Multi-Threading Programming and Inter-Process Communication

M.Sc. course on “Technologies for Big Data Analysis” - Assignment 1

CHRISTOS BALAKTSIS (506) and VASILEIOS PAPASTERGIOS (505), Aristotle University, Greece

1 Introduction

The current document is a technical report for the first programming assignment in the M.Sc. course on *Technologies for Big Data Analysis*, offered by the *DWS M.Sc Program*[1] of the Aristotle University of Thessaloniki, Greece. The course is taught by Professor Apostolos Papadopoulos [2]. The authors attended the course during their first year of Ph.D. studies at the Institution.

The assignment contains 4 sub-problems and is part of a series, comprising 3 programming assignments on the following topics:

Assignment 1 Multi-threading Programming and Inter-Process Communication
Assignment 2 The Map-Reduce Programming Paradigm
Assignment 3 Big Data Analytics with Scala and Apache Spark

In this document we focus on Assignment 1 and its 4 sub-problems. We refer to them as *problems* in the rest of the document for simplicity. The source code of our solution has been made available at `https://github.com/Bilpapster/big-data-playground`.

Roadmap. The rest of our work is structured as follows. We devote one section for each one of the 4 problems. That means problems 1, 2, 3 and 4 are presented in sections 2, 3, 4 and 5 respectively. For each problem, we first provide the problem statement, as given by the assignment. Next, we thoroughly present the reasoning and/or methodology we have adopted to approach the problem and devise a solution. Wherever applicable, we also provide insights about the source code implementation we have developed. For problems 2 and 4, we complete the respective sections with a discussion about alternatives or improvements the solution could accept, in order to successfully support more complex requirements. Finally, we conclude our work in section 6.

2 Problem 1: Concurrent Array-Vector Multiplication

We discuss here the first problem of the assignment. The main target of the assignment is using multi-threading programming in Java programming language to concurrently perform an algebraic operation.

2.1 Problem Statement

It is known that in Linear Algebra we can multiply a matrix with a vector from the right-hand side, provided that the number of columns in the matrix equals the number of rows in the vector. For instance, given a matrix $\mathbf{A}$ with dimensions $n \times m$ and a vector v with dimensions $m \times 1$, then the product $\mathbf{A} * \mathbf{v}$ is an $n \times 1$ vector, which

[1]https://dws.csd.auth.gr/
[2]https://datalab-old.csd.auth.gr/~apostol/

Authors’ Contact Information: Christos Balaktsis (506), balaktsis@csd.auth.gr; Vasileios Papastergios (505), papster@csd.auth.gr, Aristotle University, Thessaloniki, Greece.

results from the implementation of the well-known method of multiplying a matrix with a vector. An example is given following:

$$\begin{bmatrix}1 & 0 & 2 & 0\\ 0 & 3 & 0 & 4\\ 0 & 0 & 5 & 0\\ 6 & 0 & 0 & 7\end{bmatrix} \cdot \begin{bmatrix}2\\5\\1\\8\end{bmatrix} = \begin{bmatrix}4\\47\\5\\68\end{bmatrix}$$

Provided that we are capable of using k threads, where k is a power of 2 and the matrix has dimensions $n \times m$, where n is also a power of 2 and $n > k$, design a solution that computes the product $\mathbf{A} * \mathbf{v}$ using k threads with the most efficient way. Your solution has to initialize both the matrix $\mathbf{A}$ and the vector $\mathbf{v}$ with random numbers in the range $[0, 10]$.

2.2 Proposed approach

2.2.1 Setting. Our implementation is run and tested in a Linux environment with 12 cores, using the Java programming language. We have used the Java Development Kit (JDK) version 11.0.11. The source code is developed in IntelliJ IDEA Community Edition 2021.1.1. The code is compiled and executed in the terminal, using the following commands, starting from the root folder of the repository:

```
cd MatrixMultiplication
javac MatrixMultiplier.java <n> <m> <k>
java MatrixMultiplier
```

where:

- <n> is the total number of rows of matrix **A** and vector **v**
- <m> is the total number of columns of the matrix
- <k> is the total number of threads

Note that the command line arguments have to be exactly 3 and all individual arguments should be integer parsable. An exception will be thrown, in case any of the aforementioned conditions is violated, accompanied by an explanatory message.

2.2.2 Implementation. The implementation of the solution proceeds as follows.

First, the main thread initializes the matrix **A**, of size $n \times m$, and the vector **v**, of size $n \times 1$, with random numbers in the range $[0, 10]$. Next, the main thread creates k threads, where k is a power of 2. All threads are initiated with references to the matrix **A**, the vector **v**, and an empty vector **result**. The latter is used to store the result of the product of the assigned rows of the matrix with the vector.

Each thread is assigned a range of rows of the matrix **A**, determined by the *start* and *end* indices, to process. The range is calculated based on the total number of rows and the number of threads, i.e., $\lfloor n/k \rfloor$, with the last thread being assigned any remaining rows, covering cases off the assumption that n, k are powers of 2. Each thread then computes the product of the assigned rows of the matrix **A** with the vector **v** by iterating over the assigned rows and multiplying the corresponding row of the matrix with the vector. It is worth noting that although the thread retains access to the whole matrix, it only accesses the indicated rows. Finally, the result of this computation is stored in the corresponding indices of the vector **result**, which is shared among all threads, so it is filled concurrently.

Once all threads have completed their assigned computations, the main thread waits for them to finish. It then collects the results from all the threads and prints the final result. In the end, the main thread also prints the total time taken to compute the whole product, which is the time taken for all the threads to finish their computations, excluding the time taken for the initializations of the structures and threads.

2.2.3 Evaluation. Our implementation is tested for multiple matrix-vector sizes on the system described in section 2.2.1. A test class, `TestMatrixMultiplier`, can be found under the `/test` directory. We both evaluated the time difference between non-threading and threading executions, and ensured the $\mathbf{A} * \mathbf{v}$ product results are the same in each case.

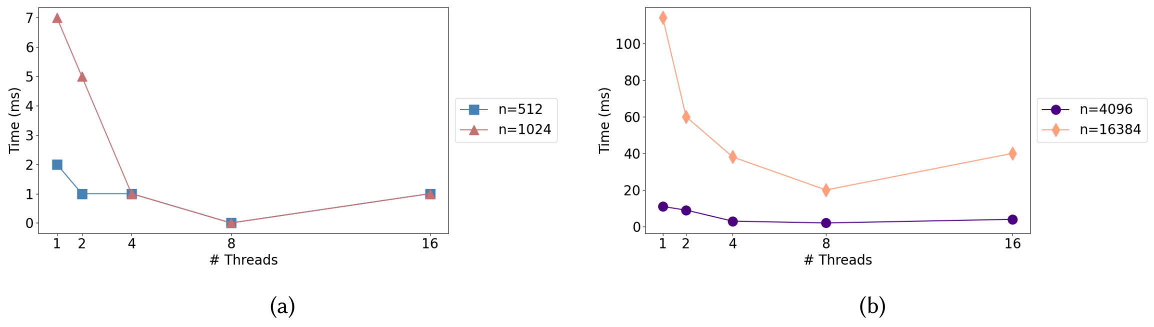


Fig. 1. Execution time for threaded matrix-vector multiplication for various numbers of threads and sizes.

Figure 1 exhibits how execution time changes for different values of n, while the number of threads (k) increases. It should be noted that the column size is considered constant. In fact, all time values refer to the mean time for different values of m, which, in deed, did not exhibit any significant variance at all. By taking a closer look to the plots, the execution time significantly decreases as the number of threads increases from 1 to 4 for all matrix sizes n. However, the impact of further increasing the number of threads from 4 to 16 diminishes, with only marginal improvements or stabilization observed. S pecifically:

- For smaller matrix sizes ($n = 512$ and $n = 1024$), there is a rapid reduction in execution time up to 4 threads, after which the time stabilizes with little improvement, and even slightly increases in both cases.
- For larger matrix sizes ($n = 4096$ and $n = 16384$), a similar trend is observed.
- The initial drop in execution time with up to 8 threads is pronounced, but adding more threads beyond that point does not yield significant time reductions. For $n = 16384$, the time even appears to increase slightly as the thread count grows beyond 8, as well as for $n = 512$ and $n = 1024$.

This behavior indicates that for the system under test, threading provides substantial performance gains up to a certain point (around 8 threads), beyond which the benefits plateau, suggesting the presence of bottlenecks or diminishing returns from parallelism. Overall, this is an expected outcome, taking into consideration that the number of available cores in CPU are 12.

3 Problem 2: Race against a pandemic

The second problem asks for leveraging multi-threading programming in Java programming language, in order to simulate a pandemic spread.

3.1 Problem Statement

In this problem, you have to simulate a (simplified) pandemic spread, taking into account new infections, hospitalizations and recoveries occurring concurrently. In particular, suppose that a thread named `DISEASE` produces periodically (e.g., every 1 second) a random number of new infected people in the range $(0, k]$ that are in need of hospitalization in ICU[3]. The Health Care System has a limited number of ICU beds, let e (e.g., 20). At the same time, a thread named `HOSPITAL` periodically (e.g., every 5 seconds) treats a random number of infected patients in the range $(0, h]$, where $h < k$. When a patient is treated the ICU bed used for their treatment is no longer occupied; thus available for use by another patient.

Develop a solution that simulates the above behavior. Your solution should also keep track of the total number of treated patients, as well as the ones that were not able to find a spot in ICU. The simulation must complete its execution after a predefined number of steps (or time period), which will be given as program argument, alongside with all aforementioned parameters (periods, k, h, e). You should test your solution against different values of the parameters, in order to verify that it correctly operates in all possible cases. How would you convert your solution if, instead of a single hospital, there were three of them, with a single, shared queue of patients?

3.2 Proposed approach

3.2.1 Setting. Our implementation is run and tested in both a Linux (Ubuntu 24.04) and a Windows 11 environment, using the Java programming language. We have used JDK 11.0.11 and IntelliJ IDEA Community Edition 2021.1.1. The code is compiled and executed in the terminal, using the following commands, starting from the root folder of the repository:

```
cd raceAgainstPandemic
javac RaceAgainstPandemic.java <dur> <beds> <d_per> <inf> <h_per> <tr>
java RaceAgainstPandemic
```

where:

- `<dur>` is the total duration to run the simulation
- `<beds>` is the total number of available ICU beds
- `<d_per>` is the time period between two infection steps
- `<inf>` is the maximum number of infections per step
- `<h_per>` is the time period between two treatment steps
- `<tr>` is the maximum number of treatments per step

Note that the command line arguments have to be exactly 6 and all individual arguments should be integer parsable. An exception will be thrown, in case any of the aforementioned conditions is violated, accompanied by an explanatory message.

3.2.2 Implementation. We develop our solution on top of two custom-defined classes, namely `Disease` and `Hospital`. Both classes are defined as subclasses of the Java's `Thread` class. We encapsulate the logic of the two classes in the overridden `run()`. In particular,

- The `Disease` thread repeats two actions: (1) creates a new number of infections and then (2) sleeps for `<d_per>` seconds.
- The `hospital` thread follows the same scheme, with the difference that it populates new treatments, instead of infections.

The communication of these two threads is accomplished via a third, custom-defined Java class, namely `HealthCareManager`. We implement this class employing the *Singleton* design pattern, to ensure that only one instance of this class will be created at runtime and this instance will be shared among all running threads. Inside this class, we encapsulate the logic of handling new infections and treatments when they occur by the `Disease` and `Hospital` threads respectively. The **key functionality** of the `HealthCareManager` class is to

[3]ICU: Intensive Care Unit

ensure **synchronization and mutual exclusion** between the two threads, when the counters are accessed and modified. The critical counters the `HealthCareManager` instance updates are the following:

- `totalBeds`: the total number of ICU beds in the Health Care System
- `availableBeds`: the number of currently available ICU beds
- `currentlyInICU`: the number of patients *currently* in ICU
- `currentlyOutOfICU`: the number of patients *currently* waiting for ICU treatment (out of ICU)
- `totalTreatments`: the total number of patients treated from the start of the simulation up to that moment
- `totalInfections`: the total number of people infected from the start of the simulation up to that moment

We refer to these counters as *critical*, since both threads are competing for accessing and updating them in a contradictory way. In particular, the `Disease` thread produces new infections, increasing `totalInfections` and possibly `currentlyInICU`, `currentlyOutOfICU`, while possibly decreasing `availableBeds`. The `Hospital` thread, on the other hand, produces new treatments with the completely opposite effects.

In order to ensure that the counters are accessed and updated consistently by the two threads, we extract all updates to a dedicated, `synchronized` method, inside the `HealthCareManager`. We note that, when the Java keyword `synchronized` is used in a method, Java ensures that **exactly one** thread can have access to this method, while all others competing for it are locked out (waiting), until the former thread exits the method. We refer the interested reader to the `handleUpdate()` method of the `HealthCareManager` class for the exact implementation.

3.2.3 Testing thread-safety. We discuss here the methodology we have adopted to test the thread-safety property of our implementation. We implement our testing methodology on top of the following property that derives from the problem statement:

Property The patients in ICU, the patients waiting for (i.e., out of) ICU and the total number of treated people must add up to the total number of infections. The property is formally expressed in equation 1.

$$\text{currentlyInICU}_t + \text{currentlyOutOfICU}_t + \text{totalTreatments}_t = \text{totalInfections}_t \; \forall \; t \in [t_{start}, t_{end}], \tag{1}$$

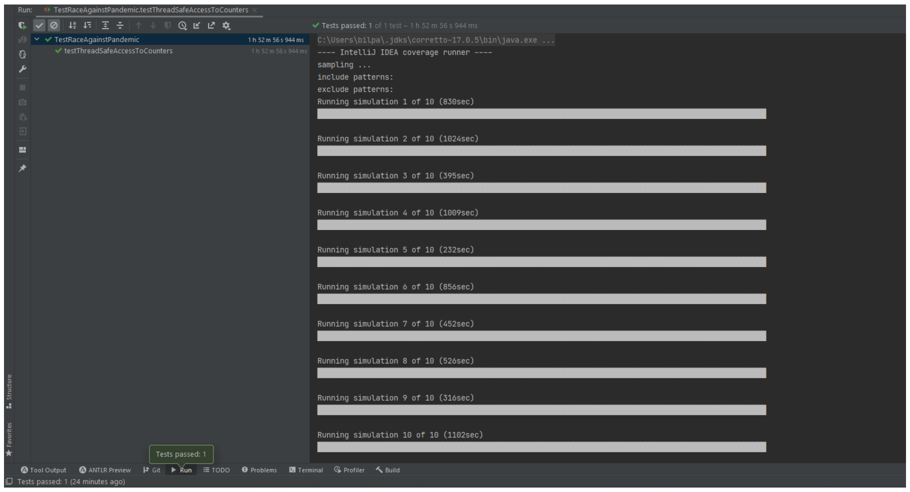
where t_{start} and t_{end} are the start and end timestamp of the simulation respectively and the subscript t indicates the timestamp that the counters are accessed at.

We leverage this property to test and evaluate the thread safety of our implementation. In order to do so, we have used the Java's `junit` library for unit testing. More specifically, we define a test case where a simulation is run. The counters are accessed at the end of the simulation, checking whether the aforementioned property holds. To strengthen our experimental testing, we opt for executing the test 10 times with random initial configurations each time. A snapshot of the test results is depicted in figure 2, where all 10 runs of the simulation (with random initial configurations each) are successfully passed (i.e., equation 1 holds at the end of all simulations). We refer the interested reader to the `test/TestRaceAgainstPandemic.java` file for more details about the implementation of the unit test.

3.2.4 Screenshots and Visualizations. In this subsection we briefly present execution screenshots and visualizations of the simulation. In particular, figure 3 shows the execution of a simulation inside IntelliJ IDEA. The program prints the simulation configurations and shows a progress bar during the simulation run. When the simulation is over, the program outputs the counter values.

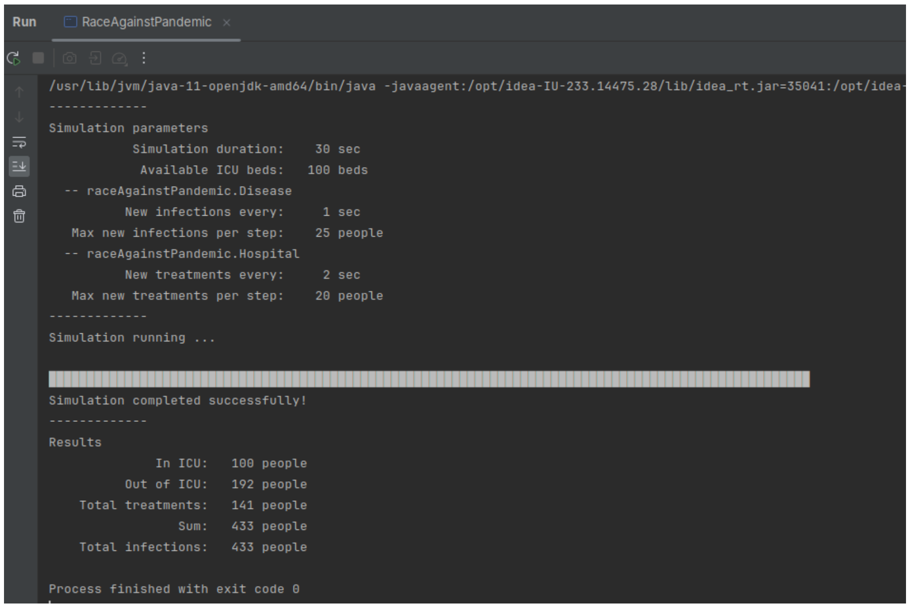
Concerning visualizations, figure 4a depicts the number of patients in and out of ICU over time for a single simulation with duration of 10 minutes and 100 available ICU beds in total. Figure 4b shows the cumulative number of infections and treatments over time for the same setting.

3.2.5 Limitations of current implementation and further improvements. The current implementation successfully addresses the `DISEASE - HOSPITAL` problem when *exactly one* thread of both entities are active every time moment. This means there is only one `DISEASE` thread that periodically produces new infections and only one `HOSPITAL`


Run: TestRaceAgainstPandemic.testThreadSafeAccessToCounters
TestRaceAgainstPandemic 1 h 52 m 56 s 944 ms
testThreadSafeAccessToCounters 1 h 52 m 56 s 944 ms
Tests passed: 1
Tool Output
ANTLR Preview
Git
Run
TODO
Problems
Terminal
Profiler
Build
Structure
Favorites

```
Tests passed: 1 of 1 test – 1 h 52 m 56 s 944 ms
C:\Users\bilpa\.jdks\corretto-17.0.5\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
exclude patterns:
Running simulation 1 of 10 (830sec)

Running simulation 2 of 10 (1024sec)

Running simulation 3 of 10 (395sec)

Running simulation 4 of 10 (1009sec)

Running simulation 5 of 10 (232sec)

Running simulation 6 of 10 (856sec)

Running simulation 7 of 10 (452sec)

Running simulation 8 of 10 (526sec)

Running simulation 9 of 10 (316sec)

Running simulation 10 of 10 (1102sec)

Tests passed: 1 (24 minutes ago)
```

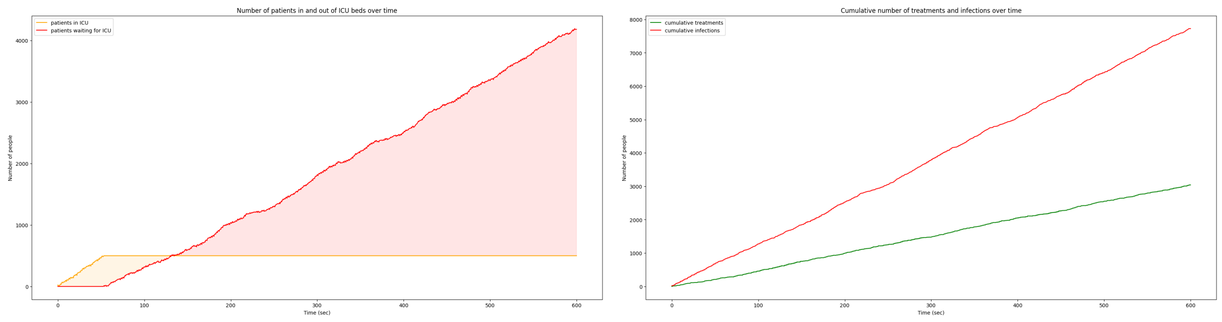
Fig. 2. Test results for thread-safety. Ten simulations are run with random initial configurations and equation 1 is tested at each simulation end. All tests are successfully passed.



```
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -javaagent:/opt/idea-IU-233.14475.28/lib/idea_rt.jar=35041:/opt/idea-
-------------
Simulation parameters
           Simulation duration:    30 sec
            Available ICU beds:   100 beds
  -- raceAgainstPandemic.Disease
          New infections every:     1 sec
   Max new infections per step:    25 people
  -- raceAgainstPandemic.Hospital
          New treatments every:     2 sec
   Max new treatments per step:    20 people
-------------
Simulation running ...


Simulation completed successfully!
-------------
Results
              In ICU:   100 people
          Out of ICU:   192 people
    Total treatments:   141 people
                 Sum:   433 people
    Total infections:   433 people


Process finished with exit code 0
```

Fig. 3. Execution of a simulation inside IntelliJ IDEA.

thread that periodically treats some of them. We can imagine the new infections and treatments at every step as a sequence of requests, which are handled by the Health Care System. The latter lives in the main thread of our application. Thanks to the use of the `synchronized` Java keyword, the Health Care System is able to handle the requests as if they were put in an imaginary **single queue**, where only one request is processed at a time and the inner critical counters are accessed and modified consistently.



(a) Number of patients in (orange) and out of ICU (red) over time

(b) Cumulative treatments (green) and infections (red) over time

Fig. 4. Visualizations for a single 10-minute simulation with 100 available ICU beds, at most 25 new infections every 1 second and at most 20 new treatments every 2 seconds.

A more generic version of the problem could contain multiple `DISEASE` and multiple `HOSPITAL` threads running at the same time, each one of them producing a random, independent number of new infections and treatments respectively. In such a case, the Health Care System handles again all produced requests as if they were put in a single queue, where only one request modifies the critical counters at a time. **We claim that our implementation can successfully handle this generic version of the problem** with trivial changes. The only requirement for our implementation to work properly in such a scenario is that all running `DISEASE` and `HOSPITAL` threads are provided with the same instance of the `HealthCareManager` class. This requirement holds, since we have implemented the class using the **Singleton** design pattern, ensuring that it can be instantiated only once and all other objects share this instance.

Apart from this requirement, only few trivial changes need to be made. In particular, more `Disease` and `Hospital` instances should be created, meeting the needs of the generic problem statement. The mutual exclusion ensured by the `synchronized` character of the processing method simulates the handling of requests as if they were in a single queue. **We verify our claim by executing unit tests with multiple `DISEASE` and multiple `HOSPITAL` threads running at the same time**. We run 10 test simulations with random initial configurations each. All tests are successfully passed. The tests' implementation can be found in `test/TestRaceAgainstPandemic.java` file and, in particular, in `testMultiDiseaseMultiHospitalVersion()` method.

4 Problem 3: Key-value server store

We present here the third problem of the assignment. The main focus of the problem is the inter-process communication, leveraging TCP sockets in Java programming language.

4.1 Problem Statement

Develop a client-server application that works as follows.

Server. The server starts its operation by initializing an empty Hash Table. The Hash Table stores key-value pairs and has size of 2^{20}. Next, the server opens a TCP port, the number of which is passed as a program argument (e.g., 8888, 9999, etc.). The server awaits for clients to connect.

Client. The client starts its operation by trying to connect with the server, in the specified TCP port. Again, the port is passed as a program argument.

Communication protocol. After the successful connection between server and client, the client can send messages to the server, requesting some action from a predefined list of available actions. In particular, the client can perform the following operations:

Operation 0 Terminates the connection between client and server. Operation code: `0`.
Operation 1 Inserts a key-value pair in the Hash Table stored in the server-side. Operation code: (`1, K, V`), where `K` and `V` is the key and value to be inserted respectively.
Operation 2 Deletes a key-value pair from the Hash Table kept in the server-side. Operation code: (`2, K`), where `K` is the key to be deleted, jointly with the stored value that is associated with it.
Operation 3 Searches for a key and retrieves its value (if exists). Operation code: (`3, K`), where `K` is the key to search for.

4.2 Proposed approach

4.2.1 Setting. Our implementation is run and tested in both a Linux (Ubuntu 24.04) and a Windows 11 environment, using the Java programming language. We have used JDK 11.0.11 and IntelliJ IDEA Community Edition 2021.1.1. The code is compiled and executed in the terminal and comprises two parts, namely a Server and a Client. For the server, the code can be compiled and run with the following commands, starting from the root folder of the repository:

```
cd keyValueStoreServerClient
javac Server.java <port>
java Server
```

where `<port>` is the TCP port number the server listens at for incoming requests. The port number defaults to 8899, if no port is specified.

For the client, the code can be compiled and run with the following commands, starting from the root folder of the repository:

```
cd keyValueStoreServerClient
javac Client.java <filename> <port> <server_address>
java Client
```

where:

- `<filename>` is the path of the file to read the requests to execute from,
- `<port>` is the TCP port number the server listens at for incoming requests. Defaults to 8899 if no port is specified.
- `<server_address>` is the IP address of the server and defaults to localhost if no address is specified.

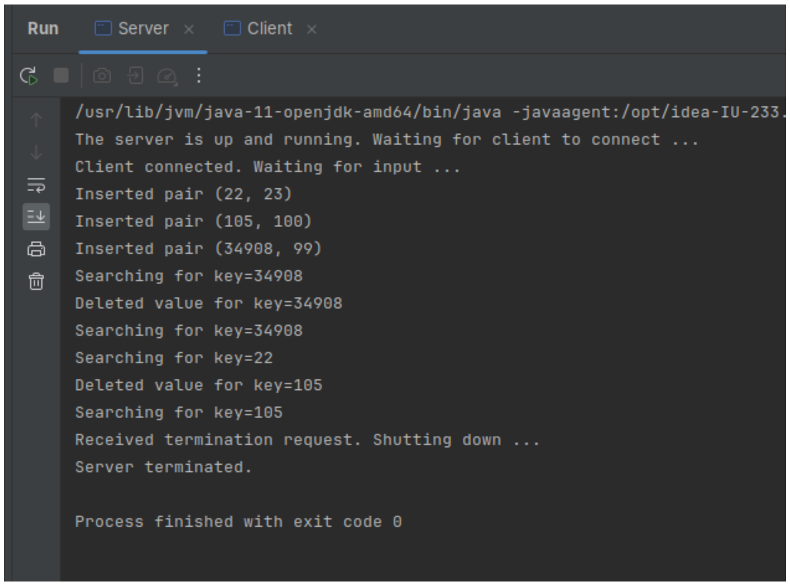
4.2.2 Implementation. We develop our solution on top of two custom-defined classes, namely `Server` and `Client`. In the former, we encapsulate the request handling rational with the various operation codes. In the latter, we implement the sequential reading of the text file lines, passing them as requests to the server-side. More specifically,

- The `Server` class initializes an empty hash table as a Java `HashMap`, creates a web socket and waits for a client to be connected.
- The `Client` class creates also a socket, connects to the server and provides it with the text file contents line by line.

Note that the communication between the two is finished when the client sends a request with operation code 0 to the server. We refer the interested reader to the `keyValueStoreServerClient` package for more details about the implementation of our solution.

4.2.3 Execution Screenshots. In this subsection we briefly present execution screenshots and visualizations of the simulation. In particular, figure 5 shows the execution result of the *Server-side* program. The Server prints informative messages to the console both for the storage initialization part and the sequential requests that it receives from the Client.

Figure 6 shows the execution result of the *Client-side* program for the same input. The client prints informative messages to the console for each one of the requests made to the Server, as well as the received response for them.



```
/usr/lib/jvm/java-11-openjdk-amd64/bin/java -javaagent:/opt/idea-IU-233.
The server is up and running. Waiting for client to connect ...
Client connected. Waiting for input ...
Inserted pair (22, 23)
Inserted pair (105, 100)
Inserted pair (34908, 99)
Searching for key=34908
Deleted value for key=34908
Searching for key=34908
Searching for key=22
Deleted value for key=105
Searching for key=105
Received termination request. Shutting down ...
Server terminated.

Process finished with exit code 0
```

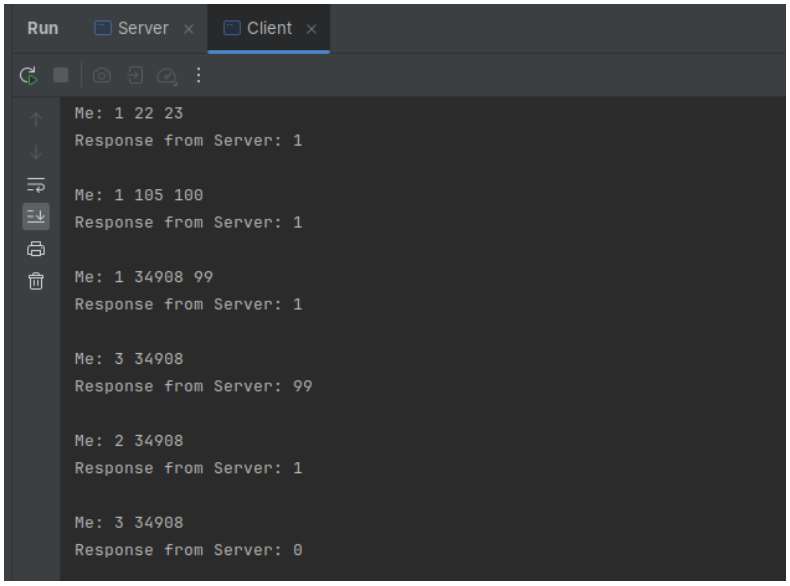
Fig. 5. Execution results of the key-value server program.

4.2.4 Limitations of current implementation and further improvements. The current implementation successfully addresses the *single-client* version of the problem. That means *only one client* can be connected to the server at a time, sending requests and receiving answers.

A more generic variation of the problem at hand raises this limitation. In particular, in this generic variation *several clients* could be connected *at the same time* to the server, sending requests and receiving back answers. **Our solution could support this generic variation of the problem with several modifications/extensions**. We discuss them in detail in the rest of this section.

First, we define the generic variation of the problem in a more formal way. The definition is as follows.

Definition 4.1. **Generic key-value store client-server problem**. There is a single server that stores a centralized hash table with key-value pairs. Multiple clients send requests to this server. Each client is implemented as a separate thread that connects to the server, sends a request and receives an answer. The server must be capable of handling multiple clients at the same time, *assuring the integrity of the key-value store.*



```
Me: 1 22 23
Response from Server: 1

Me: 1 105 100
Response from Server: 1

Me: 1 34908 99
Response from Server: 1

Me: 3 34908
Response from Server: 99

Me: 2 34908
Response from Server: 1

Me: 3 34908
Response from Server: 0
```

Fig. 6. Execution results of the key-value client program.

Based on this definition, we identify two important points in our implementation that should be altered, in order to successfully address the problem. The first is **decoupling request processing from the main server-side thread** and the second is **ensuring synchronization and mutual exclusion of threads in critical operations** of the storage content.

We claim that with the following modifications, our solution could successfully support the generic version of the problem:

- The server spawns a new thread when a new client is connected to it. The thread takes as parameters the socket of the client and the request and its responsibility is to handle the request and respond to the client. This way, the main thread of the server **is not blocked** and can await for more customers to connect. When a new client connects, the same process happens and the client is assigned to a new thread. The requests are served concurrently and can finish in an asynchronous order.
- The Java keyword `synchronized` is added to the signature of the server-side method that handles requests. This way, synchronization and mutual exclusion can be achieved, leveraging Java's thread-safe mechanism. As a result, our application will never be in a state where two client threads try to access and/or **write** a different value for the same key at the same time; thus ensuring thread-safety. In other words, when multiple clients are served at the same time, **only one** client is accessing the critical part of our application at a given time unit and all the other are waiting for it to finish the critical operation.

5 Problem 4: Multi-server producer-consumer interaction

The fourth problem of the assignment concerns a more complex scenario of inter-process communication with multiple servers being active simultaneously. The assignment asks for implementation in Java programming language, leveraging TCP sockets and multi-threading.

5.1 Problem Statement

Suppose that you want to develop a process system that consists of three types of processes: **consume**, **produce** and **server**. All three types of processes communicate with each other using TCP sockets. The system operation is as follows.

Server. Each server has a non-shared integer variable `STORAGE` that depicts the quantity of products in stock, located in the specific server. The initial value of the `STORAGE` for each server is a random integer number in the range $[0, 100]$.

Producer. Each producer connects randomly to a server and adds a random value X in the range $[10, 100]$ to the `STORAGE` variable of the specific server. In case the new `STORAGE` value exceeds 1000, then the respective server sends a message and the value is not updated. Next, the producer awaits for a random period of time in the range $[1, 10]$ seconds and connects to the next server, again randomly.

Consumer. Each consumer randomly connects to a server and subtracts a random value Y in the range $[10, 100]$ from `STORAGE` of this server. If the new `STORAGE` value is below 1, then the respective server sends a relevant message and the `STORAGE` value is not updated. Next, the consumer awaits for a random time period in the range $[1, 10]$ and connects to the next server, again randomly.

5.2 Proposed approach

5.2.1 Setting. Our implementation is run and tested in both a Linux (Ubuntu 24.04) and a Windows 11 environment, using the Java programming language. We have used JDK 11.0.11 and IntelliJ IDEA Community Edition 2021.1.1. The code is compiled and executed in the terminal from a single entry point (i.e., `main()`). running from the root folder of the repository as:

```
cd storageProducerConsumer
javac Server.java Producer.java Consumer.java
java Server <server_count> <host_port>
```

where:

- `<server_count>` is the number of servers that will start
- `<host_port>` is the port number of the first server (the rest take the next ones)

Note that the `server_count` argument is optional (default value is 5), but mandatory if `host_port` is specified (otherwise, its default value is 9000).

5.2.2 Implementation. The `Server` class is central, managing storage and handling client requests through socket connections. Each server has a unique ID, a designated port, and a storage limit, initially set to a random value between 1 and 1000. The `Server` class starts a socket to listen for client connections, with each client handled in its own thread through a thread pool. In other words, the *multi-threading* requirement for client-serving is accomplished by the `CachedThreadPool` each server has.

Key methods in `Server` include:

- `addToStorage` and `removeFromStorage`: These methods add or remove values from storage while ensuring values stay within bounds (1 to 1000).
- `start`: This method begins listening for incoming connections.
- `shutdown`: If a connection timeout is reached (default 60 seconds), the server automatically shuts down.
- `handleClient`: This method processes `ADD` or `REMOVE` requests from clients and sends a response indicating success or failure.

Each server's operations on storage are thread-safe, achieved using **`synchronized`** methods to prevent data inconsistencies when multiple producers or consumers connect simultaneously. This ensures that when a producer/consumer tries to add/remove a value on/from the server's storage, no other producer/consumer is operating on that variable, since both the whole `Server` instance gets locked, until the operation is completed.

Consumer The `Consumer` class models a client that connects to random servers and attempts to remove items from storage, simulating a consumer in the producer-consumer pattern. Each consumer operates for a configurable runtime (defaulting to 60 seconds) and chooses a random server to send a `REMOVE` request with a random value (between 10 and 100). The consumer reads the server's response and waits a randomized interval before making the next request, thus simulating realistic, staggered consumption.

Producer Similarly, the `Producer` class represents a client that attempts to add items to storage. Each producer thread operates with the same configurable runtime and randomly selects a server to send an `ADD` request. Both producers and consumers print the server's response to indicate whether their request was successful.

The main method in `Server` initializes multiple server instances, each with its own thread (where another thread pool for clients is initialized, as mentioned above). Producers and consumers are also created and launched as individual threads. Once all servers and clients are started, the system periodically checks if all servers have shut down. When no active servers remain, the program terminates gracefully.

5.2.3 Technical Highlights. The architecture demonstrates *efficient concurrency handling*, with synchronized storage operations preventing conflicts and socket-based communication enabling seamless interaction between servers, producers, and consumers. On the other hand, we comment that the random server selection approach -as the problem requires- may lead to an *uneven load distribution* across servers, with some servers potentially handling more requests than others. A load-balancing mechanism could help optimize resource utilization.

6 Conclusion

In this document we have presented our solutions and rational for solving the first assignment of the M.Sc. course on *Technologies for Big Data Analysis*, offered by the *DWS M.Sc Program*. The assignment comprises four problems that are closely related to multi-threading programming and inter-process communication in the Java programming language. The first problem considers a concurrent implementation of an algebraic operation. The second problem focuses on a pandemic spread. The third and the fourth problems relate to client-server and producer-consumer settings respectively. For each one of the problems we have presented the problem statement, the approach we adopted in order to devise a solution, as well as the source code implementation of our solutions. Wherever applicable, we have discussed extensions of our work, in order to successfully handle more generic variations of the problems.