The First Mandatory Programming Assignment:

The Java-RMI International Statistics Service

IN5020 Autumn 2023

Objective

1. To develop an object-based distributed system;
2. Using the Java-RMI programming model and remote method invocation;
3. Following the simple client/server architecture with a load balancer;
4. Handling large data sets in a distributed environment;
5. Simulating client-server remote communication on a local machine;
6. Implementing caching mechanisms as a distributed scaling technique.

Scope

The system consists of a distributed International Statistics Service. The Application functionality is provided by a remote object residing at the server side. Client objects interact with the server through remote method invocations. The client can invoke the methods defined in the server's remote interface specification.

The system consists of a load balancing server, and 5 other servers that have access to a International Statistics database described below.

The load balancing server acts as a proxy and is in charge of distributing users' requests to other 5 servers. Clients make a remote method call to the proxy server and it replies with the address and port number of one of the 5 servers to clients. Clients then invoke a remote call to the server that got their address and port and send their request.

We will discuss the technical details of the proxy server and other 5 servers further in this document.

The servers will have access to a database consisting of the statistics dataset. The dataset is available for download from the course website in a single zip archive named statistics _IN5020. The dataset contains information about 140,574 different cities in the world. For more information about the datasets refer the following website: https://public.opendatasoft.com/explore/dataset/geonames-all-cities-with-a-population-1000/. The sample representation of the dataset is given below:

Geoname ID	Name	Country Code	Country name EN	Population	Timezone	Coordinates
2798301	Fleron	BE	Belgium	15994	Europe/Brussels	50.61516,5.68062
2798438	Fauvillers	BE	Belgium	1952	Europe/Brussels	49.85116,5.66405
2798581	Ettelgem	BE	Belgium	1181	Europe/Brussels	51.17984,3.02919
2798615	Essen	BE	Belgium	10000	Europe/Brussels	51.46791,4.46901
2798680	Erondegem	BE	Belgium	1742	Europe/Brussels	50.94118,3.95611
2798868	Eksaarde	BE	Belgium	6005	Europe/Brussels	51.14876,3.96814
2798949	Eghezee	BE	Belgium	14352	Europe/Brussels	50.59076,4.91175

The service you will implement is responsible for parsing the dataset and providing clients with statistics information about different countries. The clients, using remote invocations, can ask the query server for different information it requires.

Technical features

1. Proxy (load balancing) Server

The proxy server distributes the requests based on the zone that each client is located in. To simulate geographical zones in our assignment, each query will be assigned a zone number. We will discuss the format of queries in the next subsection.

We have 5 different geographical zones in our scenario and one server with access to the database in each zone. The proxy server will try to connect a client in a specific zone to the server in the same zone. If the server in that specific zone is overloaded with client requests (equal or more than 18 requests in the waiting list), the load balancing server tries servers in the next two zones (if server in zone 1 is overloaded, then load balancer will try servers in zone 2 and zone 3) to find a server with less than 8 requests in the waiting list. The priority for establishing a communication to the neighbor zone server will be the server with fewer requests in the waiting list. If both servers have the same number of requests in the waiting list, one server will be assigned randomly. If both neighbors are also overloaded (more than 18 requests in the waiting list) the server in the same zone will be assigned for the user request.

Figure 1 illustrates how the proxy server acts in the system and how different geographical zones are connected as neighbors.

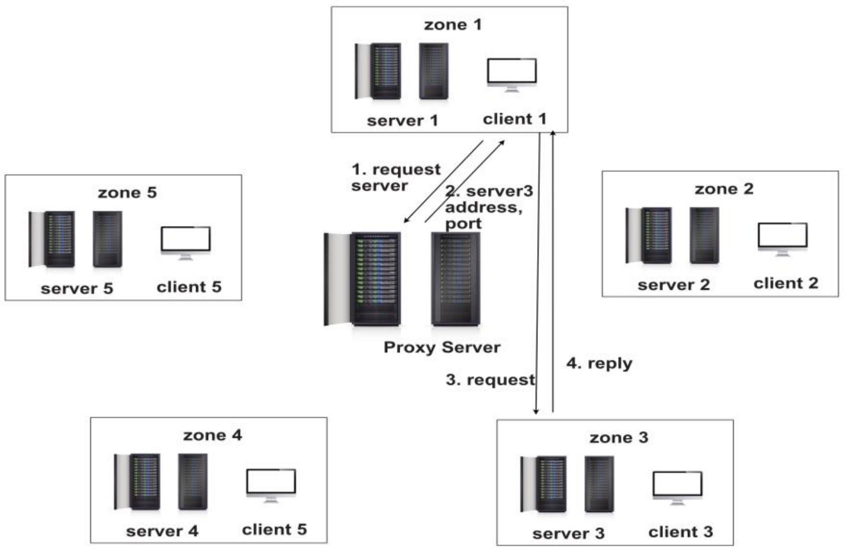

Figure 1. Proxy Server and Neighbor Zones

Figure 1 shows a client in zone 1 that asks for a connection from the proxy server. Proxy server will first check the server in the same zone and if it not overloaded (less than 18 requests in the waiting list), it will return to the client the address and port to the server in the same zone so that the client can send the request to the server and get the response (Scenario A as indexed by 3.A and 4.A) in the figure. On the other hand, if the server in the same zone is overloaded, the proxy server will check the two neighbors of that zone (Zones 2 & 3 as per our scenario and visualized in Figure 1) and return the address of nearest server that is not overloaded (or the server at same zone if both neighbor servers are also overloaded). In the visualized scenario in Figure 1 (Scenario B and indexed by 3.B & 4.B), Server in zone 3 was assigned to the client in Zone 1 since servers in both zone 1 and zone 2 are overloaded. (For server in zone 4, the next servers are server 5 and server 1)

You can decide on the design of the remote interface of the proxy server for both clients and servers on different zones. Proxy needs to provide a remote interface for clients so that clients can invoke its methods. Additionally, proxy needs access to servers' remote interface to fetch information about servers' load and waiting channel every once in a while. After every 18 requests that the proxy assigns to a specific server (18 replies to clients with the same server address), it will invoke a remote call to that server to fetch updated information about its workload and number of tasks in the waiting list. The proxy will have independent counter for each server. Hence, fetching information from each server will take place only after assigning that specific server to clients every 18 times. You also have to consider that updating the proxy with servers' workload should not interrupt servers' or proxy's execution mode and it should take place in another thread.

In real distributed systems, establishing a communication to different servers will take different amount of time. In order to simulate the connection time in this assignment, you should consider that communication to the server in the same zone takes 80 milliseconds and communication to servers in a neighbor zone will take 170 milliseconds (additional 90 milliseconds). More details on how to simulate the communication time will be presented in the next subsection.

2 Server

1. The service interface exposes the following methods that have to be implemented by a servant class
   - **a. getPopulationofCountry(countryName)**
     - i. given a country name as input, return the population of the country by summing up the population of cities in that country
     - ii. eg:- country name = "Norway" => population = 3,162,856
     - iii. eg:- country code = "Sweden" => population = 9,362,428
   - **b. getNumberofCities(countryName, min)**
     - i. given a country name and min as input, return the total number of cities in the given country that contains atleast the "min" amount of population
     - ii. eg:- country code = "Norway", min = "100,000"=> result = 4
   - **c. getNumberofCountries(citycount, minpopulation)**
     - i. Return the number of countries that contain atleast 'citycount' number of cities where each city has the population of atleast 'minpopulation' number of people.
     - ii. eg:- city count=2, min population= 5,000,000 => result = 7
   - **d. getNumberofCountries(citycount, minpopulation, maxpopulation)**
     - i. Return the number of countries that contain atleast 'citycount' number of cities where each city has population between 'minpopulation' and 'maxpopulation'.

        ii. eg:- city count=30, min population= 100,000, max population = 800,000 => result = 30

    a. The stubs of the remote objects (instance of the server class) should be instantiated and registered with unique names on different ports (anonymous ports can also be used).
    e. You can create a ServerSimulator class that will create 5 instances of the server class in different addresses and ports.
    f. A simple example will be discussed on the group meeting

2. In a distributed environment there is always communication latency. To simulate network latency, **pause the server execution for 80 milliseconds every time a remote method is invoked**. This way, you can test your code by deploying both client and server in the same machine.
    a. If a server in the neighbor zone is assigned to the client, the total communication time will increase to 170 milliseconds.
3. Each of the servers in different geographical zones maintain their own zone-specific waiting list. If the server is already in the execution mode, the new incoming task will be added to the waiting list.
    a. The servers have a FIFO policy for executing the client requests in the waiting list.
    b. Adding new incoming tasks to the waiting list should not be delayed due to server being in execution mode.
    c. You can implement different threads to isolate the waiting list and execution mode of servers. However, the server execution will take place in one single thread.
        i. In other words, each server will have 2 threads. One for executing the tasks on top of the waiting list and one to accept new tasks from the client.

3 Client

1. The client code will parse an input file containing a sequence of queries. Each line specifies a method name and an argument that should be invoked on the remote server. The input file will be provided at the course website. For each remote invocation, the client will print the result of the invocation and the time it took. The output should also be printed to a file.
2. Input file format: <method name> <arg1> <arg2> <arg3> <zone:#>
    a. The method name describes one of the 4 interfaces that server provides. Depending on the method, it can take upto 3 arguments as input. The Zone:# describes the Zone of the Server to which the request must sent to.
    b. eg:- getPopulationofCountry United States Zone:1
    c. eg:- getNumberofCities Norway 568422 Zone:4
    d. eg:- getNumberofCountries 3 1616894 Zone:5
    e. eg:- getNumberofCountries 6 1677496 4406235 Zone:4
3. Output file format : <result> <input query> (turnaround time: YY ms, execution time: ZZ ms, waiting time: TT ms, processed by Server <server#>)
    a. The result describes the result of the method invocation. The input query is the input query string (except the zone information). The third tuple describes the

turnaround time, execution time and waiting time details along with the server information. (described in the next section)

   b. eg:- : 9362428 getPopulationofCountry Sweden (turnaround time: 120 ms, execution time: 10 ms, waiting time: 100 ms, processed by Server 1)
   c. eg:- 1 getNumberofCities Norway 568422 (turnaround time: 150 ms, execution time: 10 ms, waiting time: 120 ms, processed by Server 4)
   d. eg:- 17 getNumberofCountries 3 1616894 (turnaround time: 100 ms, execution time: 10 ms, waiting time: 80 ms, processed by Server 5)
   e. eg:- 2 getNumberofCountries 6 1677496 4406235 (turnaround time: 120 ms, execution time: 10 ms, waiting time: 90 ms, processed by Server 4)

4. At the end of the output file, add 4 entries. Each entry should correspond to a type of method that server handled. Each entry should report the average turn around time, execution time and waiting time.
   a. <method name> turn around time : <X> ms, exeuction time: <Y> ms, waiting time: <Z> ms
5. Implementation tip: Define a data structure (e.g. a java class) to implement the tasks and to store all the information associated with them.
6. Note: The Client represents a simulator for set of clients. In real-world scenarios, there will be tens of thousands of clients that will send request to the servers. But we use a single Client to represent all the clients.

Summary of measurements:

- Turnaround time (from the moment a client gets to one of the servers in each zone until it receives the final response)
- Execution time (from the moment a server starts running the request until it finds the response)
- Waiting time (from the moment a server gets the request on waiting list until it starts to get executed)

Assumptions and Requirements

1 Naive Implementation

- The server parses the whole dataset every time a request is made. The server will have to read the complete dataset to answer the client's query.
- The client should name the output file as naive_server.txt. Each entry in the output file should correspond to a query from the input file

2 Server-Side Caching

- Each server maintains a cache. The cache can store upto 150 different results.
- If the information is not found on the cache, the naïve implementation should be utilized to respond to the user query. Each server will have its own zone-specific cache independent from the servers at other zones
- The server only saves and updates the cache per request it receives in real-time. Meaning that at the very beginning the server cache is empty and the servers do not perform any pre-processing on the dataset!

- Cache will be updated dynamically and the information that servers keep in cache exceed the limit, they will release the least recently used entry from their memory and replace it with the new information
- The client should name the output file as server_cache.txt. It is assumed that if the server contains the cache then the client also knows about it. The flags at the command line can be used to enable cache in the server. The flags can also be used to notify the client that server contains cache.

3 Client-Side Caching

- The Client also maintains a cache. The cache can store the results returned by the Server.The Client-Side Cache can store upto 45 results.
- If the Clide-Side Cache is enabled, then the Client should check the cache to respond to the query. If the information is not present in the cache, then it can send request to the Server.
- The Client can only save and update the cache per request it receives in real-time. Meaning that at the very beginning the cache is empty and the Client do not perform any pre-processing on the dataset!
- Cache will be updated dynamically and the information that Clients keep in cache exceed the limit, they will release the least recently used entry from their memory and replace it with the new information
- Report the same output as explained before in a file named “client_cache.txt”

Deliverables:

Via the Devilry system:

Your submission should be a compressed file containing:

- A description of your design and implementation (preferably PDF) including:
    - Screenshots of both client and server under execution.
    - a user guide for compiling and running your distributed application.
    - Description of how you divided the workload among your group members.
    - Everything that is needed to compile and run your distributed application (in a zip archive), such as:
- Any artifact/file/library that is needed to compile and run your distributed application (in a zip archive)
- Source code (should be well commented at method/algorithm granularity);
- Ready to deploy jar file;
- Three Output files containing the required measurements: naive_server.txt, server_cache.txt, client_cache.txt.
- Any other artifact that you consider relevant to compile and run your application.

You can use any sorts of available libraries to develop your system. We encourage using Maven to import libraries into your projects. However, if you chose alternative ways to import libraries, you should also document how to do so.

On the day of submission or no later than 3 working days after:

Present your assignment to the TA by explaining your design choices. You will also get asked to run the server and the client while all group members are present at the meeting.

You should email the TA to get a reserved time for your group.

Submission Deadline on devilry: 23:59 on September 22, 2023

Presentation Deadline: 17:59 on September 27, 2023