The Bridge problem

Multithreading synchronised programming visualisation in Java and React

Marcin Bator 173592
Wiktor Mazur 173669

Operating Systems

Rzeszów University of Technology 2023

Table of contents

1. Project assumptions

On a two-way north-south road, there is a narrow bridge that allows cars to pass in only one direction at a time. Synchronize the passage of cars from the south and north to avoid collisions and ensure that a car from each direction can eventually cross the bridge.

2. Implementation agenda

Solution comes as web application that can be opened in a web browser. It uses both server- and user-side code, and multithreading is implemented on the back-end server.

User can spawn a new car on the road and select whether it's coming from north or south. Time needed for passing the bridge is different for every car. Car then places itself at the end of selected queue and waits for the bridge to be empty, because bridge can only have one car at time. Once the bridge is empty, car passes the bridge and drives further down the road. User can also select how many cars will be let into the bridge from one side until the green light appears on the other one.

Application is fully *dockerized*, which means that it can be fast runned anywhere using docker-compose.yml file in main directory of the project. Both front-end and back-end layers are stored in separate images.

3. Technologies

3.1. Front-end (client side):

— React 18
— Next.js 14.0.4
— TypeScript 5
— Tailwind CSS 3.3.0

3.2. Back-end (server side):

— Java 17.09
— Spring Boot 3.2
— Apache Maven

4. Front-end implementation details

4.1. Overview

Application comes with pleasant and straightforward UI. User can add cars from both sides and increase or decrease amount of passing cars before switching side. Moreover, there is visualisation of cars and their state, red and green lights from every side and amount of cars in each queue. After hovering the car, its time of passing the bridge is shown. Time of passing the bridge is being generated as a random integer from range 1000 to 9000 milliseconds as soon as the car is being added to queue by user.

4.2. API communication

Front-end layer is being exposed on port 3000 and communicates with back-end API located on port 8080. As workflow is quite straightforward, the standard JS *fetch* and *await* is being used. To get the current state of every car, application calls an API every 10 milliseconds.

5. Back-end implementation details

5.1. Overview

The web layer of backend application is provided by modern Java web framework: Spring Boot in latest stable version 3.2. REST controller is pretty simple and consists of 6 endpoints, which deliver following features: adding a car, getting all cars in list, getting current direction with green light, getting current amount of cars being let from each side, changing mentioned amount and the last one, which is being called every 100 milliseconds, looks for the cars that have been crossed the bridge for at least 5 seconds and removes them from the list.

The service layer provides whole workflow for processing cars in the queues. Car added by user is being put into desired queue, 'SOUTH' or 'NORTH', which means the direction the car is coming from. Every 100 milliseconds program checks the queues and if necessary, starts a new thread. Simulation of time needed for crossing the bridge is done by sleeping the thread for desired amount of milliseconds. During the car begins to cross the bridge, its status is changed from 'WAITING' to 'PROCESSING'. After successful passing the river, status is updated to 'PROCESSED'.

There are 2 threads used in program: one for each queue. New thread is run if the queue is not empty and there is no thread that currently processes the queue.

To make sure that only one car is currently crossing the bridge, the method for that uses *synchronized* keyword. Moreover, to avoid *races* between threads, program uses concurrent-safe data structures.

5.2. Endpoints (web layer)

As mentioned before, the backend web layer of the application consists of 6 endpoints:

- *GET('/api/cars'):* returns the list of all cars added to the program in JSON format;
- *POST('/api/add-car'):* allows to add a new car into the program; JSON body is required (CarAddRequest class) and should contain following fields:
    - *String name (text),*
    - *Source source (enum; NORTH, SOUTH),*
    - *Int processingTime (number);*
- *GET('/api/direction):* returns the direction from which cars are currently passing the bridge;

- *GET('/api/max-cars):* returns the current amount of cars that will be let to pass from one side;
- *POST('/api/max-cars):* allows to set new max-cars. Requires a number in body;
- *SCHEDULED: performs checks if running new threads is necessary and deletes all cars that have passed the bridge for at least 10 seconds;*

All endpoints doesn't require any authentication or headers to be accessed and are also doesn't have any specified allowed domains, which mean they can work with any front-end code from all domains.

5.3. Service (server layer)

The most important class of the program is *BridgeService*. It has 7 private fields:

- *Queue<Car> cars*: queue of all cars added to the program,
- *Queue<Car> northQueue*: queue of cars coming to the bridge from the north,
- *Queue<Car> southQueue*: queue of cars coming to the bridge from the south,
- *int maxCarsAmount*: stores the amount of cars that can pass from one side – default value is 2,
- *Source currentDrivingSource*: stores the direction which currently has green light,
- *Thread southThread*: thread for the south queue,
- *Thread northThread*: thread for the north queue.

It has also 3 public methods: *addToQueue(Car car)* which allows to add a new car to queue, *runThreads()* that makes and starts new threads if necessary and *deleteProcessed* which is called to remove all processed cars from list.

Private method *processQueue(Queue<Car> queue)* mentioned before is method that comes through the queue passed as an argument and if they are present, allows them to pass the bridge.

Private method *CrossBridge(Car currentCar)* is method that simulates the passing the bridge by the car.

6. Summary

The implemented multithreading solution in Java addresses the bridge problem in a two-way road scenario. The web application, using React on the front-end and Spring Boot on the back-end, allows users to spawn cars from the north or south, each with a customizable crossing time.

The backend employs a simple REST controller with endpoints for adding cars to queues and retrieving their states. Program manages the workflow, employing synchronized threading to ensure only one car crosses the bridge at a time.

Overall, the system provides a visual representation of the classic synchronization problem in a web-based environment, enhancing understanding of multithreading concepts.