Project Clojure: E-commerce Shop

Multicore Programming 2024

Janwillem Swalens (janwillem.swalens@vub.be)

For this project, you will implement and evaluate a parallel version of an e-commerce system. The system contains several third-party stores that offer many products at different prices. Customers attempt to buy their products in the cheapest store. When multiple customers attempt to search for and buy products concurrently, your task is to ensure that their orders are handled in parallel while 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, **2nd of June** 2024 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.

Application: An E-Commerce Shop

The project consists of an e-commerce shop like Amazon or Bol.com, which offers products from several third-party stores for sale. The input thus contains a list of **products** and **stores**, e.g:

```
(def products ["Apple" "Avocado" "Banana" "Pear"])
(def stores ["Aldi" "Carrefour" "Colruyt" "Delhaize" "Lidl"])
```

Each store has different **prices** for the different products, as illustrated in the table below. The columns correspond to the stores and the rows to the products. For instance, an apple in Aldi costs €0.25 (first row, first column).

```
(def prices
  ; Aldi Carr Colr Delh Lidl
  [[0.25 0.30 0.28 0.29 0.27]   ; Apple
   [1.37 1.20 1.25 1.20 1.32]   ; Avocado
   [0.41 0.35 0.35 0.36 0.45]   ; Banana
   [0.19 0.21 0.19 0.25 0.18]]) ; Pear
```

Furthermore, we also keep track of how many of each product are still available in each store. This is tracked in the **stock** table, as shown below. For instance, Lidl has 20 pears in its stock (last row, last column).

```
(def stock
  ; Aldi Carr Colr Delh Lidl
  [[  15   25   30   29   15]   ; Apple
   [   5    7    6   10    2]   ; Avocado
   [   2   10   20   17    8]   ; Banana
   [  25   17   31   18   20]]) ; Pear
```

A **customer** is interested in buying several products. A customer is represented using a map:

```
{:id 1 :products [["Apple" 7] ["Banana" 5]]}
```

This customer, with ID 1, would like to buy 7 apples and 5 bananas. To process their shopping list, a customer goes through three steps:

1. First, the customer looks at which stores still have the requested products in their stock. For instance, customer 1 cannot go to Aldi, as Aldi has only 2 bananas in stock while the customer needs 5. This results in a filtered list of stores.
2. Next, the customer calculates the price of his shopping list in all found stores, to find the cheapest one. (The customer buys all products in one store.) For instance, 7 apples and 5 bananas cost €3.85 in total in Carrefour, €3.71 in Colruyt, etc. In this example, Colruyt is the cheapest store for the requested items.
3. Finally, the customer buys the products by updating the stock table. In the example, this yields (note the two updated cells in the Colruyt row):

```
(def stock
  ; Aldi Carr Colr Delh Lidl
  [[  15   25   23   29   15]   ; Apple
   [   5    7    6   10    2]   ; Avocado
   [   2   10   15   17    8]   ; Banana
   [  25   17   31   18   20]]) ; Pear
```

These three steps are repeated for every customer.

Furthermore, there is an extra addition to the system: every once in a while, one of the stores decides to do a **sales period**. At the start of the sales, a random store is chosen, and the prices of all products in that store are decreased by 10%. 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 e-commerce system. We provide a sequential implementation, in which customers are processed one by one, using atoms to encapsulate mutable state (the prices and stock). 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, customers should not be able to buy products that are sold out (leading to negative numbers in the stock table), a customer's order should only be processed once, after a sales period the original price should be restored, etc.

Most importantly, during the three steps of the buying process, a customer should not see inconsistent data. This means, firstly, that when a customer looks for stores that have the requested products in stock in step 1, they should still be available for him in step 3. Secondly, when a customer looks for the cheapest store in step 2, these prices should still apply in step 3, regardless of the start/end of a sales period. A customer must therefore see the sales period applied to all prices of the discounted store, or none. You should add unit tests to check whether these kind of 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 their characteristics (as seen in class) and the requirements of the application.

This assignment is accompanied by several files. `web_shop.clj` contains a sequential implementation of the project, as a starting point. It uses atoms to encapsulate the prices and stock tables, but you are free to change this. `input_simple.clj` contains simple example input, and `input_random.clj` contains randomly-generated input. You can use these as a starting point for your evaluation, to come up with more scenarios. 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: correctness and performance.

Correctness

You should validate the correctness of your implementation 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. negative numbers of a product in stock) 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 several parameters, such as the number of products, stores, and customers, the number of products in each customer's shopping list, the number of products in stock, 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 to measure throughput, 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 products, stores, and customers. 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 there is on shared data structures, whether a scenario causes transactions or `swap!` blocks to conflict more often, etc. If 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.

Sharing: To have a wider range of experiments, you may share your *input files* with other students. You can do this through the forum on the course page on Canvas. However, make sure to only share the input data, consisting of the data structures `products`, `stores`, `prices`, and `stock`, similar to the files `input_simple.clj` and `input_random.clj`! Remember that every student should have their own implementation of the system, their own code to actually run the benchmarks, and their own report.

Report

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

1. **Overview**: Briefly summarize your implementation approach and the experiments you performed. (1 or 2 paragraphs)
2. **Implementation**: Describe your implementation. 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?
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: execution time, throughput, number of conflicts.
     - What (independent) *parameter* did you vary? For example: number of threads; number of products, stores, customers; length, frequency of sales period.

- *Report* results appropriately: use diagrams (e.g. box plots), report averages or medians and measurement errors. 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. How does your choice of concurrency mechanism influence the results? 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 performance, ease of implementation.)