Technical University of Denmark

02220 - *Distributed System*

Project of Distributed System

Authors:
Ali Chegini s150939
Quentin Tresontani s151409

15th May 2016

Summary

Introduction

1. Presentation of the project
    1.1. Problem Statement
    1.2. Concept to offer free internet access via wifi
    1.3. Basic design consequences, first ideas

2. Design of the project
    2.1. System requirements
    2.2. System architecture
    2.3. Communication paradigms and protocols

3. Implementation of the project
    3.1. Objectives and simplification decisions
    3.2. Achievements
    3.3. Discussion of the prototype achievements

4. Conclusion

Appendix
    Appendix A - Glossary
    Appendix B - Bibliography
    Appendix C - Work repartition

Introduction

This paper presents a solution for providing wifi access in exchange for the use of hardware capacities of people waiting in airports. This concept has been inspired by projects such as seti@home which uses grid computing in order to create calculators based on the computer of individuals agreeing to help the project. Our project aims to improve this concept by proposing a counterpart to the individual users performing computations.

This report is based on three different parts. The first one introduces the project and gives an overview of the system based on our problem statement. A short explanation of the proposed solution and its related concepts are presented. The second part of the project discusses different aspects of the design of the project such as system requirements, system architecture and communication paradigms. In the requirements part, we aim to specify functional and nonfunctional requirements that would be necessary to design a reliable and secure system able to fulfill our objectives. We propose then accordingly a platform independent design proposition based on multi-layer architecture, on concepts related to grid computing, on communication protocols and paradigms that are discussed in detail. Finally, the third part discusses a java implementation of a prototype we have made. Even if this prototype remains unfinished and is not usable as such, it has enabled us to apply some important concepts of distributed system in theory as well as in practice. Concepts such as data transmission protocols, socket programming, network topologies, network security, multi threading and have been started to be implemented.

1. Presentation of the project

1.1. Problem statement

Our solution is proposed based on the two following problem statements we made.

It is really hard to live without internet in this modern era, especially when travelling abroad, it will be much harder to have free and unlimited access to internet. We chose to focus on airports because in our experience, airports are the worst places to find free internet access. Most of the time, you either have a free limited connection (limited time, limited data traffic or very low connection speed). In case, you want a reliable internet connection with good speed, you have to pay a fee to be able to access internet or to consume something from some bars or restaurants. Furthermore, in one hand, there are some people in need of free internet access.

This problem statement is closely linked to the fact that most users do not use their hardware capacity fully. As a consequence, we have individual users who need free internet access and have unexploited hardware capacities as a resource.

On the other hand, there are some companies that occasionally need to perform huge amount of specific computation. It will not be reasonable for them to build up and maintain a huge infrastructure because nowadays any infrastructure will get underrated soon and they need to upgrade their hardwares constantly. They need to find a way to have access to huge computation capacity for a decent price, which excludes for them paying ownership costs and maintenance costs for hardware capacities. These companies would then be interested in requesting services in order to have their computations done. We have so companies with a need for computations and money as a resource.

1.2. Concept to offer free internet access via wifi

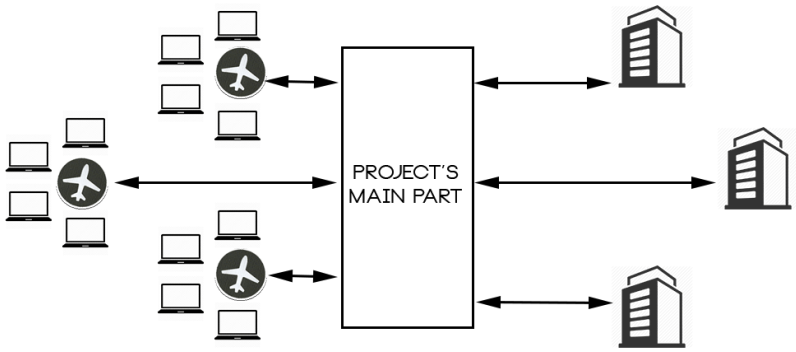
Our goal is to create a multi-sided-platform which brings together companies and individual users such as the one waiting in airports. At first, we intended to provide our service for broader range of clients, but limitations on grid computation did not allow us to do that. Instead we decided to have a narrowed target market which is consist of specific companies which require huge amount of computation to be done is short time. This way our business model is a niche market and quality of services will be guaranteed because we only have to deal with certain number of companies that are able to afford high price.

In fact, on the one hand, all these individual users have a computational capacity which they are not using and could be put in use. On the other hand, there are companies which have money that we could use to pay the internet connection for people in airports. Better choice for them would be to use the dynamic hardware capacity of the people who will upgrade their computer constantly without any cost for the company.
The idea is so to provide free internet access via wifi in the airports for people travelling and waiting there. In exchange, we would use their computers in order to perform computations

for the companies we mentioned while they are connected to wifi. The choice of airports is a strategic one since people often spend a few hours there. This idea was inspired by the computational P2P implemented in the seti@home project, but is improved by providing a counterpart to the peers.

1.3. Basic design consequences, first ideas

Our solution would imply so two different types of users - companies and individual users- we should so provide two specific interfaces for them that are matching their needs. We would also need at least two components: a main server and some subsystems located in airports as described on the following image.



2. Design of the project

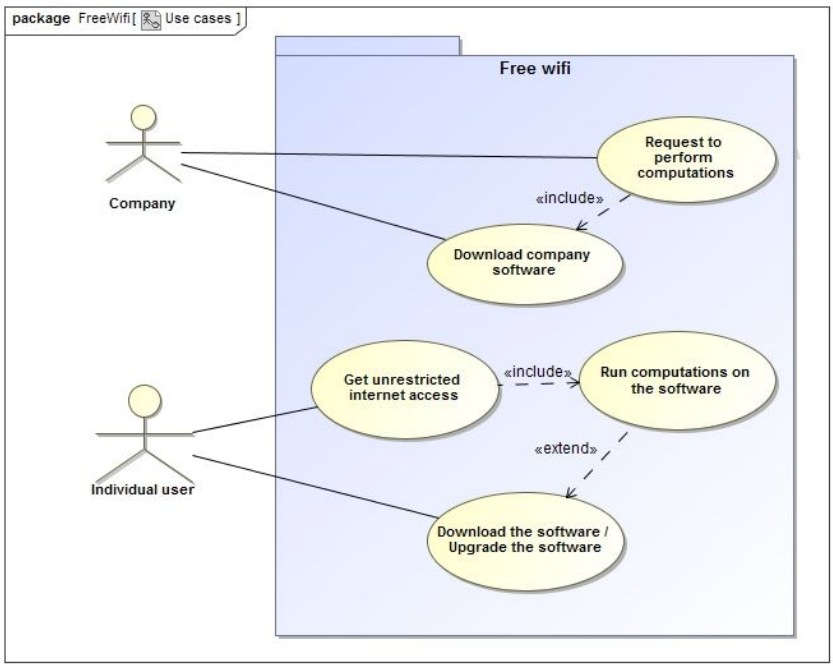
2.1. System requirements

2.1.1. Functional requirements

As a multi-sided-platform, our system focuses on bringing together two different types of users that have access to different resources. On the one hand, we have *companies* which have money as a resource and lacks of computational resources as a need. On the other hand, *individual users* who have a temporary need to connect to the internet inside airports and who owns some computation power thanks to their computers as a resource. These statements are the basis of the elicitation of the functional requirements of our project.

Without considering all side activities that may be related to companies, such as billing clients or providing customer assistance, we have only a few basic use cases. Companies should be able to request us to perform computations and should be able to get back the results of our computations. Individual users should be able to download the user software in order to access unrestricted internet by connecting to our wifi and to run our computations. Some side use cases are also needed. Users should also be able to update their user software if needed.

The softwares mentioned are due to technological constraints that will be explained later on in the design decision.



2.1.2. Non functional requirements

The following section describes and discusses the non functional requirements related to the design challenges that our system, as a distributed system, needs to face. These challenges are related to the following subjects: *Openness, Heterogeneity, Transparency, Security, Failure Handling, Scalability* and *Concurrency*.

Heterogeneity: Our system must be able to deal with many kinds of computers and operating systems on the individual user side and on the company side.

Openness*:* Our system has to face the openness challenge. In fact, the system must be able to be extended to new subsystems in new airports. It represents openness at the hardware level. The system must also be able to face openness at the software level since we have to deal with two types of software that might need to be expanded in the future.

Security*:* Our system must enable secure transactions. The system and its communication protocols must include concepts related to *authentication*: proving identities of the communicating entities, *authorization*: only identities allowed to access a message should access it, *confidentiality*: messages can be read only by the authorized ones, and *integrity*: message cannot be changed without the receiver or the sender knowing it.
Another common important concept related to security with distributed system is *availability*. It represents the protection against interference when users are trying to access resources. However, considering the type of service we provide, we don't have any critical section to consider. So availability is not an issue for us.

Scalability*:* Our system must be scalable in order to be expanded when the number of users grows. However, we don't think it would be an big issue for us. In fact, we have some control on the number of users since users can only connect to our system from airports. The number of user will only grow with the number of subsystems we are implementing. We shouldn't face a huge unexpected growth in the number of users.

Failure Handling: Our system should be able to handle most failures at all levels. Even if a failure could slow the computation process, it should not stop it completely. To achieve that, each airport system should be able to work independently from the other ones. The system should also be able to keep track of the computations, to handle the case of a user leaving the airport before pushing its results back.

Concurrency: As mentioned for the availability challenge, the system doesn't have any critical section. Concurrency is then not an issue for the system.

Transparency*:* The system would have to deal with many kinds of transparency as *location transparency:* users and company should not now where resources are located, especially they should not be able to locate the main server, *replication transparency:* since the result of our computations should appear to the companies as unique even if we have some replications of the resources shared to the individual users or *failure transparency:* since

individual users and companies should not be able to see the failures and recovering of the system, especially considering the case of a user leaving the airport before pushing its results back. Some other aspects are for us not to consider. For instance, we don't need any *concurrence transparency* since we don't have any critical sections or *performance transparency* since we are not performing most of the operations we have from client.

2.2. System architecture

2.2.1. Global overview

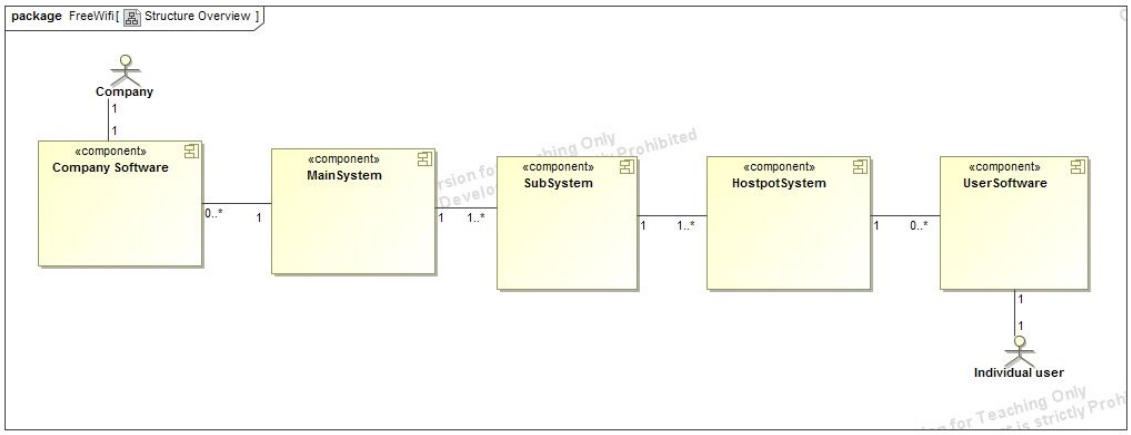
According to our problem statement, the solution we want to provide and the requirement elicitation we have made, this section focuses on answering four basic questions on distributed systems. *Which are the communicating entities? Which are the communication paradigms and protocols used by these entities? Which are the roles and responsibilities? How is the placement of these entities made?*

Communication entities?

The design we choose for our project is based on the principles used for grid computing which consists in "creating a collection of computer resources from multiple locations to reach a common goal" [1]. The point of grid computing is so to use the computers of individual users in order to perform commonly orchestrated computations. The main system coordinating the grid computations should so take care of providing these individual users with resources for the computations. Usually, to perform computations on the user computer, the user has to download a software which already contains the treatment that should be applied to the resources.

From this basic description of grid computing, we can point out two components of our system: a *main server* that should provide all individual users in airports with computations and a *user software* used in order to perform computations.
The main difference of our project with other classical grid computing projects is that we want to perform different types of computations specifically designed for each client company whereas other projects as seti@home always perform the same treatments. In order to fix this challenge, we choose that a *company software* should also be integrated in order for companies to send us their computations to perform and for us to update the user software accordingly.

According to our problem statement, we also need some systems in airports in order to delegate some of the functionality of the main server and some hotspots point that would take care of providing access to the authorized users. In order to have more flexibility, we choose to have this two subsystems separated. In one airport, there would be so only one *subsystem* in charge of coordinating resource transmission to users, and many hotspot points to provide them with an internet connection. Our system would so be composed at least of five elements, a software for the user, a software for the company, some hotspot points and subsystems in airport and a main server.



Communication paradigms?

When we started to work on the project, we were interested in creating a system that would rely on peer-to-peer characteristics. However, considering the requirements we made for our project, individual user computers (that would have been our peer nodes) and the main server have different roles they cannot switch. We considered then working with a hybrid structure. Subsystems located in airport would have then been considered as our peer node and would have been able to communicate together. However, we finally dropped this option for security reasons. This is why we used a client server based communication paradigm. On the individual user side, our system is so based on a hierarchical structure with different levels: on top is the main server, then the subsystems and at the basis are the individual users. On the company side, our system is a classical direct client server relation.

Our communication paradigm is based on interprocess communication which includes the following concepts: low-level support for communication between processes in the distributed system, including message-passing primitives, message queues and socket programming. For the relation to individual users, remote invocation paradigm was not possible since we want to have communication made on the individual user computer and not on the main server or on the subsystems.

The communication protocols applied to these paradigms will be discussed in another part of the report. It is, however, important to keep in mind while designing them that we have an asynchronous system without any time coupling. It means that our senders and receivers, especially here individual users, should not need to exist at the same time.

Roles and responsibilities?

The following section sums up the role and responsibilities of each component of the system. Referring to this table, we can easily see that components have fixed responsibilities that cannot be switched. This observation confirms our choice of a design centered around client-server relations instead of peer-to-peer relations.

Component	*Client software*
Role	Sending and getting the computations to the main server
Responsibilities	- Send computation to the main server - Get computation results from the main server - Encrypt the messages

Component	*Main Server*
Role	Coordinate the work repartition between airports
Responsibilities	- Get the computations from companies - Split the computation resources between the subsystems according to security standards - Check if the availability of the subsystems

Component	*Subsystems*
Role	Coordinate the work repartition inside their airport
Responsibilities	- Get computation resources from the main server - Split computation resources between individual users - Keep track of the work repartition between individual users - Check if users are still connected - Check the status of the hotspots of the airport

Component	*Hotspots*
Role	Enable the wireless internet connection
Responsibilities	- Allow or deny each individual user to access unrestricted internet depending on if the user software is running or not on his computer - Allow restricted internet connection enabling to download or update the user software.

Component	*User software*
Role	Run computations on the individual user computer
Responsibilities	- Get resources from the subsystem - Run computation resources on the computer - Push back computation results to the subsystem - Enable full internet access

Placement?

Based on our system description and requirements, the geographical placement of our communicating entities should at least match the following constraints. Subsystems should be located in airports, while the main server should be hosted somewhere else.

Since our system is composed of different servers, one main server and some subsystems which are located in separated host computers and interacting as necessary to provide a service to the client, our architecture is comparable to a *service provided by multiple server*.

Our system would use a proxy on the server-to-company direction. We don't need it to have any cache for clients since we are not storing any data objects for them. The purpose of having a proxy is mainly related to security reasons in order to avoid having direct communication with the main server. The proxy would help us in order to keep the main server address anonymous. This proxy implementation has to be related with the subsystem role in airports. At some extend, subsystems are also playing a role of proxy since they are intermediary which prevent the individual users from directly dealing with the main server.

2.2.2. Main server and sub systems

The purpose of the main server is, as explained before, to bring together individual users that can run simple computations and companies that need to run computations. The communication with the individual users is made through different sub-systems located in airports. These subsystems are responsible to share computations to the individual users connecting to them and to send the result of the computations back to the main server. The subsystems in airports are so comparable to different services of the main server used to perform computations. This is the reason why we chose to use a Service Oriented Architecture structure for the main server and its subsystems based on a multi tier architecture.

Multi tier architecture

In order to achieve these different tasks, we chose to use a Multi-Tiers architecture. The main server and the subsystems are so composed of three different tiers: the presentation tier, the logic tier and the data tier. All interfaces are implemented in the presentation tier. The logic tier is used here in order to split the computations and communicate their tasks to the different subsystems. The data tiers is used to store all the data including the keys used in messages.

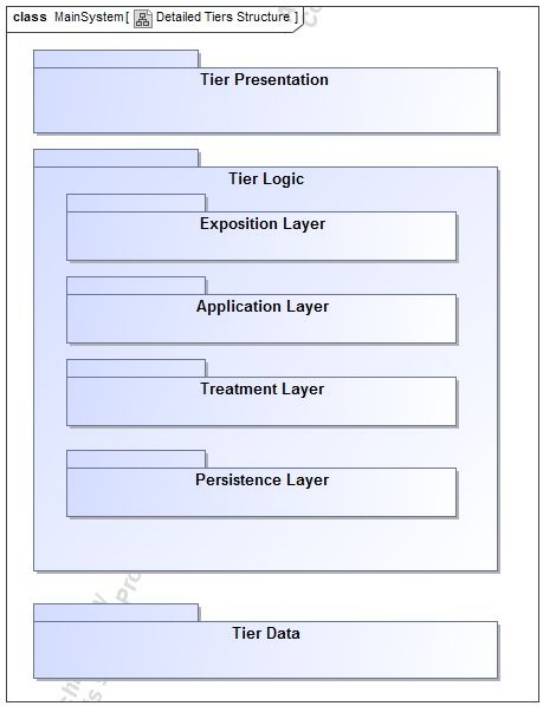
1 - Tier Presentation

The presentation tiers is centered around a Model View ViewModel architecture: view classes are the one rendered on screen, model classes are containing all information that should be presented to the user and viewModel classes are the ones assigning to the model elements their places into the view. This tier would be beneficial for us for maintenance purposes, but might also be involved in the relation to companies, depending on the type of relation we want to provide them with.

2 - Tier logic and multi layer architecture

In order to increase modularity and ease maintenance in the main server and subsystems, the logic tier is centered around four different layers. These four layers are matching the following pattern and should so fulfill the following functions:

Exposition	*Layer used to handle the communication with the interface tier.*
Application	*Layer is used for the communication between services (here, subsystems).*
Treatment	*Layer used for the business treatment (here, splitting the computations).*
Persistence	*Layer used to handle the communication with the data tier.*

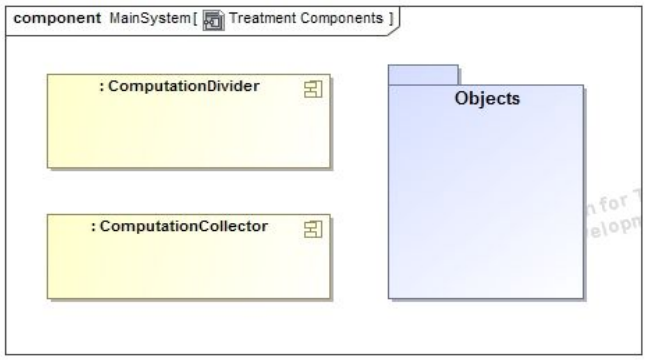


The ***exposition layer*** contains the various ViewModels used in the presentation tier. This way, we can easily change them in order to have a completely independent presentation tier.

The ***application layer*** contains the classes used to communicate with other services. This layer contains the call protocols - server and clients-, classes defining the Data Transfer Objects, the objects we send and some mapper in order to map these different objects.
In fact, in order to have a more modular system, the objects used to communicate between the subsystems and the server are extending the AbstractDTO class which stands for Abstract Data Transfer Object. These classes are only defining containers. It means that they don't have any logic and are only composed of getters and setters in order to store values of objects.

For the main system and the subsystem, the application layer is structured the same way, but it slightly different. This layer is composed of a set of clients and of server thread that can be instantiated in order to communicate with other parts of the system. The thread classes will be described in further details in the part centered around communication protocols and paradigms.

The ***treatment layer*** contains all the classes used to apply treatments on the objects. Its most important task is to split the computations between the different services when getting a new computation, but also to store the results that are sent back by the sub systems and to put them together before sending it back to the clients. Its structure would be the following one:



The component computation divider would so focus on splitting resources. It should be able to match some security conditions. For instance, it should be able to split computations in a way that even an attack on one specific sub system would not allow the hacker to get enough information on the whole project. The component computation collector should be able to put all the results together.
The treatment layer is centered around a set of objects named POJO which are the translation of DTO objects to the treatment level. The name POJO stands for Plain Old Java Object, but similar kind of objects can be found in other language. POJO have basically the same main class members as the DTO, but they also have an internal behaviour (methods). They have so an internal logic that can be used. However, POJO don't have any persistence on their own. It means that they cannot save themselves inside the database and will need the action of other classes. Saving the POJO objects inside the database is the reason of the persistence layer.

For the subsystems the treatment layer would be less expanded since it doesn't need any real computation divider and collector.

The ***persistence layer*** is used in order to save the object in the database. It is usually composed of *UnitOfWorks* and *Repositories*. *UnitOfWorks* are used as a way to keep track of all the transactions made in the database. *Repositories* are used to save objects in the database. Communication with the database is usually done by using some library or framework as Linq for C# in order to directly make SQL requests in the database inside the code.

3 - Tier data

The database would take the form of a basic SQL database allowing us to store the POJO and objects and their related informations.

2.2.3. Hotspots and user software

Structure and objectives of the hotspot

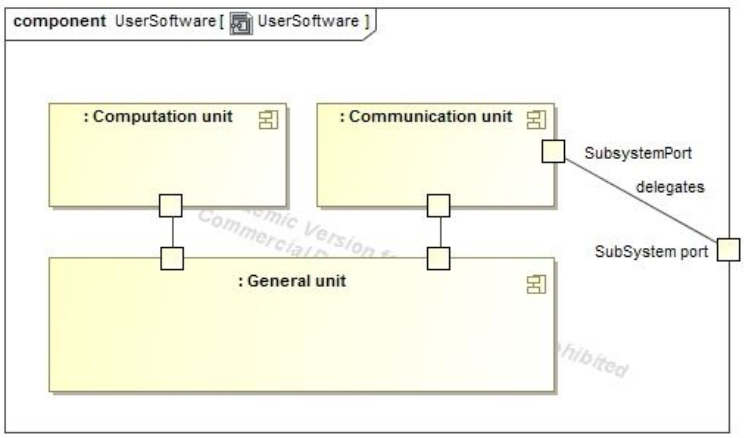
Their main purpose of the hotspot is to allow or deny a full connection to the internet for the individual users in airports. If the user is not running computations on his computer, the software should enable the user a limited connection to the internet in order to allow him to download or update his user software. If the user is running computations on his computer using our software, he should be able to access the internet without time and data amount restrictions (only some domain restrictions related to moral or ethical purposes).

The hotspot should have a list of the current versions of the software that can be used depending on the computations for companies that are not finished yet. If the user has a previous version, the hotspot should not allow the user to connect to unrestricted internet connection, should send a message to the user software in order to ask for an update on the web site.

Structure and objectives of the user software

This software is due to a constraint because of grid computation. Right now, it is hard (impossible) according to some articles to push computations. So, we had to find a way to enable changing the treatment made inside the computer. Our solution is inspired by the common one used in grid computing, but also, to some extent by mobile code placement strategy.
This can be done using two different ways. Firstly, we could force the user to do regular updates of a full software. The other solution would be to have a modular structure that automatically download new packages containing the treatment to apply on resources. This solution has been chosen since it involved less user actions and makes it more user friendly.

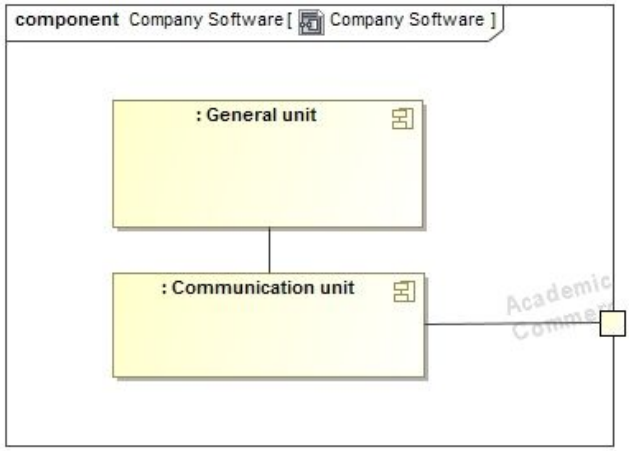
The structure of the software would be so designed around 3 components.



The first component would be the computation unit. Its role is to perform the computation for a specific company and its specific resources. This part of the software should be updated regularly by the users in order to match the current computations performed by the main server. The communication unit is the component managing the communication with the subsystems and to the wifi hotspot. The general unit would be responsible for coordinating the two other components in order to provide the computation unit with resources received from the communication unit. This general unit would also be responsible for proposing a small user interface to the user while the software is running.

2.2.4. Client software

The structure of the software would be so designed around 2 components.



The communication unit is the component managing the communication with the main server. It is responsible of requesting a dedicated port for the company and of the encryption of all messages. The general unit would also be responsible for proposing a user interface to the companies and to act as a controller on the communication unit.

2.3. Communication paradigms and protocols

Based on the requirements specified, our system should be reliable, secure, consistent, maintainable and fault tolerant. Our system should make sure of that all the parties involved in communication are safe and also what we broadcast through the channels are delivered intact, without any package loss while keeping the same order. Speed is not the main factor in our system because we are dealing with computations which can take long time to get finalized. Client companies waiting for users to generate the result is unavoidable.

Choice between TCP and UDP

In order to implement our communications we needed to choose between UDP and TCP. UDP is a connectionless protocol which means one program could send a load of packets to another program and that would be the end of relationship. UDP does not have an inherent ordering which is important for us. UDP does not guarantee reliability which means we can not be sure if the messages are delivered intact. In addition, UDP does not have option for control flow, recovery, acknowledgment and handshake which make it unsuitable for our

program. UDP is useful for systems where speed is a major concern. However, whenever reliability is needed UDP is not a good candidate.
The most important characteristic of TCP is that it is a connection oriented protocol and establishes viable connectivity, which is really important for this project. Delivery is guaranteed in TCP and messages will remain intact while keeping the same order. Our applications need high reliability and speed is less critical for us.

Based on the characteristics we mentioned above, TCP has lot of advantages for us compared to UDP. TCP has been, therefore, chosen over UDP in order to implement our communications. Furthermore by rolling out UDP we avoided unnecessary complexity in the code implementation since TCP take care of the ordering by itself and is reliable. Based on the requirements of our project UDP shouldn't be used in any of our communications. The points discussed in this section are happening inside the application layer of the logical tier which was discussed before.

Other concepts discussed for the project

The following section describes concepts related to distributed systems we have been considering while thinking on the communication protocols. Some of them should be integrated, some others don't need to be.

Integrated concepts:

Logical clocks: Since our system is a distributed system, it does not have a physically synchronous global clock. In order to capture chronological and causal relationships in our system we need to use a mechanism which facilitates global ordering on events from different processes. Mainly logical time assigns a number to each event based on its logical ordering. Consequence of this assignment is that later events will have higher numbers than earlier.

Three-way handshake protocol: Since all the communications in our system are bi-directional, three- way handshake protocol is a concept we use by choosing TCP.

Sockets and Ports: In all of the channels we use specific port and socket to establish a connection between two endpoints.

Not integrated concepts:

RMI and RPC: As explained previously, our project cannot use neither RMI (Remote Method Invocation) nor RPC (Remote Procedure Call) since it does the computations on the server side which is the opposite of our objective. In remote invocation the user call the server to run a process inside the server whereas we would like to run the process on the user computer.

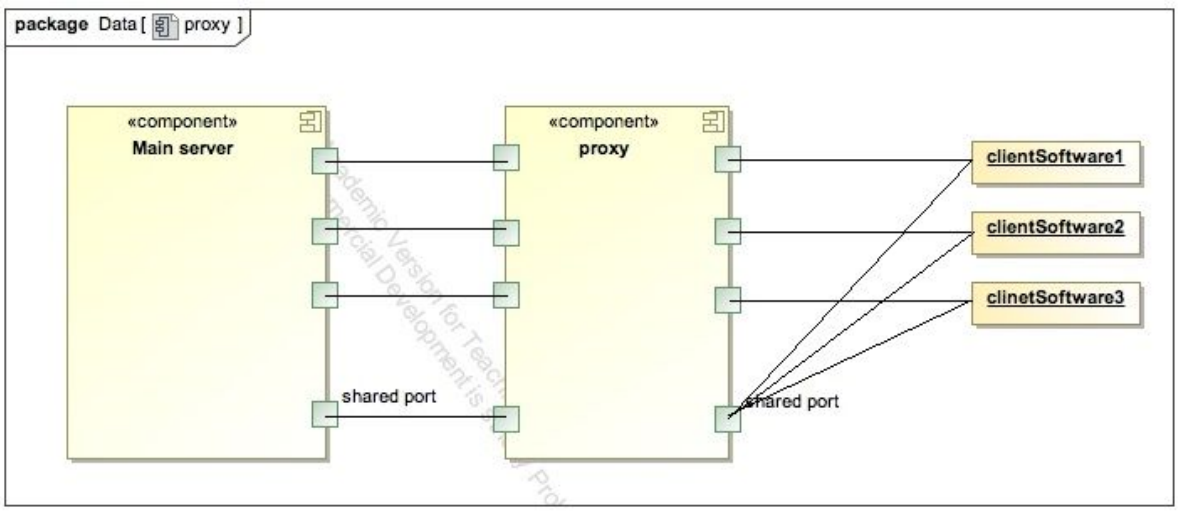
Mutual Exclusion: We do not have any critical section since different users in our program are accessing totally different packages. Mutual Exclusion is not applicable in our project.

2.3.1 Communication between companies and the main server

In this section three main protocols which are used in this part of communication are discussed including TCP, FTPS and unicast.
We provide each client with a version of software that is responsible for task such as sending out a poll request through a shared port, establishing reliable connection between main server and clients via a dedicated port, and taking care of encryption process. After receiving a poll request through the shared port which all the clients have access to. In case of successful request we open up and dedicate a specific port to that client for all the further communication. The main system will get computations from client via its software. This communication between them makes use of TCP. It should be mentioned that we could also provide web services instead providing a software, but we thought reliability and security using a software would be better. It allows us to avoid problem with exchanging keys for encryptions and would be good for us, in order to fit with the diversity of computation we would have to deal with.

Another protocol we make use of in this part is FTPS which is an extension to the commonly used File Transfer Protocol that accommodates the Secure Sockets Layer known as SSL. FTPS is the best candidate to use in this part because we are dealing with files which needs to be transferred safely in absence of web services and HTTPS.
Unicast is also used in this part of communication because one to one reliable connection is needed.

We use a proxy for this channel in order to make the communication safe. This way if one of these clients has security breach, our main server will not be compromised since we are hiding our real IP address.



Each client will have a dedicated port to send and receive data and we will use different keys for encryption through this channel. Only us and the client will have that key. Encryption through this channel will be end to end, it means whatever is transmitted back and forth will not be usable unless someone has the key. Encryption and decryption will happen in our

main server and clients side. This way even if a third party can intercept the messages he will not be able to get useful data. Each client will have a specific version of software which has a first key to facilitate encryption process.

2.3.2 Communication between the main server and the sub systems

We have different type of communication processes in this part such as ping pong, DTO and encryption which will be discussed in detail.

Ping pong port - checking subsystem availability

Server is constantly sending ping pong messages to check if subsystems are alive and updating their status based on the availability. In case any of the subsystems does not respond we consider that he is down due to any kind of failure. Since all the subsystems are working independently, failure of one will not cause any corruption for the global system.

All of our subsystems and packages have a unique ID so that, in case of any failure, we can exactly know which subsystem has which package and we will be able to resend lost packages to another subsystem.

DTO Port - Providing subsystem with resources

Main system split the computations and send repartitioned work to subsystem via TCP. As we discussed before, by using TCP, we can be sure about delivery and also having a well established connection. Subsystems constantly sending poll request to get new data from the main server. Accordingly main server responses to every poll request and feed all the subsystems.

We also use encryption between main server and subsystems in order to make sure if someone is intercepting communication he won't be able to read or use the information without having the encryption key. We only use encryption for important messages like computations and result, mainly what we called DTO before. We don't use encryption when sending and receiving ping pong messages. In this part is much easier to exchanges the keys because we have them.

2.3.3 Communication between subsystems and individual users

We provide a version of the user software for each user which will be responsible for sending out poll request to get computation and establish a reliable connection with subsystem. Individual users connect to subsystem via wifi hotspots by using the software and request for computations by sending poll request, again all the data will be transferred back and forth via TCP for the reasons we have discussed above. We do not use encryption in this part since data packages are already shuffled before.

The broadcast type in this channel is unicast since we are using TCP and we want to make sure user receives a particular message. Unicast is a one-to-one connection between two endpoints. Reason to chose unicast over multicast corresponds to our design choice to pick TCP.

We use ping pong messages to make sure about liveliness of each individual user. In this part it is the subsystem which will be responsible to check and also update the user status.the software is implemented in a way that automatically pushes back result of computations after every 4 ping pong iteration. This way we make sure if a user leaves the airport suddenly we still get some result back. We give each user a unique ID and also identify data packages with a unique ID. . If we do not receive a result for a particular package from user we know which package is missing and we can resend the same package ID to different users.

3. Implementation of the project

3.1. Objectives and simplification decisions

As part of a project of the course of distributed systems, the implementation part focuses on the communication protocols used between the system components. Our objective was so to implement the basic architecture of the communicating entities and to be able to make them communicate together, at first separately and then commonly in order to be able to perform one computation requested by a client software until the end.

In order to get to that results, we took the following simplification decisions concerning the design previously described:

- We won't implement any interfaces and will just use console printing. Our presentation tier will so be kind of empty.
- We won't use any real database. We chose to only deal with lists in the persistence layer and using the library *collection* (similar to Linq for C# that you can use with a real SQL database). Our database tier would so also be empty.
- We won't deal with more than exactly two subsystems.
- We will perform only one specific treatment that is checking for prime numbers.
- We won't consider having any real hotspot point and wifi portal. Thus, the part with the wifi checking if the software is really running won't be handled in the implementation.

3.2. Achievements

The whole project implementation can be found here,
on github, on the following repository:
https://github.com/Qww57/DistributedSystems

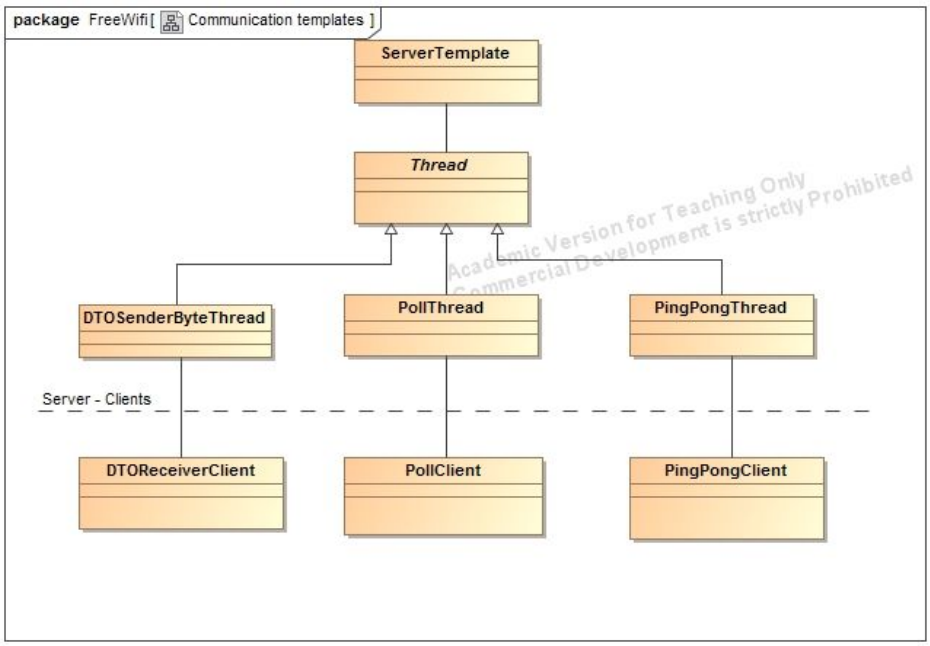
This repository is containing four different projects and one folder (*ComProtocols*) that has been used as a sandbox to get in touch with communication protocols.

- *MainServer* is the project containing the main server,
- *SubSystem* is the project containing one subsystem,
- *UserSoft* is the project containing one user software,
- *CompanySoft* is the project containing one company software.

3.3.1. Implementation of communication protocols

We have been working at first on communication protocols inside the folder *ComProtocols*. Afterwards, the communication protocols should be implemented inside the application layer of the main server and the subsystem and inside the communication unit of both software.

In order to have something that would be easier to maintain, we created specific classes for clients and server in order to use them as objects. We created notably a server template class which takes a thread as input. This way we can have the following structure for the servers and clients we are using inside the project.



References in the project:

- MainServer/Logic/Application/Sockets/ServerTemplate.java

Since, the TCP protocols automatically takes care of making the communication reliable. We have not been implementing any acknowledgement protocols in our server threads and in our clients. We have been implemented three threads and their related client.

DTOSenderThread and DTOReceiverClient

These two classes are used in order to distribute the computations. The client sends a poll request to the server. In its related server thread, the server sends then to the client the next set of resources from the waiting queue of computation resources. In the project implementation, the client will be implemented inside the software user and the subsystems. The thread will be implemented inside the main server and the subsystems.

References in the project:

- MainServer/Logic/Application/Sockets/Threads/DTOSenderObjectThread.java
- SubSystem/Logic/Application/Sockets/ObjectClientTemplate.java

Remark: While working on this thread we spent some time trying to find a way to send method from the server side to the client. We thought, at first, it was only a programming purpose. The main challenge was because the client does not know the type of the class you are sending. We tried many ways in order to achieve it using interfaces, byte conversion of the classes. However, we have not been able to implement it, we were getting exceptions on the client side such as unknown class. We found out after a few researches, as explained previously in the part on grid computing that we might not be possible to achieve it.

PollPortThread and PollPortClient

These two classes are used in order to start the communication between the client company and the main server. The client sends a Poll request to the server. The server is then checking its available ports and respond to the poll request by giving the port number to the client. The poll client will so be inside the company software whereas the poll thread will be inside the main server.

References in the project:

- ComProtocols/PollProtocol/PollPortServer.java
- ComProtocols/PollProtocol/PollPortClient.java

PingPongThread and PingPongClient

These two classes are used in order to check the availability of the clients by the thread. The thread is used by the main server in order to check the availability of its clients located inside the subsystems and by the subsystems in order to check the availability of its clients located inside the user software that have been connected previously.

In order to check their availability, the thread is sending messages every X seconds to the clients. If the clients are getting the message, they are responding to the server. This way, considering the answers of the clients, the server is able to know if its clients are still connected or not.

References in the project:

- ComProtocols/PingPong/PingPongServer.java
- ComProtocols/PingPong/PingPongClient.java

3.3.2. Implementation of the components

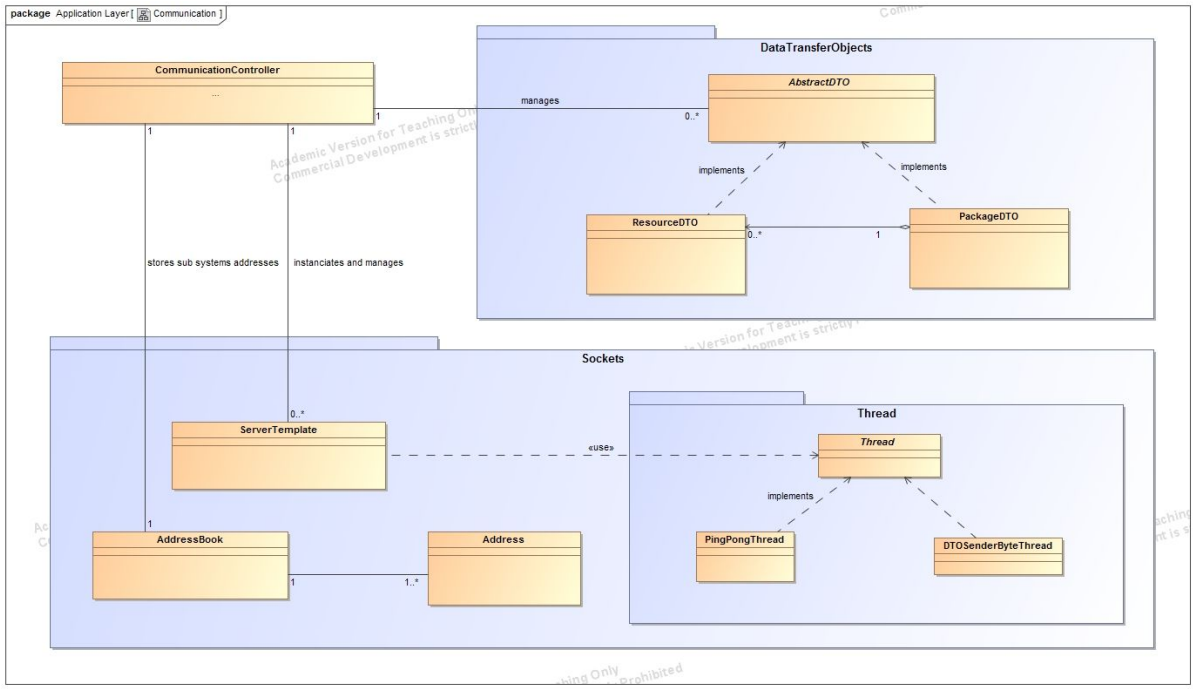
This part will discuss the implementation we have done for each of the component. It will discuss in more details the main server since this part is the central part of our project and other parts as the subsystems are made on the same model. Other ones will be discussed in a shorter way depending on the progress we made.

Main server

We have been able to implement most of the structure of the main server. As explained in our assumptions, we had to focus on the treatment tier of the server.

Application layer:

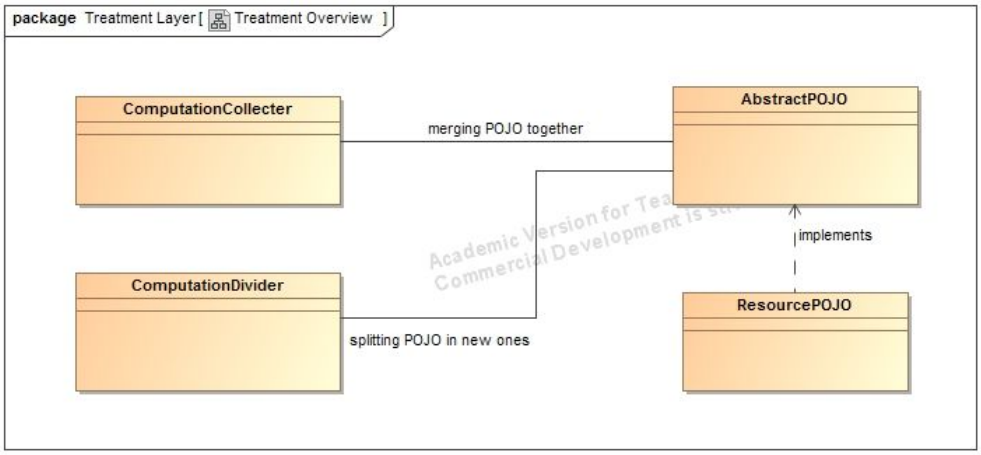
The application layer of the main system contains all classes used for communication and the Data Transfer Objects used to exchange and communicate between the main server and the subsystems. In this layer we can so find some of the protocols described previously. For instance, it uses two threads, in order to send DTO objects and in order to check the availability of the subsystems using ping pong messages.



The main class of the application layer is the communication controller. This class is used to store information about subsystems and to manage the waiting queue for the packages that should be sent. The waiting queue of this class is used in order to select the next DTO packages that should be send by the DTO server. This class contains also a hash map connecting the IDs of all subsystems and their current status according to their specific PingPongThread -there is one ping pong thread per subsystem.
In order, to deal with the addresses and ports of each of the subsystems, two specific classes have been created. *Address* contains the host location name, the ping-pong port and the dto receiving port for one subsystems. All addresses are then stored inside the *AddressBook* class.
This layer contains also two specific classes used in order to map objects coming from the treatment layer (POJO) or from the client interface (ClientDataRequest) to the DTO type.

Treatment tier:

The treatment tier is based on POJO objects and contains two main classes responsible of splitting and putting togther the computation resources and results.



The computation divider has been implemented. It allows the user to split the resources of computations for prime number checking according to some parameters. In fact, there are three different factors involved when we want to describe packages:

- the size of one element, for instance 1 to 1000.
- the number of elements we want in one package, for instance 10.
- the number of packages we want to create.

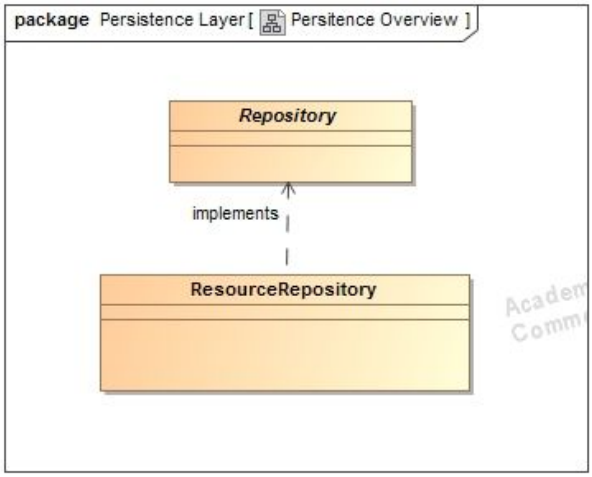
Based on the wanted number of packages and wanted number of elements per package, the constructor will compute the size of one element and create the packages accordingly.

During the process, this class is also responsible of using the repository classes in order to store the new created POJO objects in the database. This class is also responsible of creating DTO packages for the created POJO resources and to put them in the waiting queue of the communication controller.
The behaviour of this class has been tested in the unit tests:

The class computation collector which has not been implemented yet would have been responsible to do a similar treatment by updating the status of the computations which have been done inside the database, to bring these results together and to create the *ClientDataResults* element to send to the client.

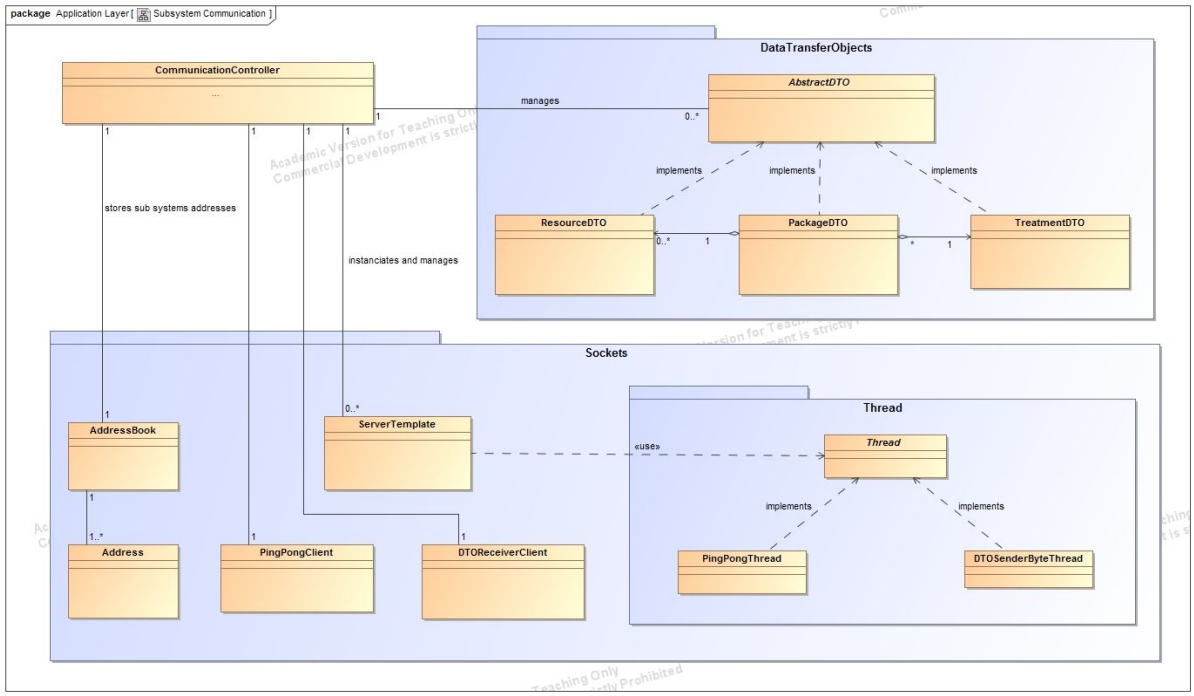
Persistence tier:

Our persistence tier is composed of a resource repository based on the library *coollection* which allows us to make requests with a syntax similar to the SQL one inside lists.



Subsystem

The implementation of the subsystem has been started in order to test some aspects as the sending and receiving of DTO objects. However, we have not been able to implement all of it, since we needed to wait to finish the main server first, if possible. As described in the following class diagram, the subsystem of our prototype would have a similar class diagrams for the application layer. There would be ping-pong clients in order to respond to the main server, ping-pong server in order to check the availability of users, DTO clients in order to receive computations from the main server and DTO servers to send resources to the users. Everything should have been coordinated by some controller classes as the communication controller that we have not been implemented.



Since the subsystems only deal with sharing computations, but does not do any splitting treatment, no treatment layer is needed. Only a persistence layer should be implemented in order to remember which user was assigned with which packages. This way, if one user

leaves the airport, the communication controller would be aware of it thanks to the ping pong servers and would be able to reassign the user's computations to other ones.

Client and user software

We have not been implementing this part.

Client and company software

We have not been implementing this part.

3.3 Discussion of the prototype achievements

We have not been able to fully implement our prototype, even with the simplification decisions we added previously. We have been focusing on the main server and the related subsystems and on their application and treatment layers.
Even if, the system is not able to fulfill its objectives right now, we have been able to implement:

- The full structure of the main server and subsystem
- Some of the communication protocols
- Create object oriented client and servers that can instantiate and use easily
- Some part of the logic on the example of prime number checking
- Unit tests on the implemented parts

We have also been able to validate or invalidate not some of our design decisions. For instance, we have been able to see the limitation of grid computing inspired systems when you want to push treatment from main server to individual users. It has helped us to face this problem in a concrete example and has validated our decision of having a static treatment inside the software and not a dynamic one pushed by server.

We are not fully satisfied with the current status of our project, since we have not been able to fully implement what we expected at first. We intended to have a small prototype -a proof of concept- running, but we are still happy with the knowledge and competences we got from that implementation.

If we had to keep working on this project, tasks for us would then be the following ones:

- Finish the implementation of our proof of concept according to our simplification decisions, which includes creating the user and company softwares and finish all the code implementation commented as TODO or FIXME
- Create user interfaces for all the components in order to have a thing that would be easier to maintain and to deal with,
- Work on a real algorithm that could split computations for different kinds of computations and not only for prime numbers as we have it here,
- Add all things related to non functional requirements as encryption,
- Do some real life experiment and not only unit tests.

4. Conclusion

Our objective was to design a multi sided platform improving the concept of grid computing by proposing a free wifi connection as counterpart to the individual users helping to run the project.

We started by specifying all functional and nonfunctional requirements that our system should be able to fulfil. We have been discussing the main challenges our system should be able to take up in order to be reliable, viable and secure. Heterogeneity, openness, security and failure handling have been described as major concerns for the project. According to these requirements, we have proposed a platform independent design based on multi tier architecture that integrates communication protocols based on the client server paradigm and matching both our needs and requirements. Usage of local databases and of replicated data has also been a central part in our design decision to make our system fault tolerant.
We have tried then to validate our decisions by implementing a prototype that could be used as a proof of concept. This prototype was not supposed to contain all features and functionality as described in the design part, but to implement the most important aspects such as distributed interprocess communication between the different entities of the system. Our main focus was to get some real hands-on experience and to get a grasp of how the different parts of system would interact and fit together on in a java implementation. Even if we have not been able to fully implement our proof of concept, we have tried to cover most of the major concepts related to communications in distributed systems. This part has been time-consuming, but also really interesting and challenging. We have been able to improve our understanding and our knowledge on how to apply theories in practice.

Some work remains in front of us before being able to fully validate our design based on our proof of concept, but we are pretty confident that this project and our vision could lead us to a reliable and viable business model. We hope to be able to carry on in a near future on this project and to get feedback and improvement suggestions.

Appendices

Appendix A - Glossary

- **Asynchronous System:** System which do not have a global clock or consistent clock rate, and each computer processes independently.
- **Companies**: (Also named clients) are companies that want to use our services.
- **Company software:** Specific version of software designed for clients which is responsible for establishing connection, taking care of encryption and other necessary task.
- **Hotspot**: Physical locations at airport where users may obtain internet access by using Wi-Fi
- **Individual User**: People at airport who needs wifi and are willing to share their hardware capacity for it
- **FTPS**: File Transfer Protocol Secure
- **Grid Computing**: Collection of computer resources located in several locations attempt to reach common goals
- **Main System**: Our main server which handles important tasks like verifying companies request, gathering data which needs to be computed, dividing the computations and sending them to subsystems
- **User software:** Specific version of software designed for individual users in airport, which is responsible for establishing connection, getting data from subsystems, and taking care of other necessary tasks.
- **Service oriented architecture**
- **RMI**: Remote Invocation Method
- **RPC**: Remote Invocation Procedure
- **Unicast**: one to one connection between two endpoints
- **SSL**: Secure Sockets Layer
- **Subsystem**: each airport has a subsystem which handles all the individual users in that airport. Subsystem is also responsible for sharing computations to users and sending back the results to Main System.
- **Multi-sided-platform:** Organization which creates value by facilitating interactions between two or more parties
- **TCP**: Transmission Control Protocol is a standard that define how to establish and maintain network communication among applications which exchange data.

- **UDP**: User Datagram Protocol or Universal Datagram Protocol is an alternative communication protocol to TCP mainly used for establishing low latency and loss tolerating connections.

Appendix B - Bibliography

[1] Grid computing, Wikipedia, https://en.wikipedia.org/wiki/Grid_computing

[2] Service oriented architecture, Wikipedia, https://en.wikipedia.org/wiki/Service-oriented_architecture

[3] Mutltitier architecture, Wikipedia, https://en.wikipedia.org/wiki/Multitier_architecture

[4] *Scheduling Algorithms for Grid Computing: State of the Art and Open Problems*, Fangpeng Dong and Selim G. Akl, Queen's University Kingston, Ontario, 2006, http://ftp.qucis.queensu.ca/TechReports/Reports/2006-504.pdf

[5] A do-it-yourself framework for grid computing, JavaWorld, http://www.javaworld.com/article/2073356/soa/a-do-it-yourself-framework-for-grid-computing.html?page=1

[6] Harold, Elliotte Rusty. *Java Network Programming*. Beijing: O' Reilly, 2005. Print.

[7] Minoli, Daniel. *A Networking Approach To Grid Computing*. Hoboken, N.J.: Wiley, 2005. Print.

[8] Wu, Jie. *Distributed System Design*. Boca Raton, FL: CRC Press, 1999. Print.

Appendix C - Work repartition

This report has been made commonly based on weekly meetings every Thursdays, additional meetings during the week ends and distributed work on our own. All design decisions have been discussed together in order to get to an agreement and to be validated by both members before being added to the design decisions.

The following section describes the repartition that has been made for the writing part of the report:

Introduction - Ali
1. Presentation of the project - Ali
2. Design of the project - Ali & Quentin
    2.1. System requirements - Quentin
    2.2. System architecture - Quentin
    2.3. Communication paradigms and protocols - Ali
3. Implementation of the project - Quentin
    3.1. Objectives and simplification decisions - Quentin
    3.2. Achievements - Quentin
    3.3 Discussion of the prototype achievements - Quentin
4. Conclusion - Ali

Appendix
    Appendix A - Glossary - Ali
    Appendix B - Bibliography - Ali & Quentin
    Appendix C - Work repartition - Quentin