1 Modelagem da Simulação

Para a simulação do tráfego, utilizamo-nos do paradigma de Programação Orientada à Objetos, em `Python`. A decisão para a escolha dessa linguagem, em detrimento de `C++`, foi a maior fluência dos membros da equipe.

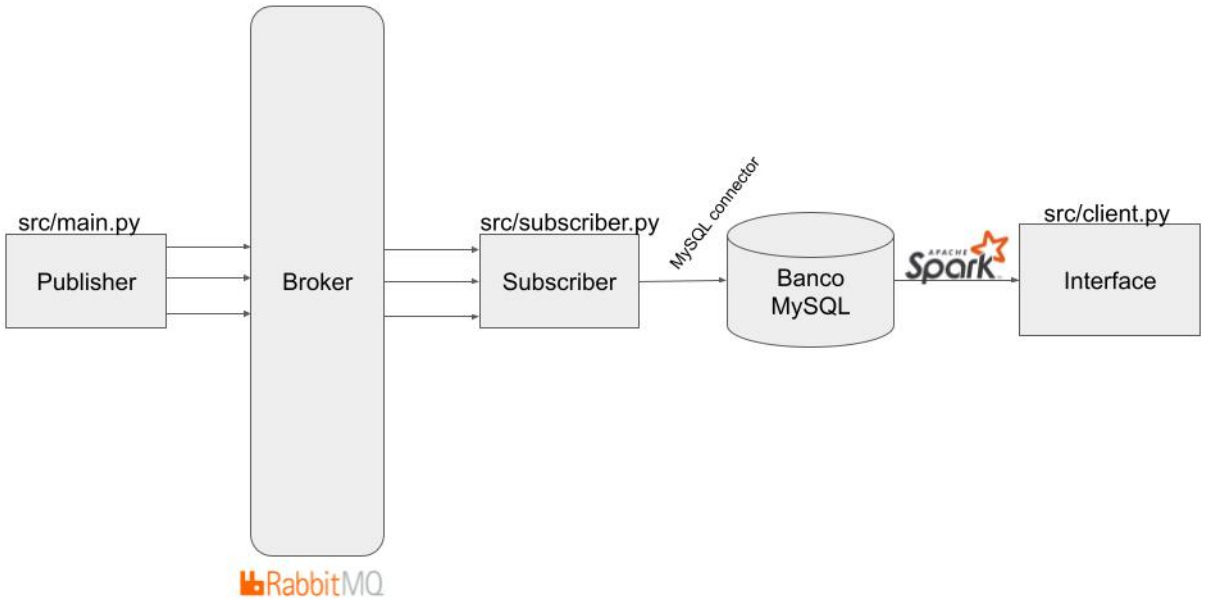
Temos então quatro classes: *Simulation*, representando a simulação em si, *Street*, representando as ruas, *Intersection*, representando as interseções e *Car*, representando os carros.

As ruas são geradas decidindo aleatoriamente se elas serão verticais ou horizontais, e então decidindo (também de modo aleatório) um ponto de início e de fim, criando assim um grafo de nós e arestas. Todas as ruas tem um mesmo limite máximo de quantos carros podem andar em uma direção.

Cada interseção tem uma posição no mapa, uma rua vertical e uma rua horizontal associadas (que juntas geram a interseção) e um dicionário de semáforos, indicando se o sinal está aberto ou fechado para cada uma das direções possíveis (norte, sul, leste e oeste). O estado desses semáforos é atualizado a cada ciclo, com uma probabilidade maior de estar fechado do que de estar aberto.

Cada carro tem uma velocidade entre 1 e 5, gerada aleatoriamente. Ao passar por uma interseção, ele decide, com igual probabilidade, para qual dos três lados (excluí-se a direção na qual ele veio) ele irá virar. Se o semáforo para a direção escolhida estiver fechado, o carro fica parado quantos ciclos forem necessários até que o sinal se abra novamente. A velocidade do veículo também é aleatorizada novamente a cada interseção. O carro é deletado da simulação se chega ao fim da rua e não vira para uma direção em uma rua cujo limite máximo de carros naquela direção já tenha sido atingido, seguindo então em frente (comentaremos essa abordagem na seção de dificuldades). Para cada espaço sobrando de cada rua (ou seja, a diferença entre a quantidade máxima de carros permitidos e quantidade de carros existentes), existe uma chance de 1% de um carro novo ser gerado. O carro novo é gerado somente no ponto inicial ou final da rua.

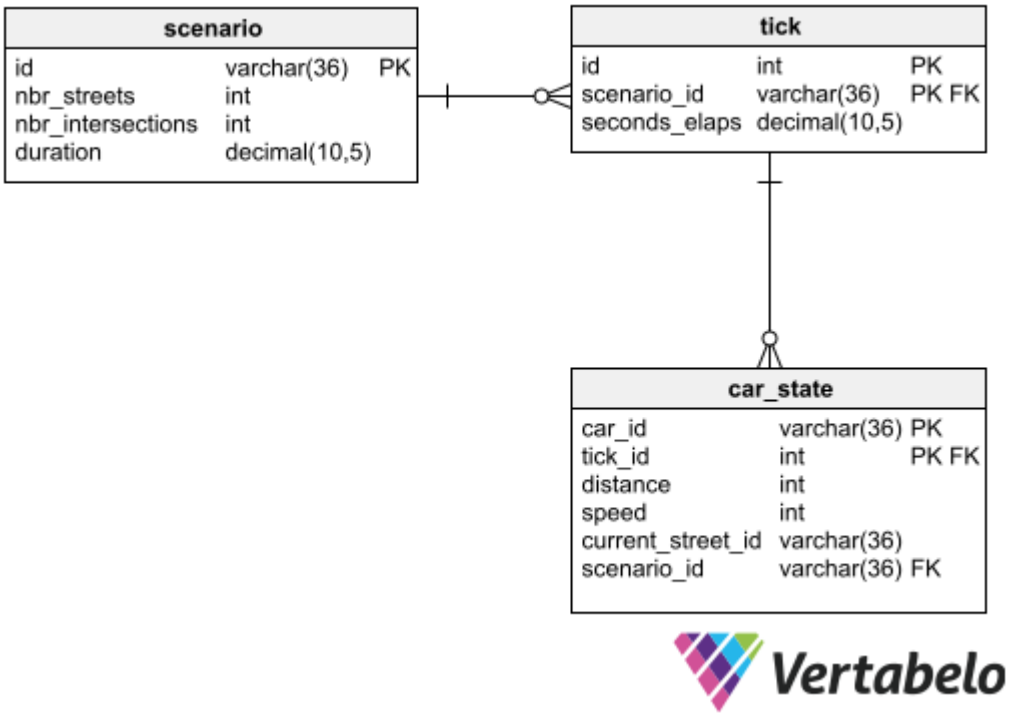
2.2 Modelagem da Arquitetura



Para a modelagem da arquitetura temos, no arquivo `main.py`, uma instância de simulação rodando por vez, em thread diferente da principal, e enviando mensagens para um Broker do `RabbitMQ`. Esse Broker possui uma fila de mensagens, que o subscriber consome para registrar no banco `MySQL` por meio do MySQL connector do Python. Por meio do `Spark`, os dados são extraídos por linha de comando para a interface, onde é possível visualizar as análises de dados.

2.3 Modelagem da Base de Dados

Segue abaixo a modelagem da nossa base de dados.



Há alguns comentários a serem feitos: os atributos das tabelas são aqueles que serão utilizados pra as análises, como número de cenários em execução, número de veículos, de ruas e cruzamentos.

Também utilizamos chaves primárias concatenadas: o cenário é definido tanto pelo seu id, quanto pelo id do tick (a iteração, o ciclo). Assim, temos os dados referentes ao "cenário tal" na "iteração tal". O mesmo foi feito para os carros, e assim podemos coletar as informações do veículo a cada iteração.

Pela forma como os atributos foram modelados, não há necessidade da tabela *car_state* estar diretamente ligada à *scenario*, pois a chave primária do cenário está na tabela *tick*, e a chave primária de *tick* está na tabela dos carros, unindo indiretamente as duas tabelas.

3 Decisões de Projeto

- Utilizar a linguagem Python, pela fluência dos membros da equipe e pela maior eficiência no desenvolvimento e na criação de uma interface gráfica.
- Utilizar, para a base de dados, o MySQL, também pela maior familiaridade dos membros quando comparado com outros sistemas.

- Usar o RabbitMQ como servidor de mensageria, para rodar o publisher e o subscriber.
- Achamos que o acesso exclusivo ao par de arestas por parte dos carros, ou seja, o carro aguardando o espaço na rua ser liberado para entrar, era muito mais condizente com o problema do barbeiro dorminhoco do que com o problema dos filósofos jantando, como dito no enunciando do exercício. Afinal, os carros criam uma fila de espera até liberar "lugar" para eles na rua, semelhantemente aos clientes da barbearia. O número máximo de cadeiras seria o número máximo de carros na rua, porém como carros novos não entram e nem são gerados em ruas já cheias, não precisamos nos preocupar com o caso de termos "mais clientes chegando sem cadeiras de espera".

4 Problemas técnicos da dupla

Houveram alguns problemas técnicos relacionados às maquinas dos membros da equipe, que afetaram parcialmente no desenvolvimento do projeto.

Para a Maisa, `Windows 11`, a instalação do programa Docker Desktop falhou, de forma que não foi possível rodar a biblioteca `RabbitMQ` nem a parte do código referente à troca de mensagens. Após tentar diversas soluções de vários fóruns de ajuda, incluindo alternativas o funcionamento do Docker pelo `Ubuntu` no WSL2 sem o Docker Desktop, e nada se provar eficaz, foi decidido a abertura de um branch no repositório, onde o sistema de mensagens estaria comentado de forma temporária, para poder realizar alterações em outras partes do código sem a aparição de erros. Além disso, o pacote de threads, quando aplicado ao código, também levantou erro somente nesse SO (veja: https://stackoverflow.com/questions/30081961/multiprocessing-works-in-ubuntu-doesnt-in-windows).

Para o Gianlucca, `Ubuntu - Linux,` uma atualização corrompeu o sistema nos últimos dias de entrega. A única solução foi a reinstalação de todo OS, atrapalhando o fluxo de progresso.

5 O que não conseguimos implementar

Até a data de entrega do trabalho, não conseguimos solucionar o problema do barbeiro. Criamos um dicionário na classe *Street*, que ditaria, para cada direção da rua, uma lista representando a fila de carros aguardando para entrar na mesma. Essa lista é inicializada vazia. Na classe *Cars*, fizemos então um if onde, na função que altera a direção do carro, caso o carro decida entrar em uma rua que já está cheia, a sua velocidade é definida para zero e o próprio carro (self) entra no final da fila, por um .append. Até essa parte da implementação, tudo funcionou. Então, de volta na classe *Street*, para cada update, ele checa se a fila de espera para entrar na rua está não vazia e se a rua está não cheia. Nesse caso, o primeiro carro da lista, que é o primeiro da fila, atualizaria sua posição e direção e seguiria o seu caminho para a rua que agora há espaço, e então faríamos um .pop(0) para retirá-lo da fila.

Porém, após rodar o código, os carros permaneceram parados na interseção, nunca virando para a rua, mesmo quando essa liberava lugar, e não conseguimos descobrir o erro

a tempo da entrega. Para não atrapalhar o fluxo da simulação, fizemos então com que os carros seguissem reto quando a direção selecionada para virar estivesse cheia. Todo o código da tentativa pode ser encontrado nos arquivos, porém o deixamos comentado.