Project Clojure: Flight Reservation

Multicore Programming 2023

Janwillem Swalens (janwillem.swalens@vub.be)

For this project, you will implement and evaluate a parallel, thread-safe flight reservation system. The system consists of several flights, each of which has many seats at different prices. Customers attempt to book seats on these flights, by trying to find seats within their budget. Multiple customers attempt to book seats concurrently. Your task is to ensure that these requests can be handled in parallel and that correctness is guaranteed.

Overview

This project consists of three parts: an **implementation** in Clojure, an **evaluation** of this system using benchmarks, and a **report** that describes the implementation and evaluation.

- **Deadline**: Sunday, **4th of June** 2023 at 23:59.
- **Submission**: Package the implementation, your benchmark code, and the report as a PDF into a single ZIP file. Submit the ZIP file on the Canvas page of the course.
- **Grading**: This project accounts for one third of your final grade. It will be graded based on the *submitted code*, the accompanying *report and its evaluation*, and the *project defense* at the end of the year. If you hand in late, two points will be deducted per day you were late. If you are more than four days late, you'll get an absent grade for the course.
- **Academic honesty**: All projects are individual. You are required to do your own work and will only be evaluated on the part of the work you did yourself. We check for plagiarism. More information about our plagiarism policy can be found on the course website.

A flight reservation system

The project consists of a flight reservation system, in which customers can book seats on flights. The input consists of flights and customers. A **flight** looks like this:

```
{:id 0
 :from "BRU" :to "ATL"
 :carrier "Delta"
 :pricing [[600 150 0] ; price; # seats available; # seats occupied
           [650  50 0]
           [700  50 0]]}
```

Each flight has an id, an origin and destination airport, a carrier, and pricing information. The pricing information is a list of triples `[price, available seats, occupied seats]`. For instance, in the example above there are 150 seats available at €600, 50 seats at €650, and 50 seats at €700. No seats have been reserved yet. Each flight is uniquely identified by its id. There can be several flights between the same two airports, by the same or by different carriers.

A **customer** is represented using the following map:

```
{:id 0 :from "BRU" :to "ATL" :seats 5 :budget 600}
```

This customer would like to book 5 seats on a flight from Brussels to Atlanta. They are willing to spend at most €600 per seat. Hence, as long as 5 seats are available at €600 in the flight above, they can be reserved by this customer. The flight is then updated to indicate that these 5 seats are no longer available but occupied:

```
{:id 0
 :from "BRU" :to "ATL"
 :carrier "Delta"
 :pricing [[600 145 5] ; price; # seats available; # seats occupied
           [650  50 0]
           [700  50 0]]}
```

If no more seats are available within the customer's budget, on any flight between the chosen origin and destination, no seats are reserved and the customer's reservation fails.

Furthermore, there is an extra addition to the system: every once in a while, a **sales period** starts. At the start of the sales, a random carrier is chosen, and all prices for this carrier are decreased by 20%. When the sales period ends, the original prices are restored. The duration of the sales period and the time between sales can be configured using parameters.

Implementation

The aim of this project is to parallelize the flight reservation system. We provide a sequential implementation, in which customers are processed one by one, using an atom to encapsulate mutable state. You should change the implementation to process multiple customers concurrently. Herein, the focus is first on correctness, second on performance.

For **correctness**, you should ensure that concurrent access to shared data structures is thread-safe, and does not lead to corrupt data. For example, flights should not be overbooked (more seats reserved than there are available on the flight), customers should only make one reservation, after a sales period the original price should be restored, etc. Additionally, when a customer is looking for flights, there should either be no discounted flights, or all discounts should be applied. It should not be possible for a customer to observe discounted and non-discounted flights of the same carrier. You can add unit tests to check whether these conditions are satisfied.

For **performance**, you should attempt to maximize the throughput, i.e. process all customers in minimal time. As the number of cores increases, you expect throughput to increase.

You can use any of Clojure's concurrency mechanisms to implement this, e.g. futures, agents, refs, atoms, and/or promises. It is up to you to select the most appropriate mechanism(s) taking into account these two concerns.

This assignment is accompanied by several files. `flight_reservation.clj` contains a sequential implementation of the project, as a starting point. It uses an atom to encapsulate the flights, but you are free to change this. `input_simple.clj` contains simple example input, and `input_random.clj` contains randomly-generated input. Furthermore, the package contains the same `clj` script and `libs` folder that we used in the lab sessions.

Evaluation

Next to your implementation, you should perform a thorough evaluation of your application. Your evaluation should focus on two points: validating the correctness of your implementation and evaluating its performance.

Correctness

You should validate the correctness of your implementations by creating a number of tests that simulate different scenarios. They should confirm that no race conditions can happen, and that no corrupt states (e.g. overbookings) can be reached.

Performance evaluation

To evaluate performance, you should create a number of experiments that measure the throughput (how long it takes to process all customers' orders) when varying a parameter, such as the number of flights, customers, and carriers, the length/frequency of sales periods, the number of customers that are processed in parallel etc.

You should perform three experiments: one in which you vary the number of threads, and two others in which you can vary any other parameter you like.

Each experiment can be based on a different scenario, i.e. a different "base line" of number of flights, customers, and carriers. These can represent a "typical" scenario, a "best case" scenario, or another scenario you are interested in testing.

To allow your results to be interpreted correctly, vary only one parameter per experiment. Explain how the varied parameter affects the results. Relate this to the chosen concurrency mechanism, e.g. how much contention is there on shared data structures, does a scenario cause transactions to conflict more often, etc.

In case you use atoms or transactions, it may also be useful to measure the number of times the `swap!` block or transaction is re-executed.

Remember that correctness is more important than performance: even in the presence of multiple concurrent clients, and even in extremely unlikely situations, the system should not return an invalid result. For your grade, we will accordingly value correctness over performance.

Report

Finally, you should write a report about your implementation and evaluation. Please follow the outline below:

1. **Overview**: Briefly summarize your overall implementation approach and the experiments you performed. (1 or 2 paragraphs)

2. **Implementation**: Describe the implementation on an abstract level. Use illustrations or code snippets to guide the reader where necessary. Answer the following questions:

- Which concurrency mechanism(s) (e.g. refs, agents, atoms) did you use and why?
- How did you parallelize the application (i.e. which mechanism did you use to create multiple execution threads)?
- How do you ensure correct, thread-safe concurrent access to the system?
- What are the potential performance bottlenecks?

3. **Evaluation**:

    - Which tests did you use to verify **correctness**?
    - Describe your **experimental set-up**, including all relevant details (use tables where useful):
        - Which hardware, platform, versions of software, etc. you used. (Include the version of the JVM as well as the version of Clojure.)
        - All details that are necessary to put the results into context.
    - Describe your **experimental methodology**:
        - How often did you repeat your experiments?
    - For your **experiments**, describe:
        - What (dependent) *variable(s)* did you measure? For example: throughput or time to process all customers.
        - What (independent) *parameter* did you vary? For example: the number of threads, the number of flights, customers, or carriers, or the length and frequency of sales periods.
        - *Report* results appropriately: use diagrams (e.g. box plots), report averages or medians and measurement errors. Graphical representations are preferred over large tables with many numbers. Describe what we see in the graph in your report text.
        - *Interpret* the results: explain why we see the results we see. Relate the results back to the changes you made. What is the optimal value for the measured parameter (e.g. optimal work group size)? If the results contradict your intuition, explain what caused this.

4. **Insight questions**: In this section you answer some insight questions. They relate to extensions to the problem and how you would change your implementation to deal with them. You should *not* actually implement these extensions or perform any experiments. Briefly answer the questions below (max. 250 words, or ~2 paragraphs, each):

    1. What is the effect of the "sales period" on your solution? If there was no such requirement, how would performance be affected?
    2. Imagine you would have implemented this system using traditional locks instead of the chosen concurrency mechanism. Briefly sketch how you would have done this. What are the benefits and drawbacks of this approach? (E.g. regarding ease of implementation, performance.)