Project Clojure: Flight Reservation System

Assistant: Janwillem Swalens
Mail: jswalens@vub.be
Office: 10F737

Deadline: 14 May 2017

Abstract

The goal of this programming project is to implement 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.

1 Procedure

This project consists of three parts: the implementation of a concurrent flight reservation, an evaluation of its correctness and performance, and a report that describes your solution and evaluation.

Deadline 14th of May at 23:59 CEST.

Deliverables Package the implementation into a single ZIP file, including the report as a PDF. The ZIP file should be named `Firstname-Lastname-clj.zip`. On the PointCarré page of the course, go to *Assignments (Opdrachten)* > *Project Clojure* and submit the ZIP file.

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.

2 A Flight Reservation System

The project comprises a 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 at € 650, and 50 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. He is 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.

3 Implementation: Parallel, Thread-Safe Flight Reservation System

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.

4 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.

To validate **correctness**, create 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.

To evaluate the **performance**, create a number of experiments that measure the throughput (how long it takes to process all customers' orders) when varying several parameters, such as the number of flights, customers, or different carriers, the length and frequency of sales periods, the number of cores you use etc. 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: how often do transactions or `swap!` blocks re-execute, how many messages are in agents' queues, how much contention is there on shared data structures, etc.

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.

Sharing As opposed to the Erlang project, you are not allowed to share your benchmarking code.

5 Reporting

Your source code should be accompanied by a brief report. Please structure your report according to the following outline:

1. Overview (max. 100 words)
Briefly summarize your implementation approach and the experiments you performed.

2. Implementation (ca. 2 pages)
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? How does this increase throughput?
- How do you ensure correct, thread-safe concurrent access to the system?
- What are the potential performance bottlenecks?

3. Evaluation (ca. 3 pages, without graphs/illustrations)

- Describe how you validated the correctness of your implementation. Which tests did you use?
- Describe how you tested scalability:
  - describe your experimental setup (platform, relevant versions, etc.)
  - describe your experimental methodology: what did you measure, how did you measure it, what parameters did you vary (e.g. number of flights, number of customers, number of cores, length and frequency of sales period, etc.)?
- Report results appropriately, graphically.
- Interpret the results: provide an explanation for the observed behavior. How does your choice of concurrency mechanism influence the results? What are the bottlenecks?

4. Insight questions (max. 250 words each)

Briefly answer the questions below. They relate to variations of the problem and how you'd change your implementation to deal with them. You do not have to implement these variations or perform any experiments.

- What is the effect of the "sales period" on your solution? If there was no such requirement, how would performance be affected?
- You used one concurrency mechanism provided by Clojure (e.g. atoms, refs, agents). Choose another mechanism and briefly sketch what your solution would have looked like using that mechanism.