Corso di Programmazione ad Oggetti - a.a. 2013/2014
Corso di laurea in Ingegneria e Scienze Informatiche - Università di Bologna

Documentation about the Engineering of CINA: a public/private multi-client chat.

Belli Stefano, Francesco Cozzolino

1 Problem Analysis

Due to the increased role of privacy in the everyday life and the request of non-traceability in the real world, users needings come to the creation of a multi-client chat that could guarantee a crypted and anonymous conversation between two (or more) clients.

More specifically, the above noted chat would guarantee:
- A public multi-clients chat
- An encrypted private chat
- The possibility to send or receive files.
- The ability to know which users are connected to the chat
- The confidentiality of every user identity.

From the WebServer side, the chat environment would guarantee:
- The ability to close connection of a client remotely
- An asynchronous handling of every event in the chat
- The management of user interaction with other users

Obviously, the integrity of every user identity is a key part of this application.
In fact, every interaction client to client should be handled with care, and with the agreement of every user involved.

From this application profile, it's clear that a clean, asynchronous GUI it's needed to get the chat to work.
Also, a private chat framework based in algorithm of some type will be needed.
Last but not least, a public chat management will be needed, so a Web Application of some type will gets the work done.

2 Architectural design

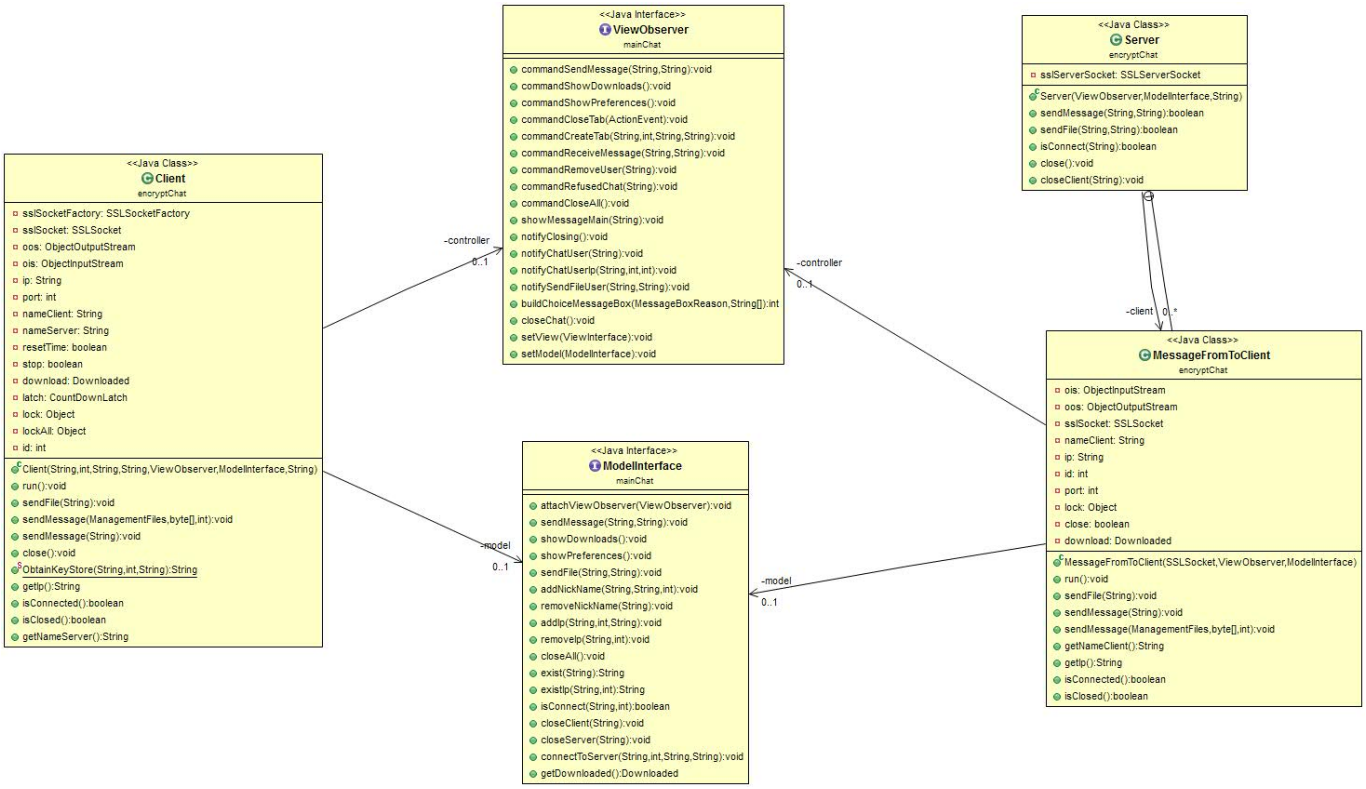
From the architectural side, the application it has been build with the MVC pattern.
More precisely, only the chat Jframes (public or private it makes no difference) are been implemented with the MVC pattern, but that's because the application heavily relies on those frames: in fact, roughly the 80% of the application lifecycle relies beside the MVC pattern.
Anyway, some others minor jframes (like Credits, or Downloaded) are not based on MVC pattern because they are relatively easy to manage.

Almost all commands that involve the chat's GUI are completely asynchronous, in order to ensure the reactivity of the GUI itself and to avoid some boring idle times for the user.

In the next pages, some UML diagrams of the project will be shown: please be aware that some methods are not called directly from the code itself, but are automatically invoked from the websocket threads.
More info about this management it's located below, on section *5.1*.

The Controller class is the core of the application; for instance, when any kind of event in the GUI it's requested (or in the data structure too) , the corespective method in the Controller class it's invoked. Those request can come from the user itself (with his actions) or by other classes such as the WebSocketHandler.

In the next UML is shown the basic interactions between the GUI and the classes for private chat:



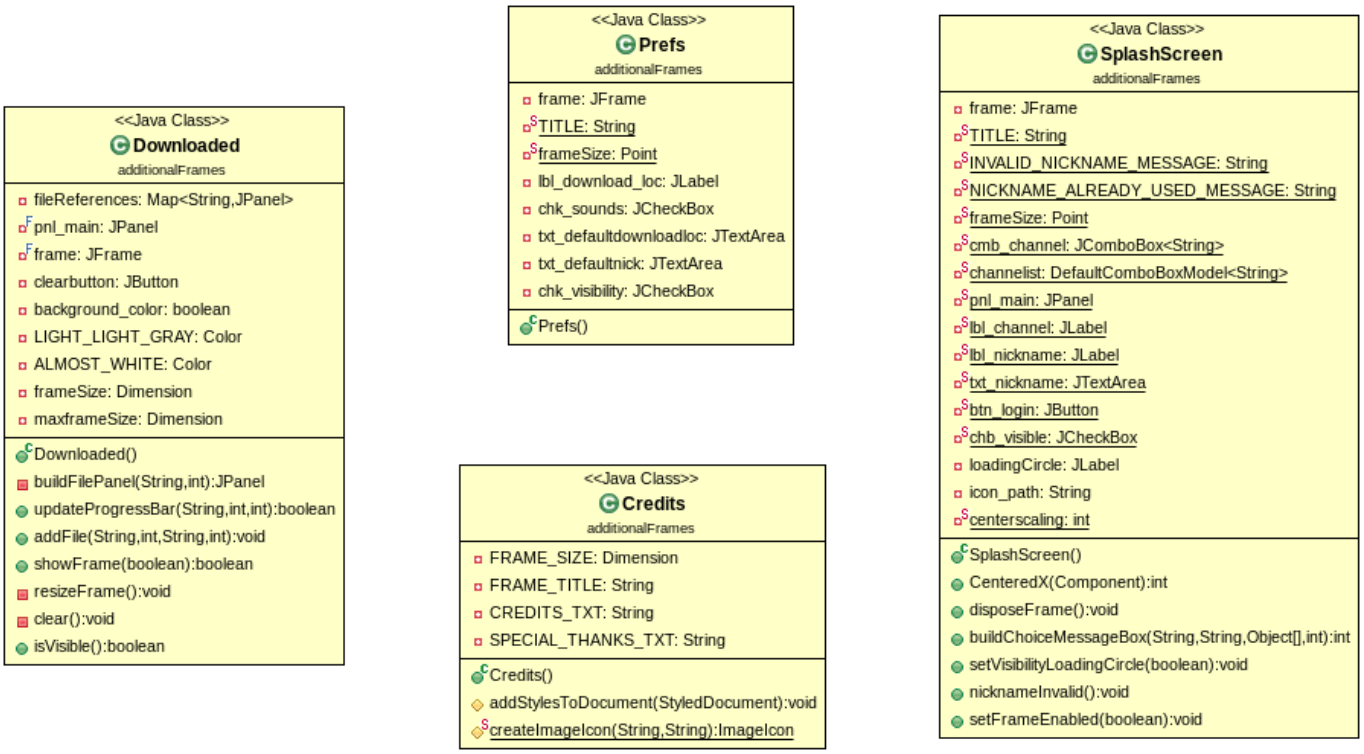
KeyStoreServer : this class instantiate a server socket that's able to exchange the user's keystore with other users who would like to have a private chat.

SendReceiveFile : This abstract class provides a schema for send and receive files and defines two abstract methods to send a message.

Server and Client : The main purpose of Server class it's to implement a SSLServerSocket.The Client class,instead, implements a SSLSocket.
Those classes allows to have an encrypted (SSL protocol) private chat between two users. They both extend the SendReceiveFile class.

ManagementFiles : Permits to set some informations about a file and store these information with an implementation of Serializable interface.

ModelInterface and Model : Interface and Implementation of data structure used from the application



Downloaded: Shows the “Downloaded” frame and handles all the incoming (and outgoing) files for the client. Be advised: an integration with the model class it’s needed, but it’s not shown in this diagram for semplicity.

Credits: Shows the “Credits” frame centered on the screen. A JTextPane it has been used as main view component.

Prefs: Shows a preferences frame on screen. This frame it’s needed due to some basic needings of personalization the user.

SplashScreen: One of the core classes of this application. Shows an initial login screen where user can select some basic options, such as his own nickname and his visibility. It is heavily dependent to the Application class.

3 Package organization

The application has been divided into the following packages :

mainChat : contains all the source codes that encapsulates the GUI and all commands related to it

encryptChat : contains all source codes that encapsulates the private chat between two users.

webSocket : contains all source codes that encapsulates all the comunication algorithms with the webserver.

preferences : contains all source code that encapsulates the information of a user. Those information are stored via the config.conf file or the Java Preferences.

additionalFrames : contains all source code related to some minor frames. Those includes the Credits class, Preferences and Downloaded.

4 Subdivision of the Application

The project it has been divided to two distinct parts: the public chat part and the private one.

All the interaction with the public (websocket) part has been implemented by Stefano Belli.
The websocket management involved all the types of messages exchanged with the server, and every type of handshake (like the INITIALIZE one).
Of course, the deployment of the server it’s also one of his works.
Also, Stefano Belli implemented the whole Preferences, Credits and SplashScreen classes.

The private chat has been implemented by Francesco Cozzolino, also has implemented the exchange of keystores between the users (necessary for chat with SSL protocol),the automatic creation of keystore (window and linux O.S.), and the management of the private chat request avoiding the use of web server.

For reason of integration of the code, some parts have been developed together as the classes model,view,controller and the management of some messages from the web server

5.1 Detailed Engineering: Stefano Belli's work

As mentioned above in the section No. 2, Stefano Belli's work relies heavily on the public chat part.
One of the main class (from the client perspective) of this management is surely the WebSocketHandler class.
The main scope of this class is to handle (as the name suggest) all the communications via the websocket protocol with the webserver.
For a better understanding of this class, here's shown some methods of this class with a brief explanation (please notice: some of those methods indentations are in POJO format, as the websocket standard for java requires)

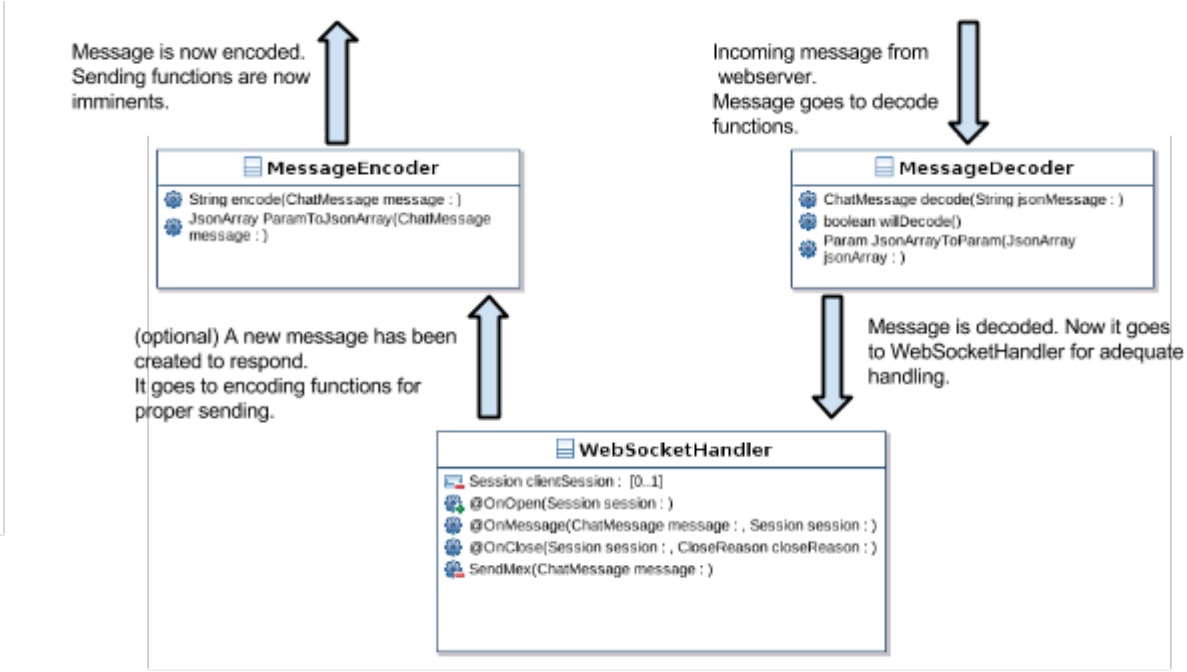
@OnOpen: this method handles the initial handshake with the server.
For example, here the availability of the nickname in the channel is checked.

@OnMessage: this method handles all the (decoded) messages incoming from the webserver.
The messages are decoded in a ChatMessage instance by the MessageDecoder class.

@OnClose: this method is mostly occurred when the connection is closed remotely by the server.

SendMex(ChatMessage mex) : this method it's used to actually send messages to the server.
In the sending sequence, an istance of MessageEncoder it's automatically called.

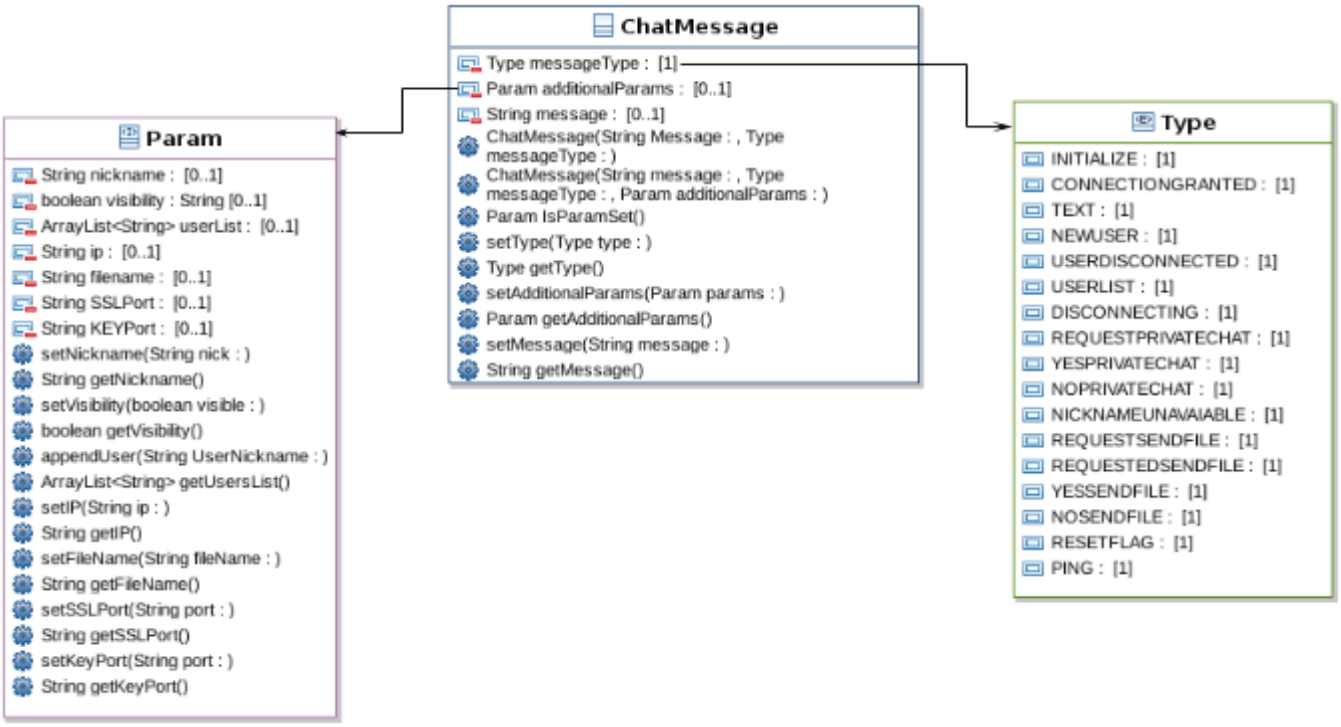
Here's a simplified schema that describes the relationships between those 3 classes(MessageDecoder,



WebSocketHandler and MessageEncoder) on a request of private chat :

Please note the webserver has an identical processing algorithm for every message sent or received.
The WebSocketHandler class is based on the Singleton pattern.
That's because it represents the connection with the webserver, which is unique. A singleton pattern is perfect for those needings, because from every section of the project the websockethandler class could be retrieved with that single, unique istance.

The actual datas sent over the network are encoded in JSON strings.
Once they are received, they are decoded in a "ChatMessage" object, which enables to have a better abstraction and handling of every circumstances.
Here's an UML diagram of this essential class:



Why json format?
JSON encoding has been chosen due to its flexibility and because it's pretty familiar with Stefano Belli's knowlnedge .
In addition, some required fields (like the users list in USERLIST chatmessage type) which involved the transmission of array-type variables can be actually easily stored in a JSONArray instance, which simplifies the transmission itselfs.
Last but not least, JSON it's a standard. It's always a good point, isn't it?

Why the above UML diagrams have missing methods/parameters?

The above uml diagrams were made before the actual implementation of this project (except for the "Type" enum).
In fact, they are missing many methods and parameters in relation with their "real" counterpart because they were a simple prototype of those classes.

Why you have used monitor(s) ?

Well, that's the tricky part.
When an istance of websockethandler is created through the ClientManager.connectToServer() calling, an independent, asynchronous thread it's automatically created.
The above thread handles the three main methods of the class: @OnMessage, @OnOpen and @OnClose.
In fact, the user which used the connectToServer() method has no control over this thread, because it has no reference for it.
So, the synchronization between the Application class (which handles the starting of the application and the drawing of the main chat) and the WebSocketHandler one become very difficult.
The monitor made the trick.

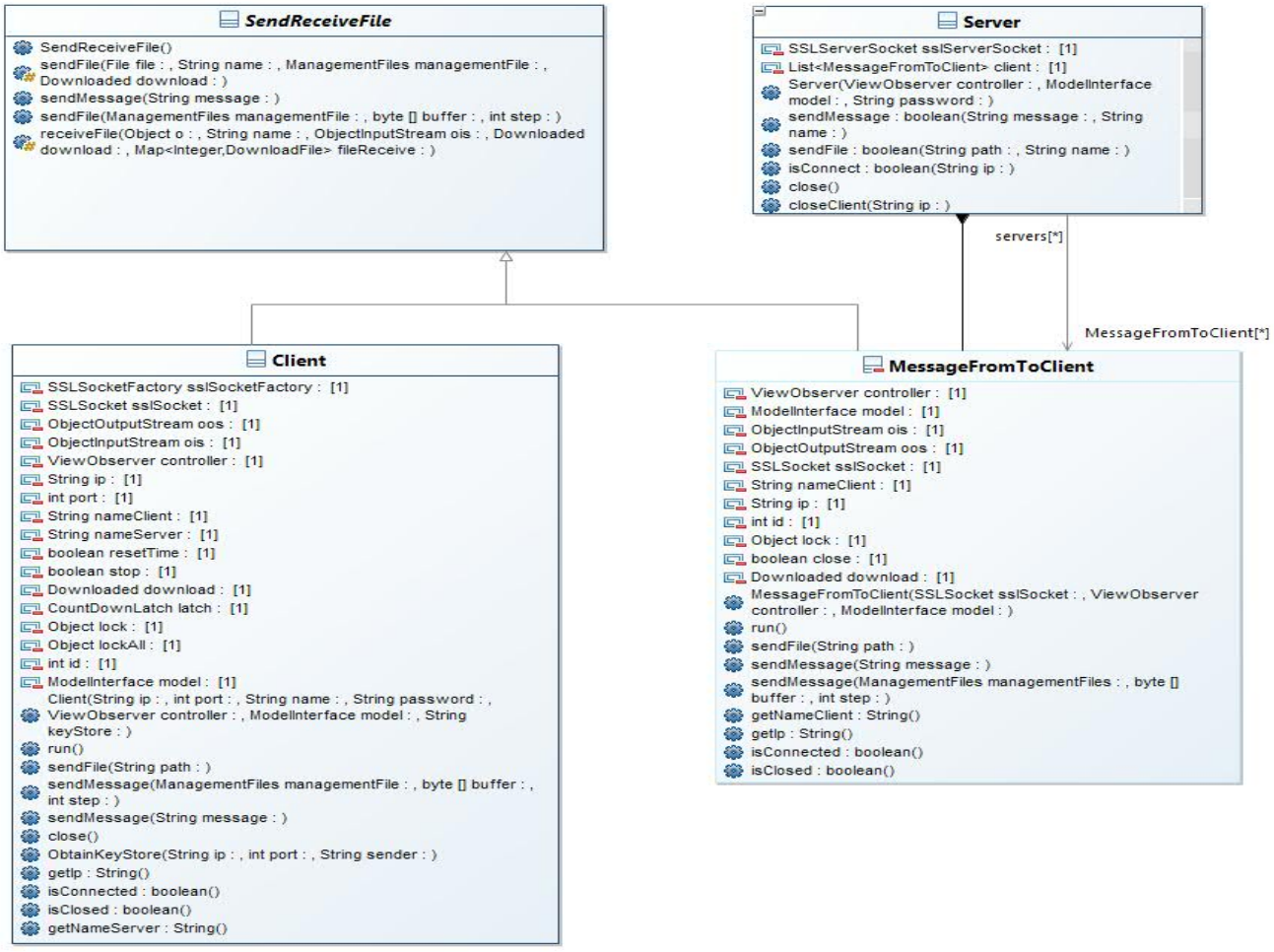
Used Libraries:

BaloonTip [1]**:** a simple library which enables to create lightweight, fast "ToolTip" components by specifying only two parameters (the actual message to show, and the component to attach to).
In this project it's has been used in the splashscreen frame, were a tooltip in the nickname field it's pretty good looking.

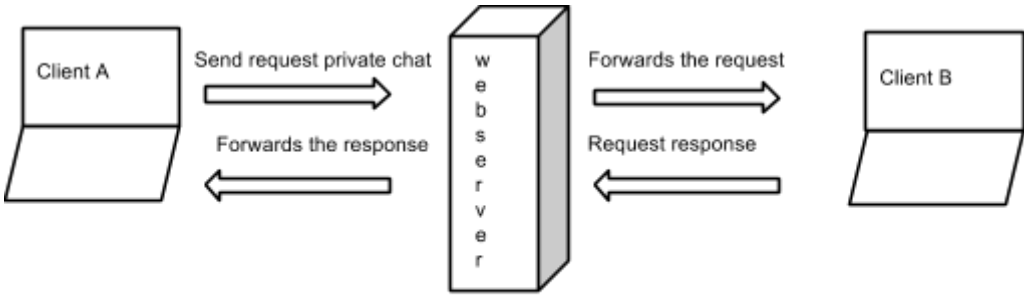
Junite [2]*:* This is a very handy library which limits the number of concurrents instances of an application by a specified number.
In this project it has been used mainly because duplicate istances of CryptoChat may cause some serious troubles with the encrypted chat, causing exceptions to be thrown due to the same socket ports used twice.

5.2 Detailed Engineering: Francesco Cozzolino’s work

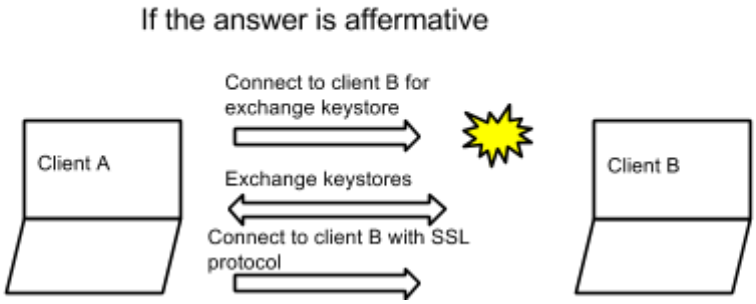
Francesco Cozzolino's work is focused on creation of private chat with SSL protocol and all management between the GUI and the handling of sends/receives messages/files. Obviously the main classes of this management are the Client and Server classes; they use a template method.
Here's the correspective UML diagram of this classes:



How does it work a private chat ?



Request type from Client A :Type.REQUESTPRIVATECHAT
Response type from Client B : Type.YESPRIVATECHAT
Type.NOPRIVATECHAT



(for some details about the messages involved with the webserver, see point 5.1)

In order to establish a connection between two users, is necessary to exchange their keystores.
When the client “A” wants to talk with the client “B”,it sends a request of private chat to the webserver, that forwards the request to the client “B”.
Client B then receives the forwarded message from the webserver: a JOptionPane with “Yes” or “No” is shown as result.
When client “B” answers,it sends the “answer” message (YESPRIVATECHAT or NOPRIVATECHAT) to the webserver,that forwards again the response to the other client.
If the answer is of type YESPRIVATECHAT, the IP and router’s socket ports of the sender is included.

If client “A” receives an affermative type response, it tries to connect to the server of user “B” (with the provided ip) to exchange their keystores. If the exchange is successful, client “A” tries to connect to the server of client “B” with SSL protocol and finally they can chat.

The same routine happens with the option “Chat to” with the difference that the webserver isn’t involved.
In fact, in this option the user inserts directly the ip to connect to.

As result, user “A” directly tries to connect to the server of “B” to exchange keystores (obviously, an user prompt of “yes” or “no” is also shown here).
When the connection is established, the user that is, in fact, a client (it does not hosts the connection) starts a timer; if it records an inactivity (no messages received or sent) for one minute or over,the connection is automatically closed. When one of the two clients would like to chat after the timeout connection, the “private chat” handshake shown above will not be necessary to be run again: a map structure stored in the model class takes trace of all connected users in the session, and stores their informations such as Ip and socket ports; However, if a client close normally the chat, all of its informations are deleted.
For this type of connection each users need to have two router’s socket ports open.
One is used for the socket to be able to exchange keystores, the other one is in order to receive an SSL connection.

In addition to sending/receiving messages, it’s possible to send and receive files. If there is no connection between two users the same mechanism mentioned before is needed in order to send or receive files; in other words, a private chat must be established to be able to send or receive files.
Since the stream for I/O is the same for both string messages and files, to avoid starvation the files are sent in chunks of bytes.In both sides the users are advised with a message which tells the starting (or finishing) of the stream.
In addition, a JProgressBar it has been implemented in order to see asynchronously the progress of the stream (in the “Downloaded” frame).

What is a keystore ?

“A Java KeyStore (JKS) is a repository of security certificates, either authorization certificates or public key certificates used for instance in SSL encryption” [3] (source : Wikipedia)

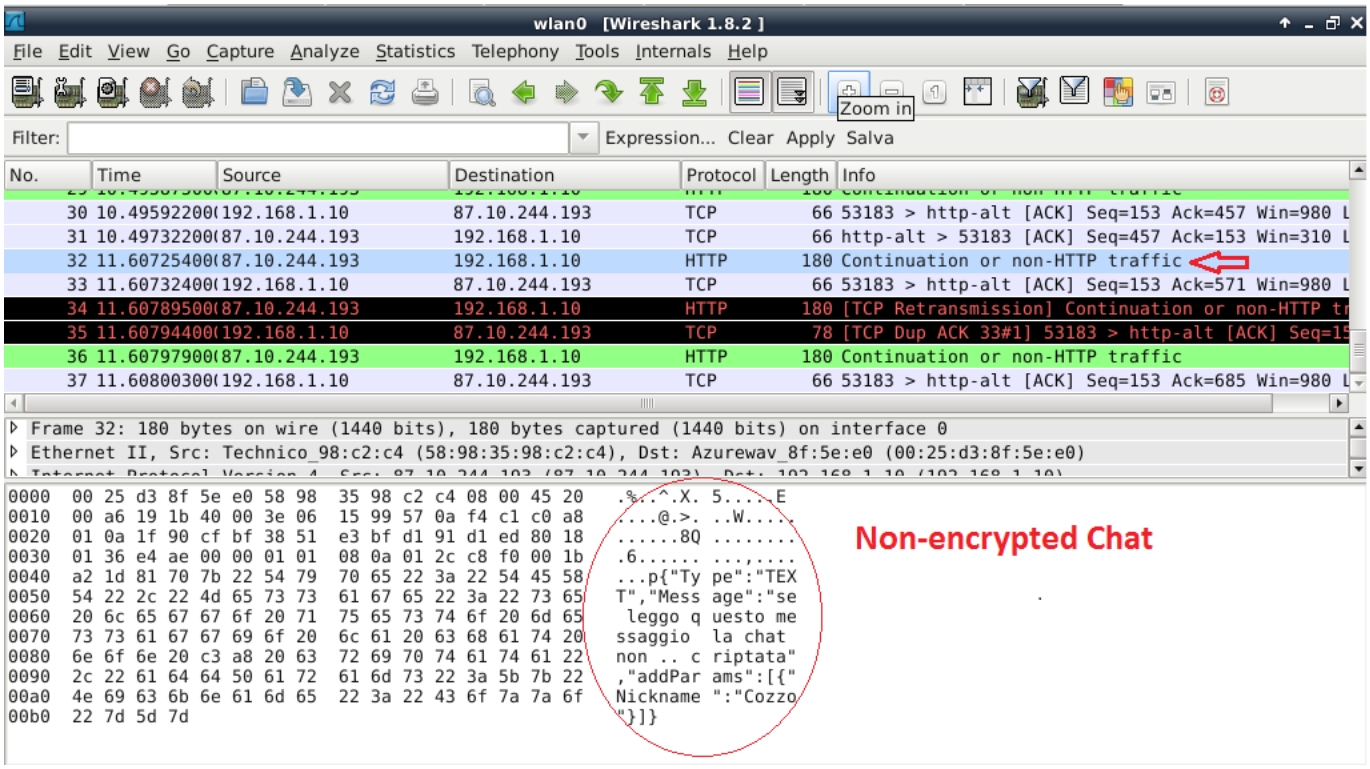
Like mentioned before, to have an encrypted chat is necessary to have the public key of the client who wants to chat and his own keys. Those keys are created automatically with a .bat or .sh script (depends from the O.S.) that contains some information of user. For improved security, scripts and the keystores are created automatically at every session; the keystores are protected with a random alphanumeric password.

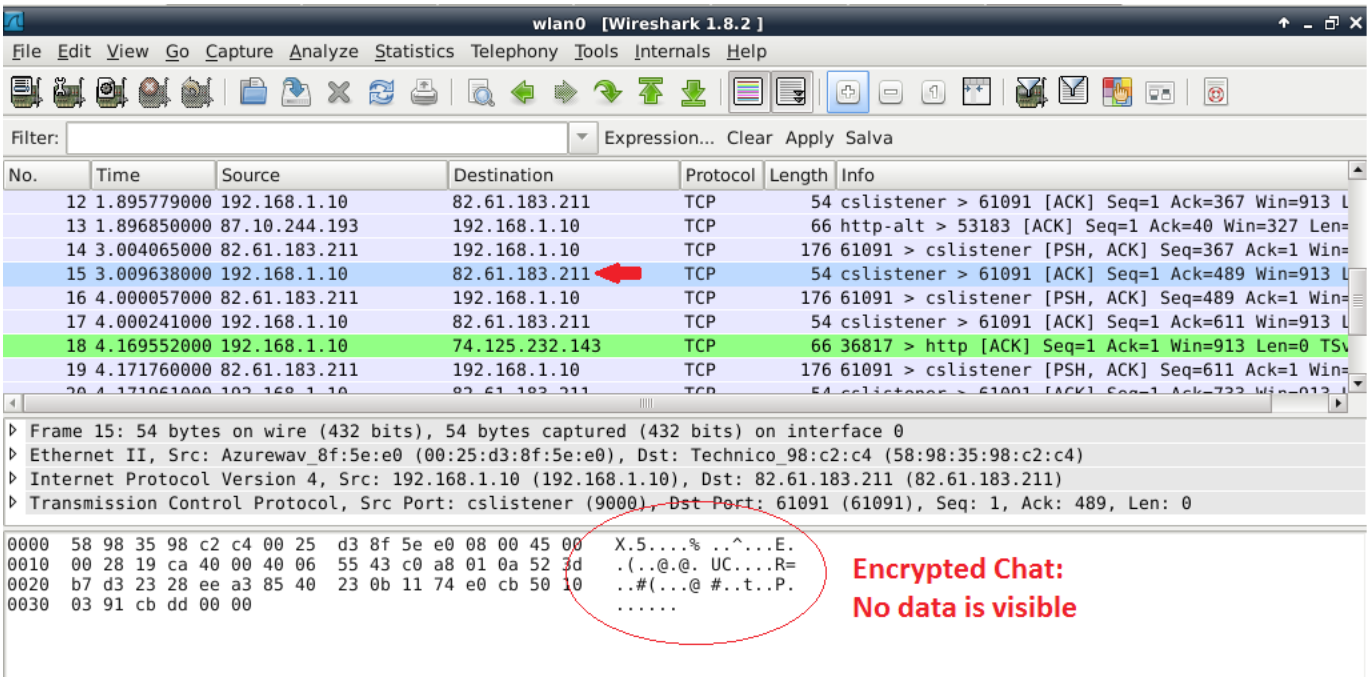
6 Testing

CryptoChat has passed many tests before this (early) release.
althoughit needs some other other important fixes, the main functions of this application are perfectly working.
However, the correct setup of this application is really important and it should not be ignored:

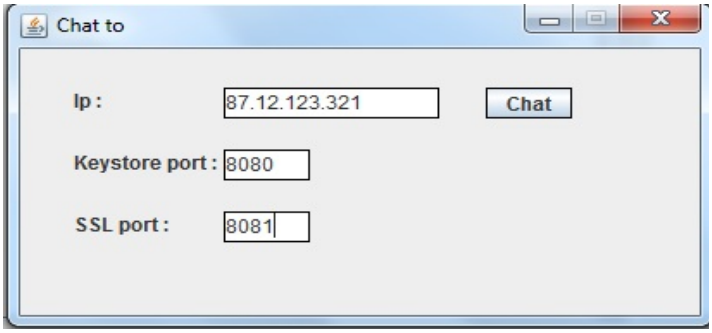
1. Check if you have a working (and binded, with JAVA PATH) JDK version 1.7 on your system.
2. Please user the given .jar file of the application within the given resources directories: in other words, extract the application .rar and leave it (and use it, of course) as it is.
3. Be sure that wherever you extract your application you have all the writing rights needed for your user (i.e. in linux set your main directory as 777).
   Please also apply “executable” rights on the provided jar.
4. Before to launch the application, be sure to have two router’s ports open or the private chat will not work correctly. For set which router’s ports use for private chat, go to ../ClientChat/config/ and open the file “config” with any text editor and set the router’s ports you want to actually use. If the file doesn’t exist, at the first running of application , the config file will be created automatically with default ports 9998 and 9999
5. If you are encountering some issues with a linux machine, run the provided jar from shell with java -jar command.

Please note: doing a “usual” JUnit test would have been really difficult to perform in this project.
A manual series of test have been made to understand if the application really work, and a small part of it it’s documented here)
Here’s some testing of the actual encryption of the messages sent:

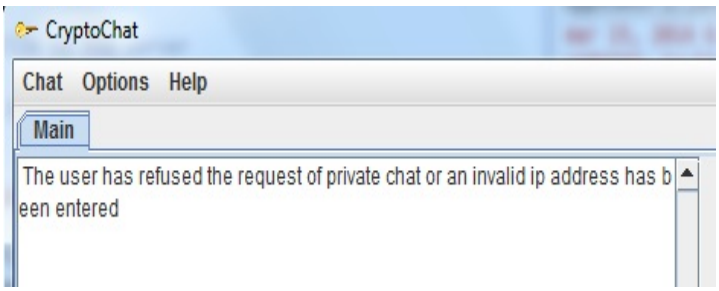




Here's another test that describes what happens when an user uses the option "Chat to" and tries to connect to an ip address not currently connected :



If any ServerSocket instance at the specified address exists (or the socket ports are closed) an exception will be thrown.
Obliviously, this exception will be caught for the user as a simple error message.



7 Final Considerations

In the workflow of this application, a small initial part it has been implemented before the agreement in the easi.polocesena.unibo forum.
More precisely, a first basic connection between client and webserver (and a first basic private chat too) were implemented to test the real practicability of the application.
No GUI were implemented in this part, it was all by Console messages.
Anyway, all the workflow within the requested 100 hours has its start from the first commit on the HG repository.

7.1 In-development changes

The workflow of the application got's a small number of radical changes from its start.
A noticeable example can be the IP communication from client to client: on the beginning of this development,Stefano Belli thought the IP would be easily extracted from the web server referencing a to his client connection (using Session object instance).
In fact, this capability was offered by default by webserver like tomcat.
However, the developers found that with a glassfish server this capability was not only not present, but even impossible to implement.
This issue were resolved with a direct exchange of the ip by the client: now it's the client itself that provides his own IP.

Another istance can be the open ports issue: due to the needing of publishing this project to a commision without the possibility to test the application in a real-world simulation (without, in other words, test it with different connections and opened socket ports for each connection) the developers needed to make some deep changes in the private communications: now a client can specify which socket ports wants to use and they are directly communicated to the correspective client they want to chat to.

7.2 Workgroup Interactions

The workgroup was formed by two members with some experience with each other : they already worked together in about 2-3 project before and so no interaction problem occured.
All the suggestions, bugfixes and request were handled at the same level from the two members, feeling free to comment each other work, or criticize something (with care, obviously).

The works were well separated on its task: Stefano Belli, for example, knows very little about the core code of Cozzolino, like all the private chat "actual" communication.
Besides, Cozzolino knows very little about the server infrastructure and public communication.
However, some code were completely shared due to similar needings.

7.3 The future of Cryptochat

It's developers intention to publish this project on some web hosting repository for open-source code like github or sourceforge.
This mostly because it's quiet hard to find documentation, tutorials or even some example code based on the technologies used in this project, and it would be useful in case of needings (or at least we hope so).

We are actually sorry of the complicated setup needed to make this application able to work normally.
Our goal on the future "commits" it's to make this application simpler.

Also, the developers hopes to finish some project's details like a multi-channel communication for public chat (a user should be able select which channel connect to) and also a multi-client support for the private chat.

Even if the application was designed as a exam project for OOP, the project's technologies involved are not casual: we believe that this has been a good chance to understand some interesting (and new) technologies such as the websocket.

8 Bibliografy & Credits

[1] BalloonTip: https://balloontip.java.net/
[2] JUnique: http://opensource.openjar.net/java/general-purpose/junique
[3] *See* http://en.wikipedia.org/wiki/Keystore

Thanks to user **A43** of www.freesound.org for sound Toc 02.wav.
http://www.freesound.org/people/A43/sounds/9448/
All the credits related goes to user **A43**.

Thanks to user **cameronmusic** of www.freesound.org for sound Notification 3 (bad).wav.
http://www.freesound.org/people/cameronmusic/sounds/138413/
All the credits related goes to user **cameronmusic**.