Net Computing
Programming project

Ruben Kip (S2756781)
Ana Roman (S2763753)
Teun Staghouwer (S2720175)

April 3, 2017

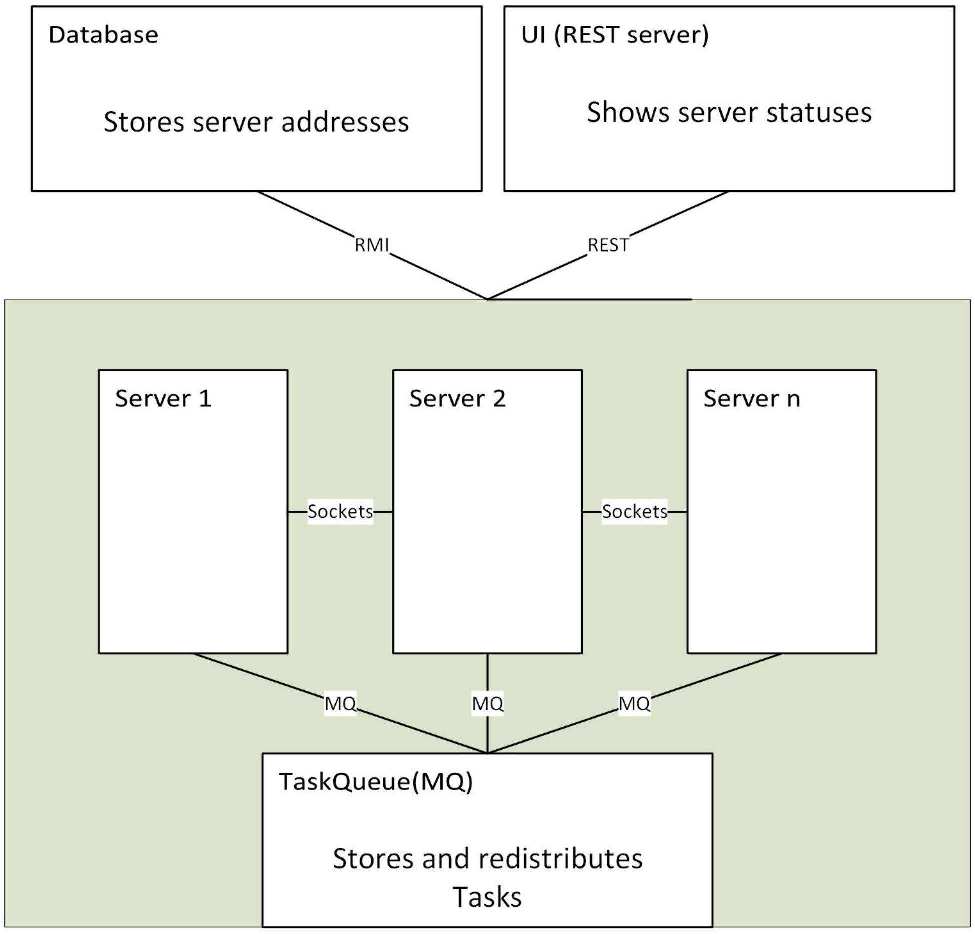
Introduction

The following document covers the final project that our group had to submit for the Net-Computing course. The students had to build an application which would implement four concepts relevant to the subject: sockets, remote method invocations, message queues and web services.

Final Project Idea, short description

The idea behind our project is to save energy by dynamically starting and stopping servers and assigning work depending on workload and available tasks. This is done by our network system. Creating a Message Queue(MQ) for storing and dividing tasks that need to be executed. We create a central database responsible for keeping track of all participating servers, this database updates all participating servers and will be updated by RMI for scalability. The servers will be communicating with each other peer to peer and turning each other on when their workload gets too heavy, while turning themselves back to idle when there is almost no workload. Finally to visualise the switching on and off of servers we use a rest server which receives put/delete request about the server state, which can then be read from the server.

Note: Starting, idling and keeping track of workload(CPU) requires low level interaction with the computer. For this assignment we assume that a library is available for this, so that we can focus on the networking part of the program. So our program doesnt actually switch starts and idles servers, it simulates the workload and prints output whenever switching a server on/off.



Explanation design and Concept use

Simulating workload:

The simulation of the hardware/workload is done by three classes, Processor, Core and Task. The hardware of the (datacenter) nodes is simulated as a Processor. Such a Processor has a number of Cores just like real life processors and a capacity of how much work each core can do per second. In normal life the capacity would relate to the clockspeed of a processor among other things. But since this isnt that important for the functioning of the application and the focus was on Net-computing instead of hardware simulation we decided to keep it simple and stick with a simple capacity.

The execution of Tasks is done in the Cores. A Core has a list of Tasks it is currently running. The sum of the workloads of these tasks is perceived as the workload of the Core. A Task has a certain average load per second, and deviation from this. This because real life applications also require different amounts of computing power each moment. A since most applications also finish after a certain time a Task has a life span or duration.

Sockets:

As mentioned in the short description the servers communicate with each other through sockets.Our group opted for the use of TCP, because for our application reliability is more important than speed. Every client will be aware of all the existing connections; this has been done through the use of a centralized database that will keep track of the links in the network. This way, we can say that our network will have a star topology. The reason behind this choice was that by using peer-to-peer in order to to dynamically add more servers would result in a lot of communication and overhead.
In the case when the workload of one computer is too high, it will connect to another computer in the network. The process of server communication using sockets works the following way:

- The server will communicate with the database server, to gain all server addresses. (see RMI for details) These server addresses consists of an InetAddress and a port number.
- For each server address known to the server we send a Message using the InetAddress and port number. The message is an custom Object, which will be serialized and sent using an outputStream, it can contain an answer or a question, in this step it will be a question.
- The contacted server receives the question which is then handled by a InputHandler thread so that the server can also retrieve other requests, eliminating the problem were a lot of servers have to wait on each other for a response.
- The Inputhandler checks if the contacted server is idle or not, then returns a Message (an answer this time) the same way we send our question. If the contacted server was not already available it will start.
- The original server which sent the question now receives the answer and will continue to look for more servers if the answer was negative, or will stop searching and wait for a few second to give the workload a change to drop. Starting the process over again if workload gets too high again.

This process only requires a few steps and needs the server to send an answer back to the client, making sockets a good solution over the other, because they offer an connection which can be used to respond our answer over and dont require a more extensive system in comparison with the rest. Lastly if for some reason a server fails, this will be no problem because it will throw a socketexception then the server can just continue to try another server.

RMI:

The server responsible for keeping track of connected servers and updating these servers, is updated by use of RMI. We have chosen for this option because if we have only one central database each individual server knows exactly where to send their request to making it more scalable and doesnt require us to start multiple RMI servers by hand. Another advantage of using RMI for this purpose is because we can send the server addresses to the database in the parameter of the method call.

To startup the RMI server/database, the required files need to be compiled and a stub needs to be created after which the Server class can be ran.

These are the steps in which RMI is involved:

- On first startup of a server, it will connect to the RMI server to supply their address and port so it can be added to the network. Here we have chosen for manually typing in your address because in some cases if trying to detect the IP automatically, a wrong IP address could be taken.
- After every cycle of checking workload (3+ sec), the server will update its list by retrieving it from the RMI server(using a get method), adding new servers if they were added to the network.

(Rabbit) MQ:

In our application Message Queues fill the role of distributor of the tasks/workload of the application/datacenter. We have made this decision because the distribution of tasks has to fulfill a number of properties, namely:

- Accessible from everywhere. This because in our design we wanted our application to be able to receive load/tasks from each and every node in the system. For this a Message Queue is ideal since it doesnt run on any node of the system and is therefore well reachable from any node of the system.
- Evenly distributed tasks. This since the aim of the application was to aim for energy efficiency in datacenters. Our approach for achieving this is by either idling as much nodes as possible when workload is low, or evenly distribute the load over the nodes where the load is high. This to reduce heat production and with that energy consumption as much as possible. For this a Message Queue is also quite handy since Message Queues by default divide the messages evenly among the connected consumers due to the underlying Round-Robin Method used for selecting the next consumer.
- Flexibility in the number of connected/available nodes. Because we want to be able to dynamically add and remove nodes (which is also quite a standard practice in modern day data centers) and have idle nodes, the data structure for the distribution should be able to deal with this in a fast and fail-safe way. Also for this property does a Message Queue lend itself very well, as it works on the basis of acknowledgements, which will prevent message from just getting lost due to an error in the connection or the removal of the node.

And as mentioned a Message Queue fulfilled all of the mentioned properties nicely.

The Message Queuing is in our application hidden within the Processor of the node. So a Node/Client can add the task to its own processor, which will then in turn put in the Message Queue, which will then again distribute these over all the connected non-idle Cores.

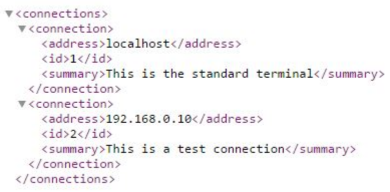
REST:

The REST concept is used as our Web UI. We host a REST server using Eclipse and a Tomcat server.

Our idea was to implement REST such that a user on the same network can connect to a remote address and see how many connections there are in our network and what addresses they have.
Whenever a server goes active the address of that server will be uploaded to the REST server list, where it will also be given a unique ID. Whenever the server goes back to idle we send a delete request, removing it from the REST server.
We chose to display the existing connections in two ways: first of all, we have a JSON representation which can be accessed at the link:

```
http://localhost:8080/jersey.connections/rest/connections
```

and it will display the connections like this:

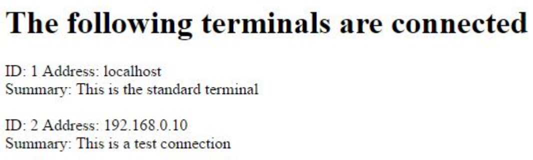


```
▼<connections>
  ▼<connection>
      <address>localhost</address>
      <id>1</id>
      <summary>This is the standard terminal</summary>
    </connection>
  ▼<connection>
      <address>192.168.0.10</address>
      <id>2</id>
      <summary>This is a test connection</summary>
    </connection>
  </connections>
```

And we also chose to display them as simple text using very simple HTML code. This representation can be accessed at:

```
http://localhost:8080/jersey.connections/rest/connections/html
```

and look like this:



The following terminals are connected

ID: 1 Address: localhost
Summary: This is the standard terminal

ID: 2 Address: 192.168.0.10
Summary: This is a test connection

The advantage of using REST for this purpose is that we can acces it from any computer on the network, also is it very easy to show different data by changing the path a little. Like zooming in on an address by taking this path:

```
http://localhost:8080/jersey.connections/rest/connections/<id>
```

Running the program

First of all, the following file must be executed in order to create the stub and to compile the classes:

```
/src/makeStub.bat
```

Then, the project should be opened with Eclipse. All the external libraries that need to be imported can be found in the folder `caches`, and they will need to be imported in the Build path of Eclipse.
In order to run the project on a single machine, two classes have to be changed: the `Server.java` and `Client.java` classes; here, the `hostname` will have to be changed to the current machine's IP address (it can also be localhost).

Then, the Main class can be ran. It will display a simple GUI. First, a server should be created, and then a client. In the Eclipse console the user will have to input his own IP address, and the system will display that only one machine is connected.

Conclusion

Our program manages to successfully integrate the four concepts taught in the course. After a lot of trial and error, we managed to make all the concepts work together and we managed to build a small application which has a lot of possibility for further extension. There would still be improvements to be made on the current prototype (for example, finding a way of correctly detecting the host's IP address without him having to input it manually if connected to Wi-Fi), but the application could be easily extended.