ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Un entorno de test colaborativo en JavaScript para Siette

A collaborative testing environment in JavaScript for Siette

Realizado por

Jesús Cautivo Chacón Ávila

Tutorizado por

Ricardo Conejo Muñoz

Departamento

Lenguajes y Sistemas Informáticos

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Noviembre 2015

Fecha defensa:

El Secretario del Tribunal

Resumen: En este proyecto nosotros vamos a hacer una pasada por un sistema colaborativo de resolución de test. En este entorno el estudiante resuelve preguntas en un grupo de estudiantes en tres fases diferentes (individual, colaborativa y final). Este sistema ha demostrado que los alumnos que resuelven los test en este entorno mejoran sus respuestas. Esto se debe a que los estudiante aprenden como sus compañeros se enfrentan a las cuestiones planteadas y como las resuelven. Este sistema esta actualmente funcionando en la plataforma siette, pero esta teniendo problemas de compatibilidad con algunos navegadores y esto presenta un problema para desarrollar nueva funcionalidad. Nosotros queremos resolver esto con la utilización de HTML5, CSS y JavaScript. Este Sistema será diseñado para funcionar en conjunto con la plataforma de resolución de tests (Siette) en un modo que llamaremos modo colaborativo que solo será usado para resolver test en grupo con otros compañeros. Siette nos propone un entorno donde trabajan profesores diseñando test y alumnos que testean su conocimiento.

Palabras clave: Siette, applet, java, JavaScript, HTML5, CSS3, colaboración, aprendizaje, tecnología de la educación, tests y aprendizaje electronico.

Abstract: This project is a documentation and implementation of a collaborative system that provides an environment where the students can learn within a group testing. This environment is presented in Siette. Siette is a platform where a student can solve quizzes to improve and test his knowledge. This environment presents to the student a test with three steps where he can solve questions and discuss about his response. The student learns how the other peers answer the questions, and learns the methodology to face a question. This system has demonstrated that the student improves his ability to solve the questions individually. In the project, we are going to replicate the functionality of an applet that now covers the functionality that the environment needs. This applet has problems of compatibility in some browsers. We want to use JavaScript, HTML5 and CSS3 to solve them. With this implementation, we want to create a system that can grow in a simple way because right now the applet is deprecated in some browsers and this is a limitation to build new features.

Keywords: Siette, applet, java, JavaScript, HTML5, CSS3, collaborative environment, ed-tech, testing and e-learning.

1 Index

2 INTRODUCTION. **1**

3 METHODOLOGY. **1**

4 ANALYSIS OF REQUIREMENTS. **2**

4.1 VERSION ONE. 2

4.2 VERSION TWO 2

5 TECHNOLOGIES WE WILL USE TO SOLVE IT AND WHY. **3**

5.1 HTML5 4

5.2 JAVASCRIPT 4

5.2.1 jQuery *5*

5.3 CSS 5

5.3.1 Bootstrap *6*

6 SYSTEM DESCRIPTION. **6**

6.1 CLIENTS 6

6.2 SERVER 7

6.3 PROTOCOLS. 7

6.3.1 Messages *7*

6.3.2 Protocols *15*

7 IMPLEMENTATION. **21**

7.1 OBJECTS 21

7.1.1 User *21*

7.1.2 Command *21*

7.1.3 Message *22*

7.2 DATA STRUCTURES 23

7.3 FILES AND METHODS 24

8 USER INTERFACE (UI). **27**

9 CONCLUSION (ENGLISH). **30**

10 CONCLUSION (SPANISH). **30**

11 BIBLIOGRAPHY. **31**

12 APPENDIX ONE: CODE **32**

2 Introduction.

During all document, we will speak about a system implemented in SIETTE that consists in a chat system to allow that multiple students can work together in a test, to share their knowledge.

This project is based in the idea of a student can learn easier with other student instead of from a teacher. The reason of this is very natural, because between students the language is the same that between the teacher and the student, normally this happens because the teacher needs to teach for a class and this produces that he uses a language more “standard” and normally more complex.

The project has two modes, the first mode is based in a common room when a group of user can send and receive messages. The second mode is a collaborative mode that work in together with the platform SIETTE where the user can do test with a group of users. In the next paragraph we are going to describe the process that the user follows to answer the questions.

In the first step, the system presents the same set of questions to a group of students. Each student in the group should answer the same question twice. An initial response is given individually, without knowing the answers of others. Then the system provides some tools to show the other partners’ responses, to support distance collaboration. Finally, a second individual answer is requested. This system we can check and improve the knowledge of the student.

All this system now is developed with an applet in Java, the finality of this project is replicate the system in JavaScript, because we have problems with the compatibility of some browsers how for example Google Chrome that in the version 45 has deprecated the applet and not allow to run the applet.

3 Methodology.

For this develop, we have used a model based in the iterative method, we will define three steps for our system, these steps are analysis, coding and testing. In the first step we are going to do an evaluation of the different issues based on our requirements. In the step of coding we will code a version of our system, fixing several issues in the case that we have problems in the code or adding new functionalities. Finally, in the step of testing we will test the current version looking for bugs and similar problems to develop the next version, we will repeat the process to develop a stable version that covers all our requirements. During the whole process, we want to develop two versions of this system. The first one has a common room where the student creates a private group to do his test with some peers. In the second version we develop a chat where the students discuss about their answers with the other student and can see what is the actual state of his peers, in the next character we define more exhaustively the process, to obtain the requirements of the two versions, in this section I only wanted to introduce our two versions.

4 Analysis of requirements.

In this section we are going to talk about the requirements and how to verify this when our system will be ready. First, we define the whole system and then, we will do a list with the principal requirements and their test to verify if they are fulfilled. We need to develop two versions. In one hand, in the version one we need to develop a chat system that consists in creating a public chat room for a subject; this version is active when the student enters in the subject to join a private room with other students to do a collaborative test. In the other hand, for the version two we need to develop a chat room that is shared by a private group of student that want to work together to pass the test.

4.1 Version one.

In this section we will do a list with all the requirements what we need to cover during the development. Actually, SIETTE has a chat system for the collaborative test that works, but has a big pain that is the fact of this software is developed in java with an applet and the problem is that the new versions of the browsers don't have support for this technology. An example can be chrome that has removed the support to this technology in the version 42. With this premise, we need to develop a chat supported by all browsers or by a big group of them. Now we are going to do a list with all of our requirements.

1. Support in a big group of browsers (Chrome, Firefox, Internet Explorer, Opera, Safari).
2. The system will need to work fine with the actual version of the client (Java applet) and with the server (Java Servlets).
3. The client will need to send and receive messages to manage the chat room in real time.
4. The client will need to manage a list with users in real time.
5. The client will have to be able to add and remove users to a chat room.
6. The client will have to be able to be displayed in two grids, vertical and horizontal.

With last list of requirements, we cover our version of the chat system.

4.2 Version two

In this section we are going to analyze the version two of our system. In this version we have the same problem of compatibility that in the version one and we need to solve it in our version. This version starts with the version one because is an upgrade over it, we need to solve the same problem that the version one, also this version needs to manage the private room and the steps in the test mode. The test mode has three steps. In the step one the student cannot use the chat to speak with his peers, because this is the individually step he needs to answer the question that SIETTE proposes for the group of student. The step two is the step of discussion, in this step the student can use the chat to speak with the others peers and argument his response, and need to do a response with the opinion of the others student in the private chat. The step three is the final step, in this step the student needs to give a definitive answer of the question that SIETTE proposes. This version has the next list of requirements.

1. Support in a big group of browsers (Chrome, Firefox, Internet Explorer, Opera, Safari).
2. The system will need to work fine with the actual version of the client (Java applet) and with the server (Java Servlets).
3. The client will need to send and receive messages to manage the chat room in real time.
4. The client will need to manage a list with users in real time.
5. The client will have to be able to add and remove users to a chat room.
6. The chat will have to manage the three steps for each question.
7. We need that the chat will be enable and disable when the steps require it.
8. The client will have to be able to be displayed in two grids, vertical and horizontal.

5 Technologies we will use to solve it and why.

How we had seen our principal pain is the compatibility with the browser, because the browsers are remove the support of this technology. Now the applications are being developed in HTML5, CSS and JavaScript for the client, we want to use this technology for us develop. HTML5 is now a standard in the language of coding website. To build a friendly look and feel, we use a framework that can do a homogeneous style to our chat and can improve the user experience and the interface (UX/UI). The framework to develop our style for the system will be Bootstrap, that is a very common framework in the current design of webs. For the part of JavaScript, we use jQuery, because it gives to us a framework very hardy to develop and now is the referent in a big group of projects. We need to use jQuery because it has compatibility with many browsers, it is interesting to talk about the possibility of using jQuery v1 to support Internet Explorer 6 and over because jQuery v2 does not have support for this browser, we can see the different type of support for browsers that the versions of jQuery have in the next screenshots.

Browser Support

Current Active Support

<table>
<tr><th></th><th>Internet Explorer</th><th>Chrome</th><th>Firefox</th><th>Safari</th><th>Opera</th><th>iOS</th><th>Android</th></tr>
<tr><td>jQuery 1.x</td><td>6+</td><td rowspan="2">(Current - 1) or Current</td><td rowspan="2">(Current - 1) or Current</td><td rowspan="2">5.1+</td><td rowspan="2">12.1x, (Current - 1) or Current</td><td rowspan="2">6.1+</td><td rowspan="2">2.3, 4.0+</td></tr>
<tr><td>jQuery 2.x</td><td>9+</td></tr>
</table>

Figure 1 – Compatibility of jQuery

Other point to analyze is how we will implement the system of messages to send and receive information of the server. To solve it we will use AJAX, because is a good way to send and receive information of the server. We want to use jQuery because it has an excellent support of this kind of technology.

In summary, we use the next list of technologies:

- **HTML5.**
- **JavaScript (jQuery 2.1.4** and **jQuery 1.11.3, AJAX, JSON).**
- **CSS (Bootstrap 3.3.5).**

5.1 HTML5

HTML5 is a markup language used for structuring and presenting content on the World Wide Web. It has been finished, and published, on October 24th 2014 by the World Wide Web Consortium (W3C). **HTML5** adds many new syntactic features. These include the new **<video>**, **<audio>** and **<canvas>** elements. These features are designed to make it easy to include and handle multimedia and graphical content on the web. Other new page structure elements, such as **<main>**, **<section>**, **<article>**, **<header>**, **<footer>**, **<aside>**, **<nav>** and **<figure>**, are designed to enrich the semantic content of documents.

HTML5 specifies scripting application programming interfaces (**APIs**) that can be used with JavaScript.

- Timed media playback.
- Offline Web Applications.
- Document editing.
- Drag-and-drop.
- Cross-document messaging.
- Browser history management.
- MIME type and protocol handler registration.

Figure 2 – Logo HTML

5.2 JavaScript

JavaScript is a programming language that can be included on web pages to make them more interactive. You can use it to check or modify the contents of forms, change images, open new windows and write dynamic page content. **JavaScripts** only are executed on the page(s) that are on your browser window at any set time.

The basic part of a script is a variable, literal or object. A variable is a word that represents a piece of text, a number, a boolean true or false value or an object. A literal is the actual number, or piece of text, or boolean value that the variable represents. A part very important of **JavaScript** is the management of the event that it offers to developers. Events can be used to detect actions, usually created by the user, such as moving or clicking the mouse, pressing a key or resetting a form. When triggered, events can be used to run functions.

As an example for use it is the following situation. A person clicks a submit button on a form. When he clicks the button, we want to check if an email is valid. When the event of submit is triggered, **JavaScript** starts his work. Now we can build a script that analyzes the email that the user has introduced and it detects if is correct. In the case that is correct, the information is sent to the server and this can permit that the server does not need a hard control over the data what receives. In the case that the information is not correct, we can notify the user with a message in the form because he has introduced an invalid email and **JavaScript** stops sending information to the server.

For our project we utilize **JavaScript** to build the code that manages our clients, but to reduce the complexity of the code and the management of the event we are going to use the framework **jQuery** as we have explained in section 5.

5.2.1 jQuery

jQuery gives to us a very good abstraction over the triggering of event, the management of the **HTML** to search and add information with his module of **sizzle** and offers a complete module of **AJAX** for the exchange of information between the client and the server. All the information that a client and the server share using the network is in **JSON**.

Figure 3 – **Logo jQuery**

Figure 4 – **Logo Sizzle**

JSON (JavaScript Object Notation) is a lightweight format to represent information for the exchange of it. **JSON** is a subsection of **JavaScript** that does not require to use **XML**.

Figure 5 – **Logo JSON**

5.3 CSS

CSS is the acronym for: 'Cascading Style Sheets'. **CSS** is an extension to basic **HTML** that allows you to style the web pages. **CSS** use selectors to define the style over an element of our **HTML** page. A selector is an expression that is associated to an element or a group, an example of this can be:

- *: this is a selector for all elements in our document.
- **p**: this selector defines the style to all tag **<p>** in our **HTML**.
- **.bold**: With this selector we can identified all tags that contains the value **bold** in his attribute class.

In **CSS** we can use operator to build selector more complex and specific, an example can be the operator +. div + p Selects all **<p>** elements that are placed immediately after **<div>** elements. To do style to our system we are going to use the framework Bootstrap.

5.3.1 Bootstrap

Figure 6 – Logo Bootstrap

Bootstrap is a powerful front-end framework for faster and easier web development. It includes **HTML** and **CSS** based design templates for common user interface components like Typography, Forms, Buttons, Tables, Navigations, Dropdowns, Alerts, Modals, Tabs, Accordion, Carousel and many other as well as optional **JavaScript** extensions. **Bootstrap** also gives you ability to create responsive layout. To develop the systems, we are going to use the version **3.3.5**. We only use a simple version of **bootstrap**, because in our versions components like JavaScript, modals and dropdowns are not necessary.

6 System description.

In this section we are going to discuss about the system of messages that connects the clients with the server. For explaining all of our examples we use only two clients (client1 and client2), one server (server) and we will use nodes to refer the clients and the server. Now the chat system has 5 protocols which are used to communicate with the other client, and it has 3 type of messages (command, message and message container) that can be sent to the server. All the clients send theirs messages to the server, it processes the message and sends other message to the client that needs to receive information.

6.1 Clients

A client is an implementation that manages the information of an user for connecting to a chat server. It has an interface that allows the a user to send information to a server, but with a structure of this communication. Now our clients have several variables to define who is him. We are going to see all of that in the following list:

- **CLIENT_ID**: it is a unique number to identify a user.
- **CLIENT_NAME**: it is the name of the user in the platform.
- **ASIGNATURA**: it is a variable when we save the ID of the subject where the user displays his actual session of our chat system.
- **IP**: It is the IP address of our client.
- **SESSION**: It is a unique sequence that identifies the client to access to a resource in the server side.

A client has three sections, the first one is a list of users that allows to see which users are online and available to speak. The second section is where a user can see the messages that have the current room of the chat, this section can be enabled or disabled depending of the stage of the user. Finally, in section three, we have an input where the user writes their messages and send them to the current chat room.

6.2 Server

A server is an implementation for a manager of the rooms and users. This server makes the users connect to different rooms. The server has the functionality of managing the users that are connected to the subject and send them information. It manages two channels for each client, messages ("data channel") and commands ("Cmd channel"). The message channel is used to send and receive messages related to the texts messages between clients. The command channel is used by clients and the server to send actions, for example "wait responses", "update user status" ...

The server is able to manage three methods to send information from a client to another client(s), "all", "others", "client_id". Most messages use a parameter to define it ("to"), each value of this parameter has different function. Firstly, in the case of "all", a client sends a message to the server, the server processes the packet and sends information to all clients that will be in a room included the user that sends the message to the server. This process is like a broadcast to all clients. Secondly, in the case of "others", the process is the same that "all", but the client that initializes the conversation with the server does not receive the message. Finally, in the case of "client_id", the server is like a bridge between two clients. A client is the sender and the other is referenced by his client_id. All transactions with the server are in JSON, and the server creates a thread for each petition received.

6.3 Protocols.

In this section we will to analyze all protocols that our client will be able to manage, Firstly, we are going to analyze separately the type of messages and command that our client will be able to manage. Secondly, we will analyze each protocol that we will need to implement.

6.3.1 Messages

6.3.1.1 Command

The command is a message that the clients send and the server does actions over the other nodes. A command can have multiple function, we define it with a type, now we have several types such as **INIT**, **ACK**, **NACK**, **ADD_USER**, ... But we will discuss about them later. Now we are going to talk about the attributes that have a command and we will define all of these attributes. The list of attributes is:

- **ID**: it is a number that identifies a command.
- **CLIENT_ID**: it is the ID of the student in SIETTE.
- **CLIENT_NAME**: it is the name of the student.
- **SESSION:** this parameter represents the current session of an user (identified by subject when the room is created).
- **IP**: it is the IP address of the client.
- **ERROR:** Normally this parameter is null and only is used when a client or the server sends **NACK** with an error.

- **TYPE**: In a command we have multiple types depending of the action that we want to do, we will dedicate a section to explain it.

```
this.id = null;
this.client_id = null;
this.client_name = null;
this.session = null;
this.channel = null;
this.ip = null;
this.error = null;
this.type = null;
```

Figure 7 - Implementation of command.

6.3.1.1.1 Type

In this section, we define all types that can have a command and we do a summary of all types s command can have. All the types are:

- **INIT**
- **END**

6.3.1.1.1.1 INIT

This command is sent by the client when access to the room. With this command the server creates all structures that needs to manage the user and the user login in the room. The use of this command is very easy, the client starts the conversation with the server. This client will send a command with the next parameter.

- **ID**: It is the timestamp that represent the command.
- **TYPE**: **INIT**.
- **CLIENT_ID**: It is the client ID of the user.
- **CLIENT_NAME**: It is the name of the client.
- **IP**: This is the IP address of the client.

The server processes the command and sends a command of **ADD_USER** to others clients that can be in the same room, we specify this process in the next sections. The server sends to the client that started the conversation a command of **ACK** in the case that all process is fine or a **NACK** if the command of **INIT** has produced an error. Next figure represents this process.

```
Se ha enviado al servidor --> {"command":
{"id":"null","client_id":"111636","client_name":"jesus2","session":"IDASIGNATURA#22","channel":"null","ip":"127.0.0.1","error":"null","type":"100"}}
```

6.3.1.1.1.2 END

This command is sent when a client wants to log out of the chat room, normally this happens when the client closes his browser or when he changes between subjects. The process is like the process of **INIT**. The client sends a command with the next information:

- **ID**: It is the timestamp that represents the command.
- **TYPE** → END

- **CLIENT_ID**: It is the client ID of the user.
- **CLIENT_NAME**: It is the name of the client.
- **IP**: This is the IP address of the client.

The server processes the command and it confirms the message to the client that started the conversation and the server notifies with a command of **REMOVE_USER** to the other clients, but we will analyze this later. Next figure represents this process.

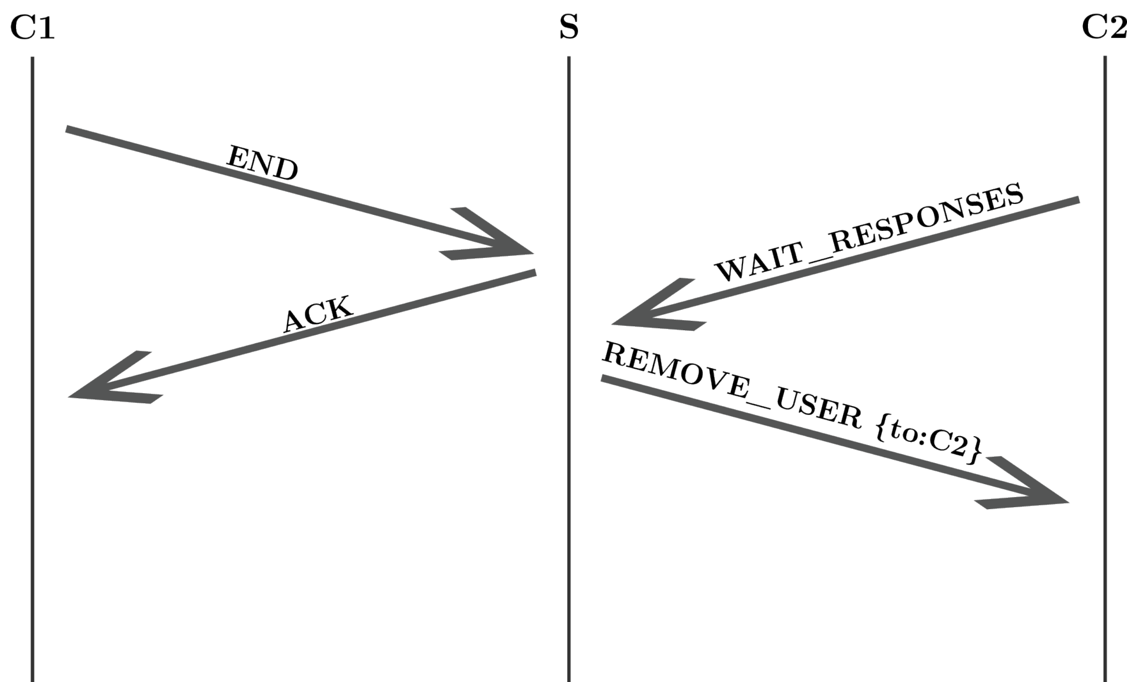

Figure 8 – Diagram command END

6.3.1.1.1.3 ACK and NACK

These commands are sent for the client and the server to confirm the reception of a command or a message. The command of **NACK** is the unique that uses the parameter **ERROR** of a message, the value of this parameter in a String with a summary of the problem what has occurred in the server or in the client with the process of last command o message.


```
{"command":{"id":null,"error":null,"session":null,"client_id":null,"type":14,"channel":null,"client_name":null,"ip":null}}
```


Figure 9 – Example ACK

6.3.1.2 Message

A message is used by the client or the server to share information with other nodes. A message has the next variables that determine its function:

- **ID:** It is a unique number that identifies the message, this number is a concatenation of the timestamp, we need to concatenate day, month, year, hour, minutes, seconds and milliseconds. 21112015172312234 →21/11/2015 17:23:12:234
- **CLIENT_ID:** It is the number that identifies the user in the data base.
- **CLIENT_NAME:** It is the name of the client.
- **SESSION:** It is the room where the user is login.

- **CHANNEL:** It is the channel where the server processes this message ("**Cmd channel**", "**Data channel**").
- **TYPE:** It is the type that defines the action (**WAIT_RESPONSES**, **ADD_RESPONSE**, **UPDATE_USER_STATUS**).
- **IP:** It is the actual IP address of the client.
- **ERROR:** It is used to send error information of a process.
- **TO:** It is the receptor of the message ("all", "others", "client_id").
- **REPLY_TO:** It is used to send a copy of the message to another user different of the users specified in the parameter **TO**.
- **SUBJECT:** It is the subject of the message that a client sends in a chat.
- **BODY:** It is the content of message that sends a user to the chat.
- **ITEM:** It is the number of question that a student is answering in a test.
- **STEP:** It defines the state of the answering (Individual, collaborative, final).

```
this.id = null;
this.client_id = null;
this.client_name = null;
this.session = null;
this.channel = null;
this.type=null;
this.ip = null;
this.error = null;
this.to = null;
this.reply_to = null;
this.subject = null;
this.body = null;
this.item = ITEM_NULL;
this.step = STEP_NULL;
```

Figure 10 – Parameters of a message

A message has multiple functions like a command. In next sections we are going to analyze these types. This type of packet is used to share information between the nodes, a common use of it is when a client communicates his actual state over a test, this communication is used to keep the actual state over the question or over the test. We need this information because a client has a list with the users in the test and needs to know who is available to speak and discuss to learn. By default, all messages have the client_id, client_name, the channel, the session, type and the id with a value and the other by default his value is null. The other parameter value depends of the message that a client sends, for example with a message of **UPDATE_USER_STATUS**, the parameter item and step will have an integer value that represents the state of the user.

```
var ITEM_NULL=-1;
var STEP_NULL=-1;
var INIT = 100;
var END =  199;
var ADD_USER= 1001;
var REMOVE_USER = 1002;
var WAIT_RESPONSES= 1003;
var UPDATE_USER_STATUS= 1004;
var ADD_RESPONSE= 1005;
var END_TEST= 1006;
```

Figure 11 – Type of message

6.3.1.2.1 ADD_USER and REMOVE_USER

These commands are used by the server to notify a client that this room has a new user or a user has left it. The client needs to send to the server a message of **WAIT_RESPONSES**, when the server receives the message, if it has a command of **ADD_USER** or **REMOVE_USER,** it sends a command with the type of action. Next figure represents this process.

Se ha recibido del servidor --> {"messageContainer":{"messages":[{"message":{"to":"111636","body":"","reply_to":"","error":null,"subject":"","type":1001,"ip":null,"id":null,"session":"IDASIGNATURA#22","item":-1,"client_id":"111635","client_name":"jesus1","channel":"Cmd+channel","step":-1}},{"message":{"to":"111636","body":"null","reply_to":"null","error":null,"subject":"wait_responses","type":1003,"ip":null,"id":"21112015172217271","session":"IDASIGNATURA#22","item":1,"client_id":"111635","client_name":"jesus1","channel":"Cmd+channel","step":1}}]}} outputlog.js:25

Figure 12 – Example of ADD_USER

6.3.1.2.2 WAIT_RESPONSES

It is a possible value of the parameter type in a message, normally is used by a client to communicate to the server that is waiting for information. The server also uses this message to know that a user is connecting in a room, when a server does not receive information of a client report to the other client that a client has left the room. The functionality of this kind of message is very simple, a client sends a message to the server with the parameter type setter to **WAIT_RESPONSES**. When the server receives the message, it searches all information pending to be sent to the client and it mounts multiples messages in the case that the server needs it, then the server sends to the client the information. In the case that the server does not have information to send to the client, it only sends a command **ACK** to confirm that it has received the message. An example of a message can be:

- **ID**: It is the mark of the time that represents the message.
- **CLIENT_ID:** It is the number that identifies the user in the data base.
- **CLIENT_NAME:** It is the name of the client.
- **SESSION:** It is the room where the user is login.
- **CHANNEL:** It is the channel where the server processes this message ("**Cmd channel**", "**Data channel**"), in this case the value id "**Cmd channel**".

- **TYPE:** It is the type that defines the action (**WAIT_RESPONSES**, **ADD_RESPONSE**, **UPDATE_USER_STATUS**), in this case is **WAIT_RESPONSES.**
- **IP:** It is the actual IP address of the client.
- **ERROR:** null.
- **TO:** null.
- **REPLY_TO:** null.
- **SUBJECT:** “wait_responses”.
- **BODY:** null.
- **ITEM:** null.
- **STEP:** null.


```
Se ha enviado al servidor --> {"message":{"id":"21112015172222360","client_id":"111636","client_name":"jesus2","session":"IDASIGNATURA#22","channel":"Cmd channel","ip":"127.0.0.1","error":"null","type":"1003","to":"others","reply_to":"null","subject":"wait_responses","body":"null","item":"1","step":"1"}}
```


Figure 13 – Example of WAIT_RESPONSES

6.3.1.2.3 UPDATE_USER_STATUS

This type of message is used by the client to notifies others clients his own state in the test. With this message all clients can know the actual state of the others clients in the room. The finality of all clients that participate in a test is communicate with them when the client needs it. It is only necessary in the test mode (Version 2) This message has a functionality a bit complex, because it depends on the value of the parameter **TO**. In other section we had seen that the parameter **TO** defines who is the receptor of the information that defines the message. In this message has the same functionality, but in this case only share the actual state of a client. The message has the next parameter setter:

- **ID**: It is the mark of the time that represents the message.
- **CLIENT_ID:** It is the number that identifies the user in the data base.
- **CLIENT_NAME:** It is the name of the client.
- **SESSION:** It is the room where the user is login.
- **CHANNEL:** It is the channel where the server processes this message (“**Cmd channel**”, “**Data channel**”), in this case the value id “**Cmd channel**”.
- **TYPE:** It is the type that defines the action (**WAIT_RESPONSES**, **ADD_RESPONSE**, **UPDATE_USER_STATUS**), in this case is **UPDATE_USER_STATUS.**
- **IP:** It is the actual IP address of the client.
- **ERROR:** null.
- **TO:** It is the receptor of the message.
- **REPLY_TO:** null.
- **SUBJECT:** null.
- **BODY:** null.
- **ITEM:** It is an integer that represents the number of question that a client is answering.

- **STEP:** It represents the state of the answer in a question (individual, collaborative, final). It is represented by an integer between 1 and 3.

In summary this message is used to notify other clients what is his actual state during a test, this message can notify other client or a group.


```
Se ha enviado al servidor --> {"message":{"id":"21112015172312234","client_id":"111636","client_name":"jesus2","session":"IDASIGNATURA#22","channel":"Cmd channel","ip":"127.0.0.1","error":"null","type":"1004","to":"others","reply_to":"null","subject":"1004","body":"null","item":"1","step":"2"}}
```


Figure 14 – Example UPDATE_USER_STATUS

6.3.1.2.4 ADD_RESPONSE

This kind of message is utilized to send message between clients. It has a functionality very similar to the message of **UPDATE_USER_STATUS**. The client needs to build a message with the parameter by default how we defined in the paragraph **6.3.2**. In this case we are going to use other parameters of the message. For the version one we will use **SUBJECT**, **TO**, **TYPE** and **BODY.** For the version two we will use **SUBJECT**, **TO**, **TYPE**, **BODY, ITEM** and **STEP**. Now we are going to analyze this two versions of the message.

In the version one, the message of **ADD_RESPONSES** has not restrictions in the receptor, but in the case of the version two, we need to know the state of the user that sends the message because a client only will can see the message related with our actual question while he is in the state of answer collaborative. Now we can see an image that represents the initialization of a message for the version one.

```
var cmd = initMsn();
cmd.channel="Data channel";
cmd.ip = dirIP;
cmd.type = ADD_RESPONSE;
cmd.to = "others";
cmd.body = texto;
cmd.subject = "answer";
cmd.id = nuevaIdMensaje();
```

Figure 15 – Initialization of a message

Now we can see an example of the two possible initialization of this kind of message.

Version one

- **ID**: It is the mark of the time that represents the message.
- **CLIENT_ID:** It is the number that identifies the user in the data base.
- **CLIENT_NAME:** It is the name of the client.
- **SESSION:** It is the room where the user is login.
- **CHANNEL:** In this case the value is "**Data channel**".
- **TYPE:** In this case **ADD_RESPONSES.**
- **IP:** It is the actual IP address of the client.
- **ERROR:** null.

- **TO:** In this message by default we are going to use "**others**"
- **REPLY_TO:** null.
- **SUBJECT:** In this case the value is "**answer**".
- **BODY:** In this parameter we are going to write the text what the client wants to send to the others client.
- **ITEM:** In this version the value of this parameter is not necessary, the value will be null.
- **STEP:** In this version the value of this parameter is not necessary, the value will be null.

Version two

- **ID**: It is the mark of the time that represents the message.
- **CLIENT_ID:** It is the number that identifies the user in the data base.
- **CLIENT_NAME:** It is the name of the client.
- **SESSION:** It is the room where the user is login.
- **CHANNEL:** In this case the value is "**Data channel**".
- **TYPE:** In this case **ADD_RESPONSES.**
- **IP:** It is the actual IP address of the client.
- **ERROR:** null.
- **TO:** In this message by default we are going to use "**others**".
- **REPLY_TO:** null.
- **SUBJECT:** In this case the value is "**answer**".
- **BODY:** In this parameter we are going to write the text what the client wants to send to the others client.
- **ITEM:** In this version we need to kwon the context of this message, to render it when it is necessary.
- **STEP:** In this version the client only can see the message in the step 2 (collaborative), and here will be this value by default.

6.3.1.3 Message Container

Until now, we had assumed that the client and the server only are able to send and receive a message, this is true, but it can introduce a problem: a big latency when a client or a server need to do complex actions, or when the server has a lot of messages sent to the clients in a room. Imagine that the room has 5 users that say "hello" in a message and "How are you?", this produces 10 messages that the server will need to send to the client. The server will need to send 40 messages in total and this can be a bit tedious, more than if we only can send a message. With this problem arises the next type of message, this message is able to container multiple messages to optimize the model to send and receive multiple messages in only one communication. We know that the server has a queue of messages for every client in the room, when in this queue the server has multiple messages to send, the server will use this kind of message. The functionality is the same that others messages, imagine that a client send a message of **WAIT_RESPONSES** and the server has five messages waiting for being sent to the client from other clients in the room, for example three **UPDATE_USER_STATUS** and two **ADD_RESPONSES**. In this case, the server built a message container that is a representation of an array of messages in **JSON**. When the server finalizes to build the message

container he sends to the client this message and the client processes one by one all the messages that he has in the message container. Actually, the server and the client have the same functionality: they only can send one message by communication but now they can pack a group of messages in one message.

Se ha recibido del servidor --> {"messageContainer":{"messages":[{"message":{"to":"111636","body":"","reply_to":"","error":null,"subject":"","type":1001,"ip":null,"id":null,"session":"IDASIGNATURA#22","item":-1,"client_id":"111635","client_name":"jesus1","channel":"Cmd+channel","step":-1}},{"message":{"to":"111636","body":"null","reply_to":"null","error":null,"subject":"wait_responses","type":1003,"ip":null,"id":"21112015172217271","session":"IDASIGNATURA#22","item":1,"client_id":"111635","client_name":"jesus1","channel":"Cmd+channel","step":1}}]}} outputlog.js:25

Figure 16 – Example message container

In the previous sections, we had analyzed all type of messages that will use our system, now in this section we will speak about the protocol that are involved in the functionality of our system. To explain all of our protocols we will use two clients (C_1, C_2) and a server (S). In all interactions between our two clients, the client C_1 will be who starts all conversations with the server. In all protocols, we will imagine that the process of the information is good and the network has no problem with the transmission of the information.

6.3.2 Protocols

6.3.2.1 Initialize session in a room.

Here we are going to see how is the full interaction when two clients enter in a room. Our client C_1 is who sends the first message. C_1 sends a command of **INIT** to our server, the server initializes all structures that he needs to manage the room and the user. Finally, the server responds to the client with a **ACK** to confirm that he has received the command and he has processed without problem the command of **INIT**. Now, in the next step, the client starts to send a message of **WAIT_RESPONSES** and let the user write in the chat room, for telling the server that he is waiting for information. Now we will imagine that the client C_2 enters to the room and executes the same code that the other client. First, it sends a **INIT** command and the server confirms it with a **ACK**, now the client C_2 starts to send the message of **WAIT_RESPONSES** to indicate that he is ready to start receiving messages. Now the client C_1 sends a message of **WAIT_RESPONSES** and the server sends a message container with two messages. The first message is a **ADD_USER** and the second is a **WAIT_RESPONSES**. Those two messages have the same origin that is C_2. In the next image we can see these messages.

Se ha recibido del servidor --> {"messageContainer":{"messages":[{"message":{"to":"111636","body":"","reply_to":"","error":null,"subject":"","type":1001,"ip":null,"id":null,"session":"IDASIGNATURA#22","item":-1,"client_id":"111635","client_name":"jesus1","channel":"Cmd+channel","step":-1}},{"message":{"to":"111636","body":"null","reply_to":"null","error":null,"subject":"wait_responses","type":1003,"ip":null,"id":"21112015172217271","session":"IDASIGNATURA#22","item":1,"client_id":"111635","client_name":"jesus1","channel":"Cmd+channel","step":1}}]}} outputlog.js:25

Figure 17 – Example ADD_USER and WAIT_RESPONSES

In the two messages, we can see that the receptor is the same in our case C_1 and the same origin C_2. These two messages were created when C_2 sends the message of INIT. Now the next step for the client C_1 is to parse the first command and add the user to a local list with the users in the room. When the client C_1 finish to parse and executes the actions necessaries for those messages, C_1 sends a message of **ADD_USER** and in the parameter **TO** the value now will be the client

id of C_2. In our example 111635, the server confirms the reception with a **ACK** and the client C_1 renders the new list with the client C_2. Now when the server receives a message of **WAIT_RESPONSES** from C_2, the server sends to him the message of **ADD_RESPONSE** of C_1. In the next figure, we can see a diagram that represents all of this process.

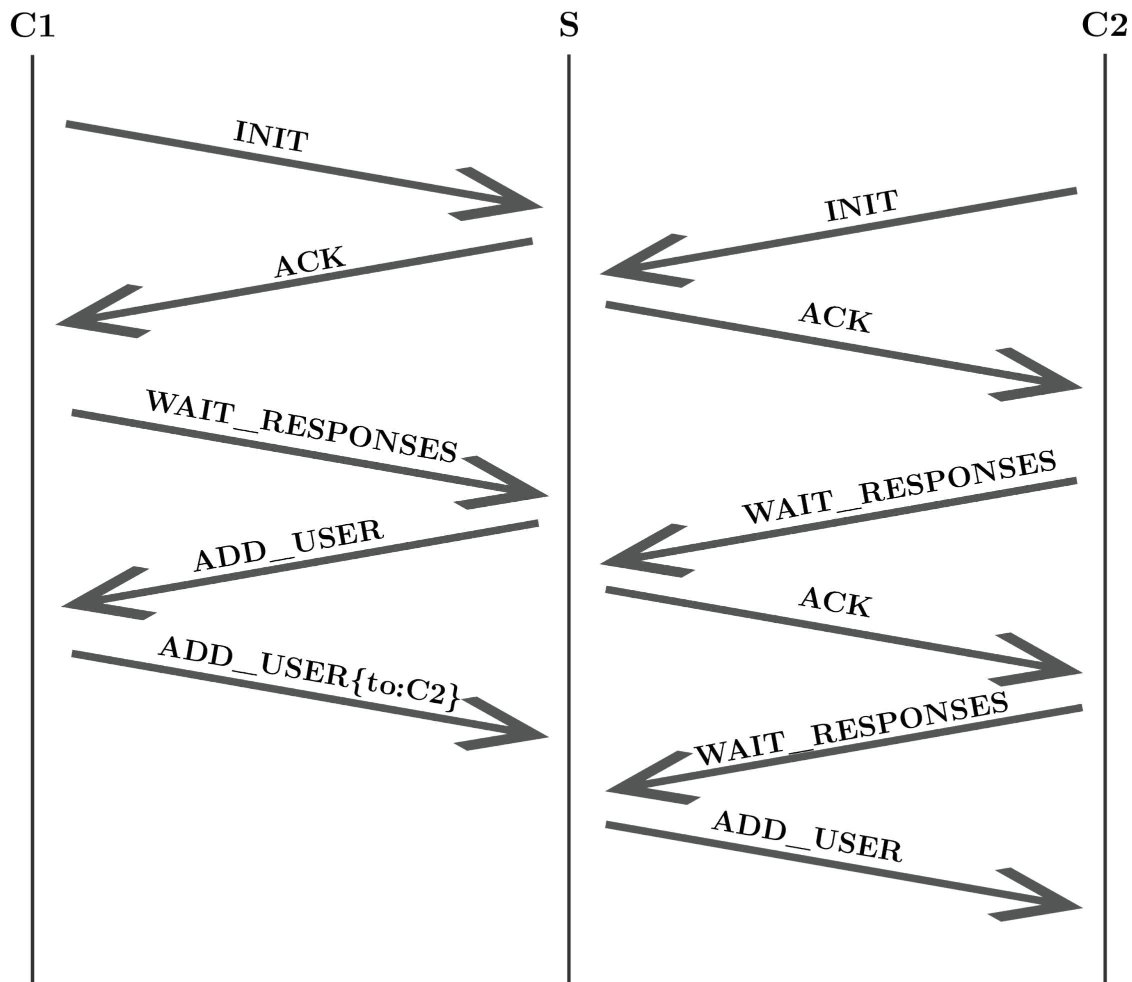

Figure 18 – Example login in a room with tow user

6.3.2.2 Leave a room.

A client leaves a room in the chat when he closes the browser or when he is isolated for a failure in the network. When the network isolates a node, the server notifies to the other client with a message of **REMOVE_USER**. The server knows that it has occurred when a client does not do communications with him. In the case that the client closes the browser or surfing to other location in the web, the client launches the event of unload. The client (JavaScript) catches the event and sends a command of **END** to the server for notifying that he leaves the room and the server confirms it with a **ACK**. When the other clients will do a communication with the server with a message of **WAIT_RESPONSES** the server will send them a message of **REMOVE_USER** and in the parameter **TO** will put the **ID** of the receptor of the message. The clients will remove from their lists of users the user that left the room. The next image illustrates this process.

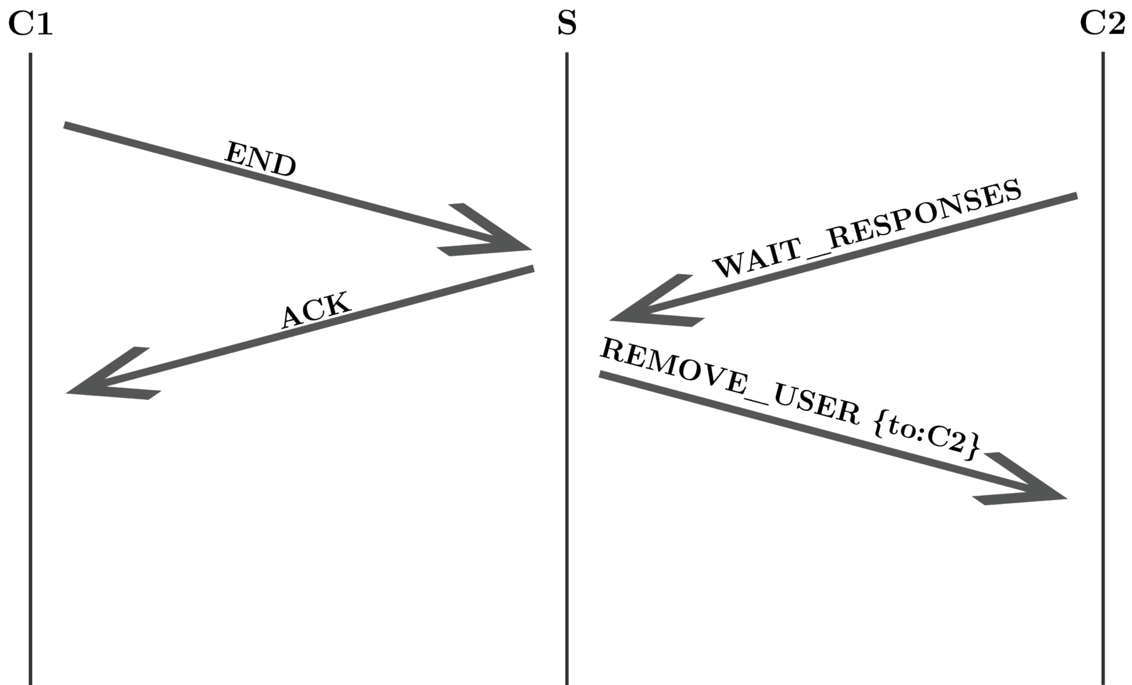

Figure 19 – Example end session

6.3.2.3 Send message between two clients.

This process is the basic protocol to send and receive information, this protocol is enabled when the client sends a text to other client. The protocol is very simple because the server is only a bridge between the client, he only need replicate the message that the client sent. Now we will to imagine that our client C_1 wants to send a message that in the **BODY** has a text that says "Hello world". In this case C_1 creates a message of **ADD_RESPONSE** that in the parameter **BODY** will have "Hello world", in the parameter **SUBJECT** the client will put "answer" and for the parameter **TO** the client is going to put the value of "other", with the finality that the server will send this message to all clients that are in the room except C_1. Now our other client C_2 sends a message of **WAIT_RESPONSES** and the server responds to this with a message of **ADD_RESPONSE** what in the parameter **TO** has the **CLIENT_ID** of C_2 , in this case, 111635. The next figure represents this process.

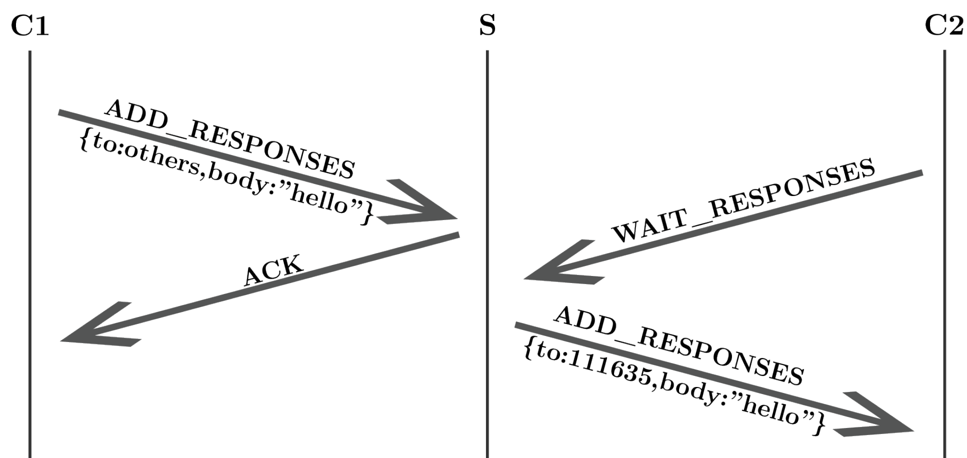

Figure 20 – Example send message in a room

6.3.2.4 *Answer a question in a test.*

This process is a bit more complex that the others. To explain it, first we need to know how is the process of **UPDATE_USER_STATUS**. The process of **UPDATE_USER_STATUS** is like the process of send a message to other client. Basically the client C_1 creates a message of **UPDATE_USER_STATUS** and in the parameter **TO** the client sets the value of "other", to notify the others clients in the room, and the parameters item and step are setting with the actual state of the client. When the server receives the message, it sends a **ACK** to confirm the reception of the message to the client C_1. Now the server replicates the message for all clients involved in the room, and the server will send this message to the clients when they will connect with the server for receiving this information (**WAIT_RESPONSES**).

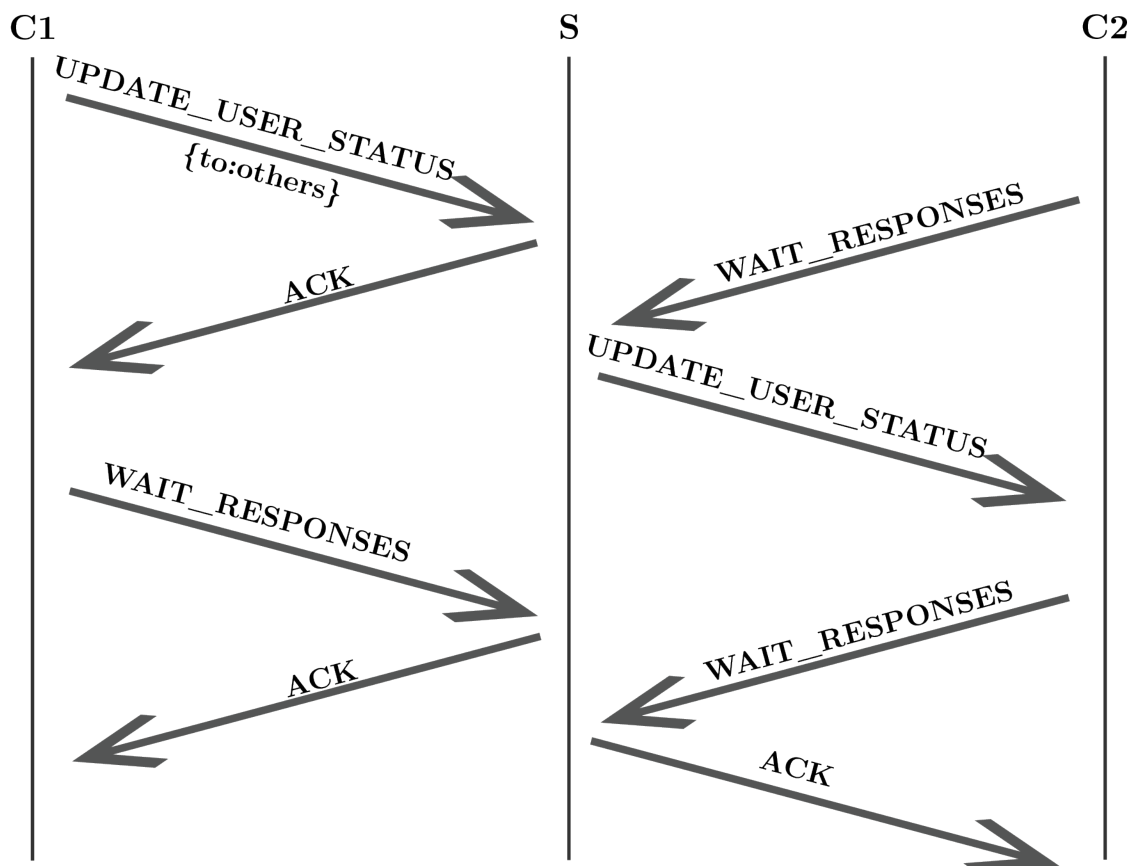

Figure 21 – Example update user status

Now we know how is the functionality of **UPDATE_USER_STATUS**, we can speak about all the process the clients and the server are doing to manage a question in the collaborative mode. We are going to imagine that our two clients are in a common room and enter in a test in collaborative mode. The first step is that our two clients will initialize session in the new room, now we will have a room where only will have two clients, C_1 and C_2. Now our clients see the first question and they do not use the chat because they are in the phase individual. When our clients will answer the question automatically the iframe where they see the question execute a function that updates the information of the client, setting the step of the client in the phase collaborative. Now our clients send to the server a message of **UPDATE_USER_STATUS** to notify the other what is his actual state and activate the chat that they can use to communicate

and discuss about his response. Now they need to answer the same question, with the information they have. In this moment, when the clients answer the question, the iframe where the question is, will execute a function to update the state of the clients. When the clients set the new state, they disable the chat and send to the server a message of **UPDATE_USER_STATUS**. Now our clients need to answer the same question they found in the step of individual and collaborative, the name of this step is “final”. When the clients will finish the question, the iframe that contains the question executes the method to update the state of the clients and the clients notify the server about it with a message of **UPDATE_USER_STATUS**. Finally, the iframe of test presents to the clients other question and all process is repeated for each new question. The next graphic show how is all of this process.

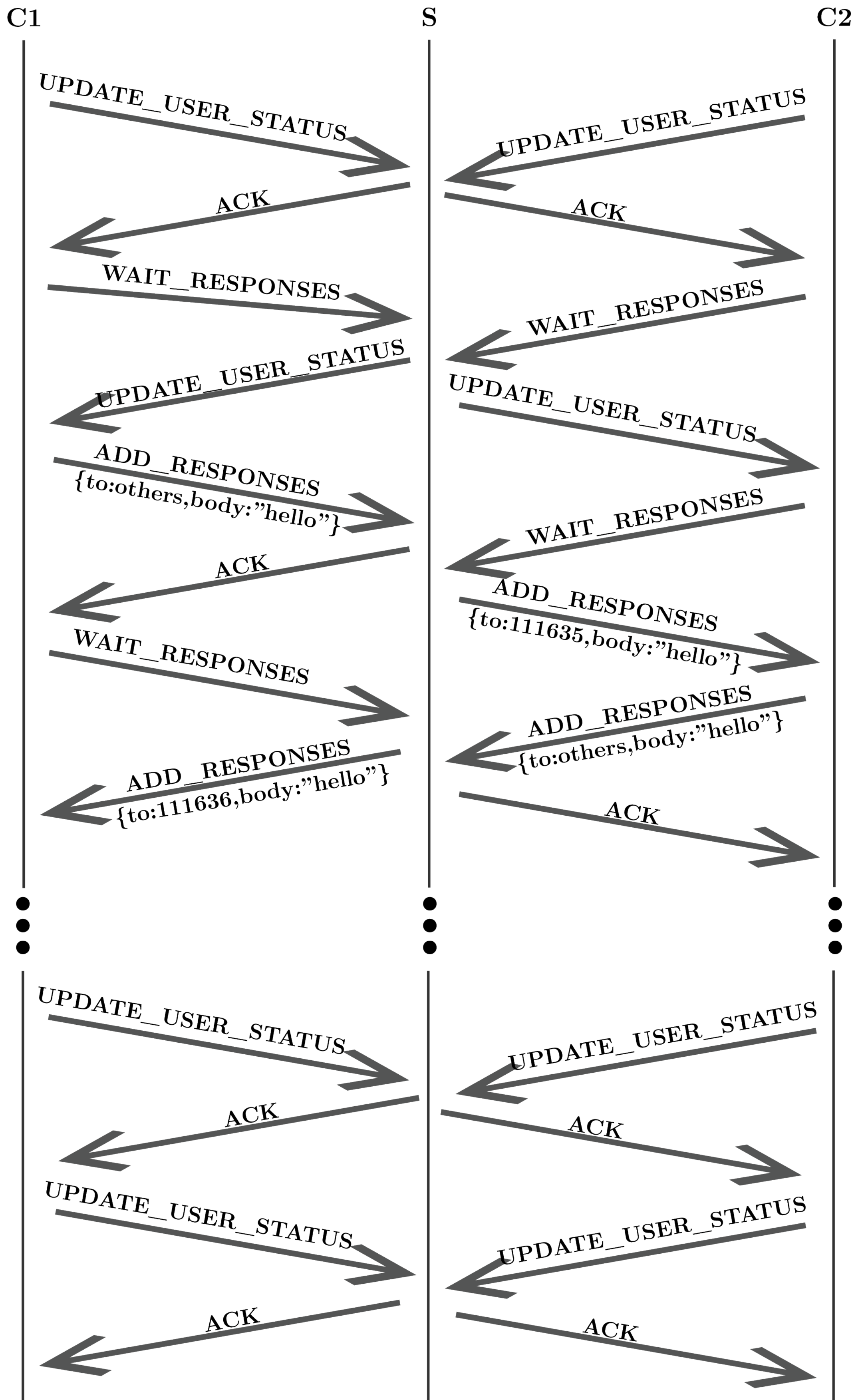

Figure 22 – Answer a question

7 Implementation.

Now, in this section, we are going to analyze all data structures and the object we need in order to solve the problem. In the first section we will analyze the object that we need to implement and in the other section we will analyze the structures that we are going to use.

7.1 Objects

In our client, we will define three main objects to manage all information. First, we will see the user, then the command, and finally the message.

7.1.1 User

A user is an object that represents all information that the client needs to communicate with him and the information that the client needs to manage it. The actual implementation has the next parameters:

- **Name**: It is the name of the client.
- **ID**: This represent the client id that assign SIETTE to the user when it is created.
- **ListIDs**: This is a list that contains all IDs of the messages that receive the client because, in several moments, the client can receive the same message in two different moments. An example of this is when the client has a problem with the network.
- **Item**: This parameter represents the number of the actual question what a user have to resolve.
- **Step**: With this parameter we can know the step where is the user.

```
var User = function() {
    this.name;
    this.id;
    this.item;
    this.step;
    this.listIds = [];
};
```

Figure 23 – User object

In our code we will have a variable that is a list of users. We will use this structure to manage all user in a room. For this structure we will need a method to search a user, other method to remove him from the list and a function to update his information.

7.1.2 Command

In the section 6.3.1 we defined the object command, but now we are going to see a more technical definition, for example the method that we need to implement to serialize to JSON the object. The basic information what we are going to save in an object command is:

- **ID**: It is the timestamp of the generation of the command.
- **Client_id**: This represents the client id that SIETTE assigns to the user when it is created.
- **Client_name**: It is the name of the client.
- **Session**: It identifies the actual room where the user is login.
- **Channel**: It identifies the channel when we are going to send the message, not in use now.
- **Ip**: It is the IP address.
- **Error**: It is a string that can contain information about an error, normally is used with the type NACK.
- **Type**: It identifies the action that we want to send to the server. It will have an integer value. INIT → 100, END→ 199, ACK → 14 and NACK → 15.

```
var Command = function(obj){
    this.id=null;
    this.client_id=null;
    this.client_name=null;
    this.session=null;
    this.channel=null;
    this.ip=null;
    this.error=null;
    this.type=null;
```

Figure 24 – Command object

This object will have multiples methods. We are going to write down a list of the methods and define its functionality.

- **get_JSON()**: This method has not parameters in the input and the return of it is a JSON representation of the object. This representation in JSON will be needed to send a command to the server.
- **toString()**: This is a method that will has not parameters in the input, and his return value is a representation of the object in string.
- **getAsString()**: This method has no inputs and the return is a String that gives a representation of the type the command.

7.1.3 Message

The message will have a structure very similar to the command. Actually, it is an extension of it. In the section 6.3.2 we analyzed how is a message and now we are going to analyze how is the real implementation and the method that we will use with it. A message has the next parameters:

- **ID:** It is a unique number that identifies the message, this number is a concatenation of the timestamp, we need to concatenate day, month, year, hour, minutes, seconds and milliseconds. 21112015172312234 →21/11/2015 17:23:12:234
- **CLIENT_ID:** It is the number (integer value) that identifies the user in the data base.
- **CLIENT_NAME:** It is a string with the name of the user.
- **SESSION:** The value of this parameter is a String that represents the name of the room where a user is logged in.
- **CHANNEL:** It is a String that represents one of two channels we can use to communicate information.

- **TYPE:** Is is an integer value that represents a message that we analyzed in the pasts sections
- **IP:** It is a String that represents the IP address of the client.
- **ERROR:** It is used to send error information of a process, the value of it is a String that has a character of "+" between the words.
- **TO:** It is a String value that represents the receptor of the message
- **REPLY_TO:** It is used to send a copy of the message to another user different or not of the users specified in the parameter **TO**.
- **SUBJECT:** It is a String value that expresses the type of the communication.

```
this.id = null;
this.client_id = null;
this.client_name = null;
this.session = null;
this.channel = null;
this.type=null;
this.ip = null;
this.error = null;
this.to = null;
this.reply_to = null;
this.subject = null;
this.body = null;
this.item = ITEM_NULL;
this.step = STEP_NULL;
```

Figure 25 – Message object

- **BODY:** It is a String value that will have the text what a client wants to send.
- **ITEM:** It is the number of question that a student is answering now in a test.
- **STEP:** It defines the state of the answering (Individual, collaborative, final). The value of it is a number between 1 and 3.

This object will have multiples methods. We are going to write down a list of the methods and define its functionality.

- **get_JSON()**: This method has no parameters in the input and the return of it is a JSON representation of the object, this is the representation that we will need to send the message to the server.
- **toString()**: This is a method what will have no parameters in the input, and his return value is a representation of the object in string.

7.2 Data structures

We are going to need a structure where the message will queue, this structure will be able to store all messages and commands the client will generate. This structure is not a real queue because it depends on the moment we will need to get an object in the center of the queue or delete it. Therefore, we are going to create our implementation of queue. For simplifying the implementation, we will use an array because **JavaScript** has multiples operations over arrays and we are not going to create a method to supply it. Our structure is only a layer over an array to simulate the regular operations in a queue but using a list. Our queue will have multiples method, summarized in the next list.

- **add(object)**: This function has an input value that is an object. The function adds an element at the end of the list.
- **get(integer)**: This function has an integer value in the input that represents the position of the object we need. The return of the function is the object if it exists.

- **getDel(integer)**: This function has an integer value in the input that represents the position of the object we need. The return of the function is the object if it exists. When the object is returned, the element is deleted of the list.
- **removeFirst()**: This function has not values in the input. The functionality is to remove the first element in the list.
- **getFirst()**:This function has not values in the input. The return is the first element in the list.
- **getDelFirst()**:This function has not values in the input. The return is the first element in the list, when the object is returned the element is deleted.
- **isEmpty()**:This function has not values in the input. The function returns a Boolean, this Boolean notifies if the list is empty.
- **size()**:This function has not values in the input. The function returns an integer that expresses the length of the list.

7.3 Files and methods

Now, in this section, we are going to see some specifications and some methods that we can use to build news functionalities. Now we are going to speak about how the distribution of our files is. Now the system uses in total 26 files distributed in 3 folders: css, fonts and js. In the css folder, we can see 5 files, all of them, are only style files. Everything with the pattern of bootstrap* belong to the framework of **Bootstrap**, in total, there are 4 files. The remaining file is our file to give specifics styles to the elements in our **HTML**. In the folder of fonts we can see the files of fonts that Bootstrap uses (6 files). Finally, in the folder of js we can see all JavaScript files that compose our system. Here, we have multiples files. Every single file implements a specific functionality listed below.

- **Ajax.js**: In this file we are developing the method related with the functionality of ajax. We can see principally two methods that we use to send and receive information. One is used to send the normal information, usually used to clean the queue of message output. The other function is utilized to send urgent information. A good example can be the message of **END**.
- **Cola.js**: In this file we can see the implementation of our queue described in the section 7.2.
- **Command.js**: In this file we can see the full implementation of the object Command described in the section 7.1.2.
- **Constants.js**: In this file we can see all the constants our system uses. E.g.: the type of our messages and commands.
- **InOut.js**: In this file we can find the consumer and producer of messages.
- **Jquery.2.1.4.js** and **jQuery.js**: These files contain the framework of jQuery. The first is the version 2 and the second is the version 1. We can use the second version if we want to increase the performance but the problem is that it has less compatibility.
- **JSON.js**: We will use this file to save all function that parses our String to JSON to transfer information or receive it.

- **listadoUsuarios.js**: Here we can find all functions that we need to manage the list of the user, for example, we have functions to render user in the **HTML** or function to get user and update the information of it.
- **Main.js**: Here we have two main functions. The first is the function that initializes all variables and parameters for the version one of our system. The second is the function that initializes the version two of our system.
- **ManejoChat.js**: In this file, we find all function related to the chat, for example we can see functions to add a response to the actual room.
- **Message.js**: In this file we can see the implementation of our object message (section 7.1.3).
- **Outputlog.js**: Here we have implemented the function that manages our log. The log of the system has three states: normal, debug and alert. The first mode is the normal mode that we will use in the version of production. The alert and debug modes are profiles that produce changes in the log form, for example the mode of alert builds a message of alert that stops the code when the system produces a new line in the log. All modes have a profile that configures the frequency the client communicates with the server.
- **Serializado.js**: Here we can find the function that serializes and deserializes the object that we need to send or process.
- **TodasRespuestas.js**: In this file, we can see the function that sends a message to the server for viewing all responses of the actual question.
- **VolteadoChat.js**: In this file, we can see the function that produces the change in the orientation of our system. We have designed the system thinking in it and we have developed our chat structure like a table. If we need to change the structure, we only need to invert columns and rows.

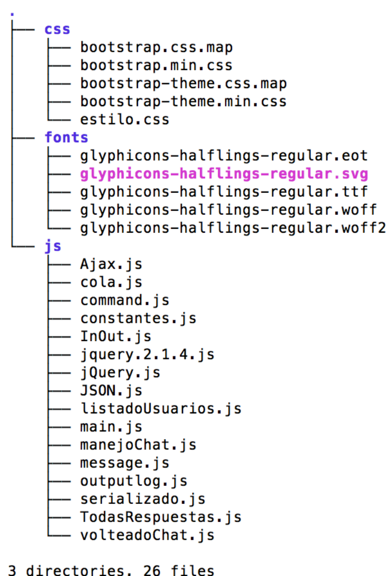

Figure 26 – Structure of folders in develop

It should be noted that this distribution of files is used in the developing mode, if we want this files in an environment of production we need to group all **JavaScript** files in one file and all

files **CSS** in another file to increase the performance. To do it we can use Grunt to provide a system to minify our code and group it.

Now we are going to see a list of methods that we can use to build new functionalities.

- **voltear()**: This method change the orientation horizontal to vertical in our system, by default the system is displayed in the horizontal mode.
- **sendInformationUrg**(object *object*): this method serializes the object in the input and send it to the server.
- **viewAllResponses**(): This method processes the event of view all answer in the version two.
- **DesSerializar**(object *object*): This method deserializes the object in the input and adds it to the queue of messages to process.
- **addUserTo**(id *integer*): This method creates a message of **ADD_USER** where the receptor is the number in the input, this number the client_id.
- **esperoRespuestas**(): This method produces a message of **WAIT_RESPONSES** and add it to the output queue.
- **prepararUpdateUserStatus**(): This method produces a message of UPDATE_USER_STATUS and add it to the output queue.
- **prepararACK**(): This method is used to produce a command of **ACK**, when the command is created the method add it to output queue.
- **prepararNACK**(): This method generates and adds a command of **NACK**.
- **initCmd**(): This method returns a command object initialized with the parameter of the local user.
- **initMsn**(): This method returns a message object initialized with the parameter of the local user.
- **logger**(text *String*): This method creates a new line in the log that is the String in the input.
- **nuevaIdMensaje**(): This method returns a String with a new id that we can use in a new message.
- **responder**(text *String*): This method sends the input text to the actual room.
- **estadoChat**(state Boolean): This method disables or enables the chat room.
- **jsSetStateLocal**(item *integer*, step *integer*): This method changes the actual state of the user when the client is in the test mode. The method update the information in the view of the user and send a message of **UPDATE_USER_STATUS** to the server.
- **updateUsersTestState**(): This method render the actual information of all user in the view of the user.
- **cargarUsuario**(name *String,* id *integer,* item *integer (optional),* step *integer (optional)*): This method add a new user to the actual list of user and render it. If we want to add a user in the version one we only need the name and the id of the client, for the version two we need to know the name, id and the actual state in the chat (item, step).

- **getParamHTML**(name *String*): This method return a String with the value of a tag param that in the parameter name has the input value (<param name='example' value='12').

8 User Interface (UI).

In this section we are going to speak about the user interface (UI). We will analyze the interface and the parts that we will need to develop for our system. Firstly, we are going to analyze the version one, for this version we will need a list with the users that are in the actual room. We also will need a section where rendering the message of the common room. Finally, we will need a section where typing our message. Furthermore, we want that the interface have two distributions, one vertical and other horizontal. With this information, we will need our interface organized in sections because we want to change the display on the fly. The first section will be something like that:

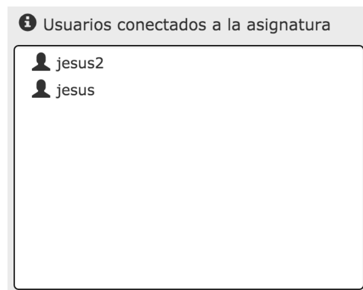

Figure 27 – List of users

In the last image, we can see that two users are logging into the room, we can appreciate that the design is very simple because we will not need to display more information in this area, but in the case of the version two this can be a bit more complex. For the section where the client writes and reads the message, we want to applicate the same design. We will need only a box with the list of the message in the room and other box to write and send the messages. In the next image, we can see those two elements.

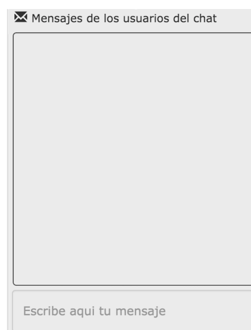

Figure 28 – Chat Room

First, in the image we can see the box where our message will be writing, and at the bottom we can see the input where the client writes the message. We can appreciate that the two boxes are disabled, this is other of our restrictions, we need that our boxes have a state of disabled and enabled. Our client can not allow a user to write in a room where he is not logged in. Finally, we are going to see the chat system in vertical and horizontal mode.

Figure 29 – Chat in vertical mode

Figure 30 – Chat in horizontal mode

Now we are going to analyze the interface of our version two. This version is a bit more complex than the version one. In this version, we will need to know the actual state of the user and we need a button to consult the response of the users in the test. In summary, we will need a label with the name of the users, a badge with the actual question, a progress bar to represent the state in the question and a button to consult the response of the client to the actual question in the state of discuss (collaborative). With those requirements, we will design the next display for the list of user.

Figure 31 – List of users in test mode

We can appreciate that now, at the bottom, there is not any chat room, but, instead, there is a button with a simple functionality. The button is enabled when the user is in the state of discuss (collaborative) and disabled in the others states (Individual and final). When the user clicks on the button, he can see the answers in the individual mode of all users that are in this question or at later question. For the chat room, we are going to use the design of the version one because we will not need to display more information. This version of the chat also has a vertical and horizontal distribution. In the next image we can see the horizontal distribution.

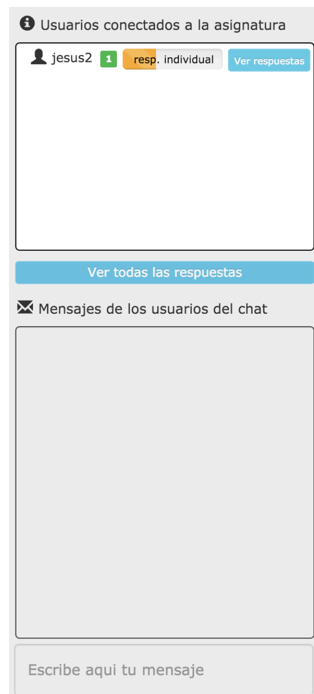

Figure 32 – Chat in test mode

9 Conclusion (english).

The whole project has been implemented in **JavaScript**, **HTML5** and **CSS3** to meet the requirement of compatibility. We have completed all of requirements of the projects and we have added a group of functions that the next developer can use to create new functionalities. This project meets the next specifications.

- ✓ The client is supported in Chrome, Firefox, Safari and Opera.
- ✓ The system works fine with the actual version of the client (Java applet) and with the server (Java Servlets).
- ✓ The client sends and receives messages to manage the chat room in real time.
- ✓ The client manages a list with users in real time.
- ✓ The client can add and remove users to a chat room.
- ✓ The chat manages the three steps for each question.
- ✓ The chat is enabled and disabled when the steps require it.
- ✓ The client can be displayed in two grids, vertical and horizontal.
- ✓ The client support multiples languages that is initialized in the **HTML** page with some tag of **<param>**.

The principal limitation of this project is to adapt the **JavaScript** client to the applet and the server, because the implementation of the applet and the servlets that manage the petitions are very complex. With the implementation of our system in **JavaScript** we have discovered some problems in the actual system, an example can be a problem of casting in the server with the String that represents the “item” in the petition to a Long object. We have solved all problem that our system has found in the actual implementation.

For future implementations we know that the system of message that use our chat is not efficient with the channel, because the client needs to send messages frequently to inform to the server that he is connected and for receiving new information. We can solve this problem with the use of an open socket, this will produce that the use of channel of the server and the client will be more efficient. For the client we can re-use the **JavaScript** client and add the functionality of an open socket with the framework of socket.io.

10 Conclusion (spanish).

Todo el proyecto se ha implementado en **JavaScript**, **HTML5** y **CSS3** para cumplir con el requisito de compatibilidad. Hemos completado todos los requerimientos de las 2 versiones necesarias y hemos añadido un grupo de funciones que el próximo desarrollador puede utilizar para crear nuevas funcionalidades. Este proyecto cumple con las siguientes especificaciones.

- ✓ El cliente está soportado en Chrome, Firefox, Safari y Opera.
- ✓ El sistema funciona bien con la versión actual del cliente (applet de Java) y con el servidor (Servlets de Java).
- ✓ El cliente envía y recibe mensajes para gestionar la sala de chat en tiempo real.

- ✓ El cliente maneja una lista con los usuarios en tiempo real.
- ✓ El cliente puede agregar y quitar usuarios a una sala de chat.
- ✓ El chat gestiona los tres pasos para cada pregunta.
- ✓ El chat está activado y desactivado cuando los pasos lo requieren.
- ✓ El cliente es capaz de ser visualizado en dos formatos, vertical y horizontal.
- ✓ El cliente soporta multiples idiomas que se inicializan en la página **HTML** con un poco de etiqueta de <**param**>.

La limitación principal de este proyecto es adaptar el cliente **JavaScript** para el applet y el servidor, debido a que la aplicación del applet y los servlets que gestionan las peticiones son muy complejos. Con la implementación de nuestro sistema en **JavaScript** hemos descubierto algunos problemas en el sistema actual, un ejemplo puede ser un problema de casteo en el servidor con la cadena que representa el elemento "item" de la petición a un objeto del tipo Long. Hemos resuelto todos los problemas que nuestro sistema ha encontrado sobre la aplicacion actual.

Para implementaciones futuras sabemos que el sistema de mensaje que utilice nuestro chat no es eficiente con el canal, ya que el cliente necesita enviar mensajes con frecuencia para informar al servidor que está conectado y para recibir nueva información. Podemos resolver este problema con el uso de socket abierto, esto producirá que el intercambio de mensajes entre servidor y el cliente sea más eficiente. Para el cliente, podemos volver a utilizar el cliente desarrollando en este projecto en **JavaScript** y añadir la funcionalidad de un socket abierto con el framework socket.io.

11 Bibliography.

- http://arstechnica.com/information-technology/2014/10/html5-specification-finalized-squabbling-over-who-writes-the-specs-continues/
- http://www.w3.org/blog/news/archives/4167
- http://www.json.org/
- http://getbootstrap.com/
- https://en.wikipedia.org/wiki/JavaScript
- https://jquery.com/
- http://socket.io/

12 Appendix one: Code

```
var NULL = 0;
/* Comandos de la clase ServerConnection. */
/** Especifica un comando de <b>abrir nueva sesiÛn</b>. */
var OPEN_SESSION = 1;
/** Especifica un comando de <b>cerrar sesiÛn existente</b>. */
var CLOSE_SESSION = 2;
/** Especifica un comando de <b>obtener sesiÛn existente</b>. */
var GET_SESSION = 3;
/* Comandos de la clase Session. */
/** Especifica un comando de <b>annadir cliente a sesiÛn</b>. */
var ADD_CLIENT = 4;
/** Especifica un comando de <b>eliminar cliente de sesiÛn</b>. */
var REMOVE_CLIENT = 5;
/** Especifica un comando de <b>crear un nuevo canal en una sesiÛn</b>. */
var CREATE_CHANNEL = 6;
/** Especifica un comando de <b>eliminar un canal existente de una sesiÛn</b>. */
var REMOVE_CHANNEL = 7;
/** Especifica un comando de <b>obtener un canal existente de una sesiÛn</b>. */
var GET_CHANNEL = 8;
/** Especifica un comando de <b>enlazar un cliente con un observador y un canal</b>. */
var JOIN = 9;
/** Especifica un comando de <b>desenlazar un cliente de un observador y un canal</b>. */
var LEAVE = 10;

/* Comandos de la clase Channel. */

/** Especifica un comando de <b>enviar un mensaje</b>. */
var SEND = 11;

/* Comandos de la clase Consumer. */

/** Especifica un comando de <b>recibir mensajes pendientes del servidor</b>. */
var RECEIVE = 12;

/*
 * Comando del servidor: avisa a los usuarios de una sesiÛn cuando uno de ellos
 * ha perdido la conexiÛn.
 */

/** especifica un comando de <b>desconexiÛn involuntario de un cliente</b>. */
var CLIENT_DISCONNECTED = 13;
```

```
/* Comandos de confirmaciÛn. */

/** Especifica un comando de <b>reconocimiento</b>. */
var ACK = 14;
/** Especifica un comando de <b>no reconocimiento</b>. */
var NACK = 15;

/** Comandos para pedir la lista de usuarios registrados en un canal/sesion */
var CLIENTLIST = 200;

var ITEM_NULL=-1;
var STEP_NULL=-1;
var INIT = 100;
var END =  199;
var ADD_USER= 1001;
var REMOVE_USER = 1002;
var WAIT_RESPONSES= 1003;
var UPDATE_USER_STATUS= 1004;
var ADD_RESPONSE= 1005;
var END_TEST= 1006;

var client_name = "Root";
var client_id;
var item = -1;
var step = -1;
var session;
var URL = "SalaServlet";
var dirIP = "127.0.0.1";

var contReenvio = 0;
var ultMensaje = '';


// vars view responses
var functionViewResponses;
var functionViewAllResponses;

var queueOUT; // It is the queue that manage all messages that send the client to the server
var queueIN;// It is a queue for receive message from the server
var queueChat;// Only for messages(A message need to be confirmed by the server to get out the
queue)
var queueMessages;//Only for save the message in the test mode
```

```
var queueUsers;

//Vars language chat
var tituloChat;
var estadoUser;
var listUser;
var tituloChatUser;
var respondera;
var todos;
var todasRespuestas;
var placeholderchat;
var placeholderescritura;
var viewResp;
var viewAllResp;
//Text of state collaborative individual, collaborative, final
var textState1;
var textState2;
var textState3;

//Vars for the mode of init
var debug = 0;
var alert = 0;

//Variables para volteado del chat y para el mantenimiento de la informacion para el volteado en vivo
var verticalMode = 1;
var chatHorizontal = '';
var chatVertical = '';
var respuestas = '';
var listaUsers = '';
var desplegableUsers = '';

//variables generables
var timeOut = 3000;
var timeOutActual;

//Vars test mode
var testMode= false;


var idMensaje = 0;
//Funcion de inicializacion de la pagina
function inicializacion() {
```

```
    logger('Disable test mode');
    //disable test mode
    testMode = false;
    //init variables of the chat (translation, client_ids ....)
    logger('loading params');
    timeOutActual = timeOut;
    URL = getParamHTML('URL');
    tituloChat = getParamHTML('term5');
    estadoUser = getParamHTML('term0');
    listUser = getParamHTML('term9');
    tituloChatUser = getParamHTML('term7');
    respondera = getParamHTML('term12');
    todos = getParamHTML('term26');
    todasRespuestas = getParamHTML('term6');
    placeholderchat = getParamHTML('term27');
    placeholderescritura = getParamHTML('term28');
    client_name = getParamHTML('username');
    client_id = getParamHTML('userid');
    session = getParamHTML('session');
    inicLanguaje();
    initQueues();
    //loading data of url
    logger('loaing url params');
    leeUrl();
    logger('init vars');
    //init the item and step to null
    item = ITEM_NULL;
    step = STEP_NULL;

    var us = new User();
    us.name = client_name;
    us.id = client_id;
    if (verticalMode == 1)
        voltear();
    logger('loading user');
    cargarUsuario(us.name,us.id);
    estadoChat(false);
    estadoEscritura(false);
    logger('creating init command');
    initSesionServer();
    stateAllResp(false);
    logger('end');
    setInterval('limpiarSalida()', timeOut);
}
```

```
function inicializacionTestMode() {
    logger('Enable test mode');
    // init test mode
    testMode = true;
    //init variables of the chat (translation, client_ids ....)
    logger('loading params');
    timeOutActual = timeOut;
    URL = getParamHTML('URL');
    tituloChat = getParamHTML('term5');
    estadoUser = getParamHTML('term0');
    listUser = getParamHTML('term9');
    tituloChatUser = getParamHTML('term7');
    respondera = getParamHTML('term12');
    todos = getParamHTML('term26');
//     todasRespuestas = getParamHTML('term6');
    placeholderchat = getParamHTML('term27');
    placeholderescritura = getParamHTML('term28');
    client_name = getParamHTML('username');
    client_id = getParamHTML('userid');
    textState1 = getParamHTML('term1');
    textState2 = getParamHTML('term2');
    textState3 = getParamHTML('term3');
    viewResp = getParamHTML('term24');
    viewAllResp = getParamHTML('term25');
    functionViewResponses= getParamHTML('jsConsultarRespuestas');
    functionViewAllResponses= getParamHTML('jsConsultarTodasRespuestas');
    inicLanguaje();
    initQueues();
    //init vars of states in the test mode
    logger('init vars');
    item = 1;
    step = 0;
    //loading data of url
    logger('loaing url params');
    leeUrl();
    session = getParamHTML('session');
    var us = new User();
    us.name = client_name;
    us.id = client_id;
    us.item = item;
    us.step = step;
    if (verticalMode == 1)
        voltear();
```

```
    logger('loading user');
    cargarUsuario(us.name,us.id,us.item,us.step);
    estadoChat(false);
    estadoEscritura(false);
    logger('creating init command');
    initSesionServer();
    stateAllResp(false);
    logger('end');
    $('#allResp').click(function() {
        viewAllResponses();
    });
    setInterval('limpiarSalida()', timeOut);
}


function getParamHTML(name){
    return $('param[name='+name+']')[0].value;
}

function stateAllResp(state){
    if (state){
        $('#allResp').removeAttr('disabled');
    }else{
        $('#allResp').attr('disabled',true);
    }
}

function initSesionServer() {
    var cmd = initCmd();
    cmd.ip = '127.0.0.1';
    cmd.type = INIT;
    addQueueOUT(cmd);
    //recibeJson();
}

//read parameter get
function getGET() {
    // catch the url
    var loc = document.location.href;
    return getParam(loc, "get");
}

function getParam(loc, tipo) {
    //Parsing the URL
```

```
    var GET;
    var enc = false;
    if (tipo == "get") {
        if (loc.indexOf('?') > 0) {
            enc = true;
            // split the string by ?
            var getString = loc.split('?')[1];
            // create an array with key=value
            GET = getString.split('&');
        }
    } else if (tipo == "post") {
        GET = loc.split('&');
        enc = true;
    }
    var get = {};
    if (enc) {
        // Analyze the array
        for (var i = 0, l = GET.length; i < l; i++) {
            var tmp = GET[i].split('=');
            get[tmp[0]] = unescape(decodeURI(tmp[1]));
        }
    }
    return get;
}

function leeUrl() {
    // Analyze the values catch in the URL
    var valores = getGET();
    if (valores)
    {
        for (var index in valores)
        {
            if (index == "debug") {
                debug = valores[index];
                if (debug == 1){
                    logger("Debug mode enable");
                }
                else{
                    logger("Debug mode disables");
                }
                timeOut = 6000;
            } else if (index == "alert") {
                alerta = valores[index];
```

```
                if (alert == 1)
                    logger("Alert mode enable");
                else
                    logger("Alert mode disabled");
                timeOut = 15000;
            } else if (index == "debugAvanzado") {
                debugAvanzado = true;
            } else if (index == "volteado") {
                verticalMode = valores[index];
                if (verticalMode == 1)
                    logger("Chat in vertical mode");
                else
                    logger("Chat in horizontal mode");
            } else if (index == "usuario") {
                client_name = valores[index];
            } else if (index == "id") {
                client_id = valores[index];
            } else if (index == "session") {
                session = valores[index];
            }
        }
    } else {
        // no se ha recibido ningun parametro por GET
        logger("The URL not has valid parameters");
    }
}

var User = function() {
    this.name;
    this.id;
    this.item;
    this.step;
    this.listIds = [];
};


var state1='33.333333%';
var state2='66.666666%';
var state3='100%';
var classbefore = "label-danger";
var classEquals = "label-success";
var classAfter = "label-info";
var classProgressBar = 'progress-bar progress-bar-warning progress-bar-striped';
```

```
// function to set the status to local user
function jsSetStateLocal(item_set, step_set){
    var msn = prepararUpdateUserStatus();
    msn.item = item_set;
    msn.step = step_set;
    item = item_set;
    step = step_set;
    if (step==1){
        estadoChat(true);
        estadoEscritura(true);
        stateAllResp(true);
        cleanQueueMessages();
    }else{
        estadoChat(false);
        estadoEscritura(false);
        stateAllResp(false);
    }
    updateUsersTestState();
    updateUserTest(client_id, client_name, item, step);
    addQueueOUT(prepararUpdateUserStatus());
}

//function to procces a set status
function jsSetStateUser(client_id, item, step){
    var user = getUser(client_id);
    user.item = item;
    user.step = step;
    updateUserTest(client_id, user.name, item, step);
    updateUsersTestState();
}

function updateUsersTestState(){
    for (var i = 0; i < queueUsers.size();i++){
        var user = queueUsers.get(i);
        updateUserTest(user.id, user.name, user.item, user.step);
    }
}

function updateUserTest(client_id, client_name, item, step){
    var user = getUserById(client_id);
    user.item = item;
    user.step = step;
    updateItem(client_name, client_id, item);
```

```
    updateStep(client_id,client_name,step);
    updateButtonViewResp(client_name, client_id, step);
}

function updateStep(client_id, client_name, step_aux){
    var bar = $('#'+client_name+'\\:'+client_id)[0];
    var newBar = getProgressBar(client_id, client_name,step_aux==-1 ? 1: step_aux);
    bar.outerHTML = newBar;
}

function updateItem(name, id, item_aux){
    $('#item\\:'+name+'\\:'+id)[0].innerHTML = item_aux != null ? item_aux : 1;
    var classColor = '';
    if (item > item_aux){
        classColor = classbefore;
    }else if (item == item_aux){
        classColor = classEquals;
    }else{
        classColor = classAfter;
    }
    $('#item\\:'+name+'\\:'+id)[0].setAttribute('class','');
    $($('#item\\:'+name+'\\:'+id)[0]).addClass('label').addClass(classColor);
}

function updateButtonViewResp(name,id,step){
    if (step==1){
        $('#btn\\:'+id).removeAttr('disabled');
        $('#btn\\:'+id).attr('onclick',functionViewResponses+'('+client_id+')');

    }else{
        $('#btn\\:'+id).attr('disabled',true);
        $('#btn\\:'+id).removeAttr('onclick');
    }
}

function getProgressBar(client_id, client_name, step){
    var textStep;
    var width;
    if (step == 0){
        textStep = textState1;
        width = state1;
    }else if (step == 1){
        textStep = textState2;
        width = state2;
```

```
    }else{
        textStep = textState3;
        width = state3;
    }
    return '<div style="width: 39%;padding: 0px 5px;" id="'+client_name+':'+client_id+'"><div class="progress" style="min-width:95;max-width:95;width:95;"><div data-id="'+client_name+':'+client_id+'" class="'+classProgressBar+'" role="progressbar" aria-valuenow="33.333" aria-valuemin="0" aria-valuemax="100"
style="width:'+width+'"><span>'+textStep+'</span></div></div></div>';
}

function getBadge(client_id, client_name, item_aux){
    var classColor = '';
    if (item > item_aux){
        classColor = classbefore;
    }else if (item == item_aux){
        classColor = classEquals;
    }else{
        classColor = classAfter;
    }
    return '<span id="step:'+client_id+'" style="padding-top: 5px;"><span id="item:'+client_name+':'+client_id+'" class="item label '+classColor+'">'+item_aux+'</span></span>';
}

function getButtonViewResponses(client_id, client_name, step){
    if (step == 1){
        return '<button style="font-size: 9px;" id="btn:'+client_id+'" class="btn btn-info btn-xs btn-view-resp"
onclick="'+functionViewResponses+'('+client_id+')">'+viewResp+'</button>';
    }else{
        return '<button style="font-size: 9px;" id="btn:'+client_id+'" class="btn btn-info btn-xs btn-view-resp" disabled="true">'+viewResp+'</button>';
    }
}

function getUser(client_id, client_name){
    if (client_id){
        return getUserById(client_id);
    }else{
        return getUserByName(client_name);
    }
}
```

```
function getUserByName(client_name){
    var indice = 0;
    var res = -1;
    while(indice < queueUsers.size()){
        var usr_aux = queueUsers.get(indice);
        if(usr_aux.name == client_name){
            return usr_aux;
        }
        indice++;
    }
}

function getUserById(client_id){
    var indice = 0;
    var res = -1;
    while(indice < queueUsers.size()){
        var usr_aux = queueUsers.get(indice);
        if(usr_aux.id == client_id){
            return usr_aux;
        }
        indice++;
    }
}

function getIndexUser(id){
    var indice = 0;
    var res = -1;
    while(indice < queueUsers.size()){
        if(queueUsers.get(indice).id == id){
            res = indice;
            break;
        }
        indice++;
    }
    return res;
}

function getName(id){
    var indice = 0;
    var res;
    while(indice < queueUsers.size()){
        if(queueUsers.get(indice).id == id){
```

```
            res = queueUsers.get(indice).name;
            break;
        }
        indice++;
    }
    return res;
}

function nextItem(id){
    var indice = 0;
    var res;
    while(indice < queueUsers.size()){
        if(queueUsers.get(indice).id == id){
            res = queueUsers.get(indice);
            res.item++;
            break;
        }
        indice++;
    }
    return res.item;
}

function repetido(id,usuario,id_usuario){
    for(var i = 0; i< queueUsers.size(); i++){
        if (queueUsers.get(i).id==id_usuario){
            var us = queueUsers.get(i);
            for (var j = 0; j< us.listIds.length;j++){
                if (us.listIds[j]==id)
                    return true;
            }
            return false;
        }
    }
    return false;
}

function addUserToList(name,id,item_aux,step_aux) {
    //load the structure of user
    var listUsers = $("#listUser")[0];
    var res = listUsers.innerHTML;
    //add a new li with the user
    res += '<li id="user:'+id+'" class="lineaUser"><span style="display:flex;"><icon
class="glyphicon glyphicon-user"></icon>' + name;
```

```
    //Progress Bar
    res += testMode ? getBadge(id,name,(item_aux == -
1)?1:item)+getProgressBar(id,name,(step_aux==-
1)?1:step_aux)+getButtonViewResponses(id,name,(step_aux==-1)?1:step_aux) : '';
    //close the structure
    res += '</span></li>';
    //update the box with the student
    listUsers.innerHTML = res;
}

function removeUserFromList(name,id) {
    //delete user from ul
    $('#user\\:'+id).remove();
    queueUsers.removeOfPos(getIndexUser(id));
}

function cargarUsuario(name,id,item_aux,step_aux){
    if (!estaUsuario(id)){
        var us = new User();
        us.name = name;
        us.id = id;
        //si estamos en el test anadimos el paso por el que va el usuario
        if (testMode){
            us.item = item_aux;
            us.step = step_aux;
        }
        us.listaId = [];
        addQueueUsers(us);
        if (testMode){
            addUserToList(us.name,us.id,us.item,us.step);
        }
        else{
            addUserToList(us.name,us.id);
        }
        addUserTo(us.id);
        logger("Se ha a&nacute;adido el usuario "+id+" --> "+ name + " al listado de usuarios");
    }else{
        logger("El usuario "+id+" --> "+ name+" ya se encuentra en la lista de usuarios");
    }
}

function estaUsuario(id){
    for(var i = 0; i< queueUsers.size(); i++){
        if (queueUsers.get(i).id==id)
```

```
        return true;
    }
    return false;
}

//Salida del chat

function inicLanguaje() {
    $("#EstadoUsuarios")[0].innerHTML += listUser;
    $("#tituloChatUser")[0].innerHTML += tituloChatUser;
    $("#allResp").text(viewAllResp);
    $("#escritura")[0].placeholder = placeholderescritura;
}
$( window ).bind( 'beforeunload', function() {
    sendMessageEnd();
} );

function sendMessageEnd() {
    var cmd = new Command();
    cmd.client_id = client_id;
    cmd.client_name = client_name;
    cmd.session = session;
    cmd.channel = '';
    cmd.ip = '127.0.0.1';
    cmd.type = END;
    addQueueOUT(cmd)
    sendInformationUrg(cmd);
}

// pulsar intro para enviar
$(document).keypress(function (e) {
    if (e.which == 13) {
        respuesta();
    }
});

var Queue = function ()
{
    this.list = [];

    this.add = function (data) {
        this.list.push(data);
    };
```

```
this.get = function (i) {
    return this.list[i];
};

this.getDel = function (i) {
    var obj = this.get(i);
    this.list.splice(i, 1);
    return obj;
};

this.getFirst = function () {
    return this.get(0);
};

this.removeFirst = function () {
    this.list.splice(0, 1);
};

this.removeOfPos = function(pos){
    this.list.splice(pos, 1);
}

this.removeN = function (n) {
    for (var i = 0; i < n; i++)
        this.removeFirst();
};

this.getDelFirst = function () {
    return this.getDel(0);
};

this.getN = function (n) {
    if (n > this.size())
        n = this.size();
    else if (n < 0)
        n = 0;
    return this.list.slice(0, n);
};

this.confirmaN = function (n) {
    this.removeN(n);
};
```

```
    this.toString = function () {
        return this.list.toString();
    };

    this.isEmpty = function () {
        return (this.list.length == 0);
    };

    this.size = function () {
        return this.list.length;
    };

    this.getlist = function () {
        return this.list;
    };
};

function initQueues() {
    queueOUT = new Queue();
    queueIN = new Queue();
    queueChat = new Queue();
    queueUsers = new Queue();
    if (testMode){
        queueMessages = new Queue();
    }
}


function addQueueMessages(obj){
    queueMessages.add(obj);
}

function addQueueOUT(obj) {
    queueOUT.add(obj);
}

function addQueueIN(obj) {
    queueIN.add(obj);
}

function addQueueChat(obj) {
    queueChat.add(obj);
}
```

```
function addQueueUsers(us){
    queueUsers.add(us);
}

function voltear(){
    var myTable = $('#chatH')[0];
    var newTable = document.createElement('table');
    var maxColumns = 0;
    // Find the max number of columns
    for(var r = 0; r < myTable.rows.length; r++) {
        if(myTable.rows[r].cells.length > maxColumns) {
            maxColumns = myTable.rows[r].cells.length;
        }
    }
    for(var c = 0; c < maxColumns; c++) {
        newTable.insertRow(c);
        for(var r = 0; r < myTable.rows.length; r++) {
            if(myTable.rows[r].length <= c) {
                newTable.rows[c].insertCell(r);
                newTable.rows[c].cells[r] = '-';
            }
            else {
                newTable.rows[c].insertCell(r);
                newTable.rows[c].cells[r].innerHTML = myTable.rows[r].cells[c].innerHTML;
            }
        }
    }
    newTable.id = 'chatV';
    chatVertical = newTable;
    chatHorizontal = myTable;
    myTable.parentNode.appendChild(newTable);
    myTable.innerHTML = '<table id="chatH"></table>';
}


function DesSerializar(obj) {
    var obj2;
    try{
        obj2 = JSON.parse(obj);
    }catch (err){
        obj2=obj;
    }finally{
        if (obj2.message != null) {
```

```
            desMessage(obj2.message);
        } else if (obj2.command != null) {
            desCommand(obj2.command);
        } else if (obj2.messageContainer != null) {
            desMessages(obj2.messageContainer.messages);
        } else {
            fallo();
        }
    }
}

function fallo() {
    logger('fallo durante el desserializado del objeto JSON');
}

function desMessages(mensajes) {
    for (var i = 0; i < mensajes.length; i++) {
        DesSerializar(mensajes[i]);
    }
}

function desMessage(mensaje) {
    addQueueIN(new Message(mensaje));
}

function desCommand(comando) {
    addQueueIN(new Command(comando));
}

function addUserTo(to){
    var cmd = initMsn();
    cmd.channel = "Cmd channel"
    cmd.ip =  dirIP
    cmd.type = ADD_USER;
    cmd.to = to;
    cmd.subject = (new Command()).getAsString(ADD_USER);
    addQueueOUT(cmd);
}

function esperoRespuestas() {
    var cmd = initMsn();
    cmd.channel = "Cmd channel";
    cmd.ip =  dirIP;
    cmd.type = WAIT_RESPONSES;
```

```
    cmd.to = "others";
    cmd.subject = (new Command()).getAsString(WAIT_RESPONSES);
    addQueueOUT(cmd);
}

function prepararReceive(){
    var cmd = initCmd();
    cmd.ip= dirIP;
    cmd.type = RECEIVE;
    addQueueOUT(cmd);
}

//funcion para enviar un update user status
function prepararUpdateUserStatus(){
    var msn = initMsn();
    msn.channel = "Cmd channel";
    msn.type = UPDATE_USER_STATUS;
    msn.to = "others";
    msn.subject = UPDATE_USER_STATUS;
    msn.ip = dirIP;
    return msn;
}

function preparaACK() {
    var cmd = initCmd();
    cmd.ip = dirIP;
    cmd.type = ACK;
    addQueueOUT(cmd);
}

function prepararNACK(){
    var cmd = initCmd();
    cmd.body = 'El cliente no ha entendido el mensaje recibido por el servidor';
    cmd.ip = dirIP;
    cmd.type = NACK;
    addQueueOUT(cmd);
}

function initCmd() {
    var obj = new Command();
    obj.initCommand(this.client_id, this.client_name, this.session);
    return obj;
}
```

```
function initMsn() {
    var obj = new Message();
    obj.initMessage(this.client_id, this.client_name, this.session);
    var user = queueUsers.get(getIndexUser(client_id));
    obj.item = item;
    obj.step = step;
    obj.session = session;
    obj.client_name = client_name;
    obj.ip = dirIP;
    obj.id = nuevaIdMensaje();
    return obj;
}

function prepararSalida() {
    return queueOUT.getFirst().getJSON();
}

function viewAllResponses() {
    var msg = initMsn();
    msg.channel="Cmd channel";
    msg.type =SEND;
    msg.subject="view_all_answers";
    msg.ip = '127.0.0.1';
    sendInformationUrg(msg);
    eval(functionViewAllResponses+'('+client_id+')');
}

function log (text){
    console.log(text);
}

function logger(text){
        if (debug == 1){
        logdebug(text);
    }else if (alert == 1){
        logAlert(text);
    }else{
        logdebug(text);
    }
}

function logAlert(text){
    alert(text);
}
```

```
function logWarn (text){
    console.warn(text);
}
function logError (text){
    console.error(text);
}
function logdebug (text){
    console.debug(text);
}

function respuesta() {
    var escri = $('#escritura')[0];
//      var destinatario = form.destinatario.value;
    var texto = escri.value;
    escri.value = "";
    // var ind = document.getElementById("listUser").selectedIndex;
    var ind = 0;
    if (texto.length > 0) {
        if (ind != 0){
            destinatario=queueUsers.get(ind).id;
            responderA(texto, destinatario);
        }else {
            responder(texto);
        }
    } else {
        escri.focus();
        logger('Por favor rellene el campo de mensaje');
    }
}

function annadeRespuesta(usuario, texto,aling,id,id_usuario) {
    if (!repetido(id, usuario, id_usuario)){
        if(usuario.id != client_id){
            var chat = $('#listText')[0];
            if (aling == "izq")
                chat.innerHTML += '<li class='+aling+'>' + usuario + ': ' + texto+'</li>';
            else
                chat.innerHTML += '<li class='+aling+'>' + texto + ' :' + usuario+'</li>';
            chat.parentNode.parentNode.scrollTop = chat.parentNode.scrollHeight;
            respuestas = chat;
        }
    }
}
```

```
function annadeRespuestaLocal(usuario, texto,aling) {
    if(usuario.id != client_id){
        var chat = $('#listText')[0];
        if (aling == "izq")
            chat.innerHTML += '<li class='+aling+'>' + usuario + ': ' + texto+'</li>';
        else
            chat.innerHTML += '<li class='+aling+'>' + texto + ' :' + usuario+'</li>';
        chat.parentNode.parentNode.scrollTop = chat.parentNode.scrollHeight;
        respuestas = chat;
    }
}

function nuevaIdMensaje(){
    var res ='';
    var fecha = new Date();
    res+=ajustar(fecha.getDate());
    res+=ajustar(fecha.getMonth()+1);
    res+=fecha.getFullYear();
    res+=ajustar(fecha.getHours());
    res+=ajustar(fecha.getMinutes());
    res+=ajustar(fecha.getSeconds());
    res+=fecha.getMilliseconds();
    return res;
}

function ajustar(dato){
    if (dato>=10)
        return dato;
    else
        return '0'+dato;
}

function responder(texto) {
    var cmd = initMsn();
    cmd.channel="Data channel";
    cmd.ip = dirIP;
    cmd.type = ADD_RESPONSE;
    cmd.to = "others";
    cmd.body = texto;
    cmd.subject = "answer";
    cmd.id = nuevaIdMensaje();
    addQueueChat(cmd);
    sendInformationUrg(cmd);
```

```
    //addQueueOUT(cmd);
    //sendInformation();
//    annadeRespuesta(usuario, texto);
}

function responderA(texto, to) {
    var cmd = initMsn();
    cmd.ip = dirIP;
    cmd.type = ADD_RESPONSE;
    cmd.to = to;
    cmd.body = texto;
    cmd.subject = "answer";
    cmd.id = idMensaje++;
    addQueueChat(cmd);
    addQueueOUT(cmd);
}

function estadoChat(estado){
    $(".tipoPruebas")[0].setAttribute('disabled',!estado);
}

function estadoEscritura(estado){
    var chat = $("#escritura")[0];
    chat.disabled = !estado;
    chat.readOnly = !estado;
}

var Command = function(obj){
    this.id=null;
    this.client_id=null;
    this.client_name=null;
    this.session=null;
    this.channel=null;
    this.ip=null;
    this.error=null;
    this.type=null;

    this.toString = function(){
        var st = "";
        st += "id = " +this.id + " ";
        st += "client_id = " +this.client_id + " ";
        st += "client_name = " +this.client_name + " ";
        st += "session = " +this.session + " ";
```

```
    st += "channel = " +this.channel + " ";
    st += "ip = " +this.ip + " ";
    st += "error = " +this.error + " ";
    return st+" type = "+ this.getAsString(this.type);
};

this.initCommand = function(client_id, client_name, session) {
    this.session = session;
    this.client_id = client_id;
    this.client_name = client_name;
};

this.setType = function(type){
    this.type=type;
};

this.getType= function(){
        return this.type;
};

this.getId = function() {
    return this.id;
};

this.setId = function(id) {
    this.id = id;
};

this.getChannel = function() {
    return this.channel;
};

this.getClientId=function() {
    return this.client_id;
};

this.getClientName=function() {
    return this.client_name;
};

this.getClient = function() {
return  new Client(this.client_id,this.client_name);
};
```

```
this.getSession =function() {
    return this.session;
};

this.getError=function () {
    return this.error;
};

this.getIp=function() {
    return this.ip;
};

this.setChannel=function(channel) {
    this.channel = channel;
};

this.setIp=function(ip) {
    this.ip = ip;
};

this.setClientClase = function (client) {
    this.client_name = client.getName();
    this.client_id = client.getId();
};

this.setClient = function (client_id, client_name) {
    this.client_name = client_name;
    this.client_id = client_id;
};

this.setClientId = function (client_id) {
    this.client_id = client_id;
};

this.setClientName=function(client_name) {
    this.client_name = client_name;
};

this.setSession=function(session) {
    this.session = session;
};
```

```
this.setContext=function(type) {
    this.dataType = type;
};

this.setError=function (error) {
    this.error = error;
};

this.put = function(str, da) {
    return "\"" + str + "\"" + ":" + "\"" + da + "\"";
};

this.getJSON = function (){
    var res='{"command":{';
    res+=this.put("id",this.id)+",";
    res+=this.put("client_id",this.client_id)+",";
    res+=this.put("client_name",this.client_name)+",";
    res+=this.put("session",this.session)+",";
    res+=this.put("channel",this.channel)+",";
    res+=this.put("ip",this.ip)+",";
    res+=this.put("error",this.error)+",";
    res+=this.put("type",this.type)+"}}";
    return res;
};



this.getAsString= function(cmd) {
    cmd = parseInt(cmd);
    switch (cmd) {
        case OPEN_SESSION :
            return "open_session";
            break;
        case CLOSE_SESSION :
            return "close_session";
            break;
        case GET_SESSION :
            return "get_session";
            break;
        case ADD_CLIENT :
            return "add_client";
            break;
        case REMOVE_CLIENT :
```

```
        return "remove_client";
        break;
    case CREATE_CHANNEL :
        return "create_channel";
        break;
    case REMOVE_CHANNEL :
        return "remove_channel";
        break;
    case GET_CHANNEL :
        return "get_channel";
        break;
    case JOIN :
        return "join";
        break;
    case LEAVE :
        return "leave";
        break;
    case SEND :
        return "send";
        break;
    case RECEIVE :
        return "receive";
        break;
    case CLIENT_DISCONNECTED :
        return "client_desconnected";
        break;
    case ACK :
        return "ack";
        break;
    case NACK :
        return "nack";
        break;
    case INIT :
        return "init";
        break;
    case END :
        return "end";
        break;
    case ADD_USER :
        return "add_user";
        break;
    case REMOVE_USER :
        return "remove_user";
```

```
                break;
            case WAIT_RESPONSES :
                return "wait_responses";
                break;
            case UPDATE_USER_STATUS :
                return "update_user_status";
                break;
            case ADD_RESPONSE :
                return "add_response";
                break;
            case END_TEST :
                return "end_test";
                break;
            default :
                return "unknown";
        }
    };
    for (var prop in obj) this[prop] = obj[prop];
};

function limpiarSalida() {
    if (queueOUT.isEmpty()) {
        prepararReceive();
        sendInformation();
    } else {
        sendInformation();
    }
}

function limpiarEntrada() {
    while (!queueIN.isEmpty()) {
        var obj = queueIN.getDelFirst();
        var tipo = parseInt(obj.type);
        switch (tipo) {
            case ADD_USER:
                if (testMode){
                    cargarUsuario(obj.client_name, obj.client_id, obj.item, obj.step);
                }else{
                    cargarUsuario(obj.client_name, obj.client_id);
                }
                break;
            case ADD_RESPONSE:
                if (obj.to == "all"){
```

```
                    if (testMode){
                        if (step == 1 && obj.item == item){
                            annadeRespuesta(obj.client_name, obj.body.replace(/[+]/g, "
"),'izq',obj.id,obj.client_id);
                        }else{
                            addQueueMessages(obj);
                        }
                    }else{
                        annadeRespuesta(obj.client_name, obj.body.replace(/[+]/g, "
"),'izq',obj.id,obj.client_id);
                    }
                }else{
                    if (testMode){
                        if (step == 1 && obj.item == item){
                            annadeRespuesta(obj.client_name, obj.body.replace(/[+]/g, "
"),'izq',obj.id,obj.client_id);
                        }else{
                            addQueueMessages(obj);
                        }
                    }else{
                        annadeRespuesta(obj.client_name, obj.body.replace(/[+]/g, "
"),'izq',obj.id,obj.client_id);
                    }
                }
                break;
            case SEND:
                if (obj.to == "all"){
                    if (testMode){
                        if (step == 1 && obj.item == item){
                            annadeRespuesta(obj.client_name, obj.body.replace(/[+]/g, "
"),'izq',obj.id,obj.client_id);
                        }else{
                            addQueueMessages(obj);
                        }
                    }else{
                        annadeRespuesta(obj.client_name, obj.body.replace(/[+]/g, "
"),'izq',obj.id,obj.client_id);
                    }
                }else{
                    if (testMode){
                        if (step == 1 && obj.item == item){
                            annadeRespuesta(obj.client_name, obj.body.replace(/[+]/g, "
"),'izq',obj.id,obj.client_id);
```

```
                        }else{
                            addQueueMessages(obj);
                        }
                    }else{
                        annadeRespuesta(obj.client_name,            obj.body.replace(/[+]/g,            "
"),'izq',obj.id,obj.client_id);
                    }
                }
                break;
            case REMOVE_USER:
                removeUserFromList(obj.client_name, obj.client_id);
                break;
            case WAIT_RESPONSES:
                addUserTo(obj.client_id);
                break;
            case ACK:
                manejoACK(obj);
                break;
            case NACK:
                manejoNACK(obj);
                break;
            case UPDATE_USER_STATUS:
                manejoUpdateUserStatus(obj);
                break;
        }
    }
}

function cleanQueueChat() {
    var aux = queueChat.getDelFirst();
    if (aux.to == "all" || aux.to == "others")
        annadeRespuestaLocal(aux.client_name, aux.body.replace(/[+]/g, " "),'der');
    else
        annadeRespuestaLocal(aux.client_name, aux.body.replace(/[+]/g, " "),'der');
}

function confirmaMen() {
    var obj = queueOUT.getFirst();
    if (obj!=null){
            var tipo = parseInt(obj.type);
            switch (tipo) {
                case ADD_RESPONSE:
                    queueOUT.getDelFirst();
```

```
                cleanQueueChat();
                break;
            case INIT:
                queueOUT.getDelFirst();
                esperoRespuestas();
                break;
            case WAIT_RESPONSES:
                if (!testMode){
                    estadoChat(true);
                    estadoEscritura(true);
                }
                queueOUT.getDelFirst();
                break;
            default :
                queueOUT.getDelFirst();
        }
    }

}

function manejoNACK(obj){
    logger("Ha llegado un NACK");
}

function manejoACK(obj){
    logger("Ha llegado un ACK");
}

function limpiarACKSalida(){
    var queueAux = new Queue();
    for (var i = 0; i < queueOUT.size(); i++){
        if(queueOUT.get(i).type == ACK){
            queueAux.add(i);
        }
    }
    for (var i = queueAux.size()-1; i>=0 ;i--){
        queueOUT.getDel(queueAux.get(i));
    }
}

function cleanQueueMessages(){
        var queueAux = new Queue();
        for(var i = 0; i< queueMessages.size(); i++){
```

```
                var obj = queueMessages.get(i);
                if (obj.item > item){
                        queueAux.add(obj);
                }else if (obj.item == item){
                        annadeRespuesta(obj.client_name,        obj.body.replace(/[+]/g,        "
"),'izq',obj.id,obj.client_id);
                }
        }
        queueMessages = new Queue();
        for(var i = 0; i< queueAux.size(); i++){
                addQueueMessages(queueAux.get(i));
        }
}

// Funtion tu manage a update user status
function manejoUpdateUserStatus(obj){
    if (testMode){
        jsSetStateUser(obj.client_id, obj.item, obj.step);
    }
}


var Message = function(obj) {
    this.id = null;
    this.client_id = null;
    this.client_name = null;
    this.session = null;
    this.channel = null;
    this.type=null;
    this.ip = null;
    this.error = null;
    this.to = null;
    this.reply_to = null;
    this.subject = null;
    this.body = null;
    this.item = ITEM_NULL;
    this.step = STEP_NULL;


    this.getJSON = function() {
        var res = '{"message":{';
        res += this.put("id", this.id) + ",";
        res += this.put("client_id", this.client_id) + ",";
```

```
    res += this.put("client_name", this.client_name) + ",";
    res += this.put("session", this.session) + ",";
    res += this.put("channel", this.channel) + ",";
    res += this.put("ip", this.ip) + ",";
    res += this.put("error", this.error) + ",";
    res += this.put("type", this.type) + ",";
    res += this.put("to", this.to) + ",";
    res += this.put("reply_to", this.reply_to) + ",";
    res += this.put("subject", this.subject) + ",";
    res += this.put("body", this.body) + ",";
    res += this.put("item", this.item) + ",";
    res += this.put("step", this.step) + "}}";
    return res;
};

this.put = function(str, da) {
    return "\"" + str + "\"" + ":" + "\"" + da + "\"";
};

this.toString = function() {
    var st = "";
    st += "to = " + this.to + " ";
    st += "reply_to = " + this.reply_to + " ";
    st += "subject = " + this.subject + " ";
    st += "body = " + this.body + " ";
    st += "item = " + this.item + " ";
    st += "step = " + this.step + " ";
    st += "id = " + this.id + " ";
    st += "client_id = " + this.client_id + " ";
    st += "client_name = " + this.client_name + " ";
    st += "session = " + this.session + " ";
    st += "channel = " + this.channel + " ";
    st += "ip = " + this.ip + " ";
    st += "error = " + this.error + " ";
    var aux = new Command();
    aux.setType(this.type);
    st += "type = " + aux.getAsString() + " ";
    return  st;
};

this.initMessage = function(client_id, client_name, session) {
    this.session = session;
    this.client_id = client_id;
```

```
    this.client_name = client_name;
};

this.InicMessage = function(to, subject, body) {
    this.to = to;
    this.subject = subject;
    this.body = body;
};

this.InicMessageBody = function(to, reply_to, subject, body) {
    Message(to, subject, body);
    this.reply_to = reply_to;
};

this.getBody = function() {
    return this.body;
};

this.getReplyTo = function() {
    return this.reply_to;
};

this.getSubject = function() {
    return this.subject;
};

this.getTo = function() {
    return this.to;
};

this.setBody = function(body) {
    this.body = body;
};

this.setReplyTo = function(reply_to) {
    this.reply_to = reply_to;
};

this.setSubject = function(subject) {
    this.subject = subject;
};

this.setTo = function(to) {
```

```
    this.to = to;
};

this.getType = function() {
    return this.type;
};

this.setType = function(type) {
    this.type = type;
};

this.setData = function(data) {
    this.datos = data;
};

this.getData = function() {
    return this.datos;
};

this.setDataType = function(type) {
    this.dataType = type;
};

this.getId = function() {
    return this.id;
};

this.setId = function(id) {
    this.id = id;
};

this.getChannel = function() {
    return this.channel;
};

this.getClientId = function() {
    return this.client_id;
};

this.getClientName = function() {
    return this.client_name;
};
```

```
this.getClient = function() {
    return  new Client(this.client_id, this.client_name);
};

this.getSession = function() {
    return this.session;
};

this.getDataType = function() {
    return this.dataType;
};

this.getError = function() {
    return this.error;
};

this.getIp = function() {
    return this.ip;
};

this.setChannel = function(channel) {
    this.channel = channel;
};

this.setIp = function(ip) {
    this.ip = ip;
};

this.setClientClase = function(client) {
    this.client_name = client.getName();
    this.client_id = client.getId();
};

this.setClient = function(client_id, client_name) {
    this.client_name = client_name;
    this.client_id = client_id;
};

this.setClientId = function(client_id) {
    this.client_id = client_id;
};

this.setClientName = function(client_name) {
```

```
        this.client_name = client_name;
    };

    this.setSession = function(session) {
        this.session = session;
    };

    this.setContext = function(type) {
        this.dataType = type;
    };

    this.setError = function(error) {
        this.error = error;
    };
    for (var prop in obj) this[prop] = obj[prop];
};

function sendInformation(){
    var content = prepararSalida();
    $.ajax({
        url: URL,
        type: 'POST',
        dataType: 'text',
        data: {data: content},
        success: function (data){
            var params = getParam(data, "post");
            var data = "";
            if (params) {
                // hacemos un bucle para pasar por cada indice del array de valores
                for (var p in params) {
                    if (p == "data") {
                        data = decodeURIComponent(params[p]);
                    }
                }
            }
            logger("The server has sent --> "+ data);
            logger("Ajax transfers success");
            var tam = queueIN.size();
            var tam2;
            DesSerializar(data);
            tam2 = queueIN.size();
            if (tam < tam2){
                confirmaMen();
```

```
            }
            limpiarEntrada();
        }
    })
    .fail(function() {
        logger("Error");
        logger('Fail in ajax');
        logger('Fail with the message -> '+ decodeURIComponent(content));
    });
}

function sendInformationUrg(message){
    var content = message.getJSON();
    logger("Send to the server -->"+content);
    $.ajax({
        url: URL,
        type: 'POST',
        dataType: 'text',
        data: {data: content},
        success: function (data){
            var params = getParam(data, "post");
            var data = "";
            if (params) {
                // hacemos un bucle para pasar por cada indice del array de valores
                for (var p in params) {
                    if (p == "data") {
                        data = decodeURIComponent(params[p]);
                    }
                }
            }
            contInf++;
            logger("The server has sent --> "+ data);
            logger("Ajax transfers success");
            var aux = DesSerializarUrg(data);
            processInfUrg(message);
            if (aux != null){
                    limpiarEntrada(aux);
            }
        }
    })
    .fail(function() {
        logger("Error");
        logger('Fail in ajax');
        logger('Fail with the message -> '+ decodeURIComponent(content));
```

```
    });
}

function processInfUrg(obj){
        var tipo = parseInt(obj.type);
      switch (tipo) {
         case ADD_USER:
            if (testMode){
               cargarUsuario(obj.client_name, obj.client_id, obj.item, obj.step);
            }else{
               cargarUsuario(obj.client_name, obj.client_id);
            }
            break;
         case ADD_RESPONSE:
            if (obj.to == "all"){
                if (testMode){
                        if (step == 1 && obj.item == item){
                                annadeRespuesta(obj.client_name,    obj.body.replace(/[+]/g,    "
"),'der',obj.id,obj.client_id);
                        }else{
                                addQueueMessages(obj);
                        }
                }else{
                        annadeRespuesta(obj.client_name,        obj.body.replace(/[+]/g,        "
"),'der',obj.id,obj.client_id);
                }
            }else{
               if (testMode){
                        if (step == 1 && obj.item == item){
                                annadeRespuesta(obj.client_name,    obj.body.replace(/[+]/g,    "
"),'der',obj.id,obj.client_id);
                        }else{
                                addQueueMessages(obj);
                        }
                }else{
                        annadeRespuesta(obj.client_name,        obj.body.replace(/[+]/g,        "
"),'der',obj.id,obj.client_id);
                }
            }
            break;
         case SEND:
            if (obj.to == "all"){
               if (testMode){
```

```
                        if (step == 1 && obj.item == item){
                            annadeRespuesta(obj.client_name,           obj.body.replace(/[+]/g,           "
"),'der',obj.id,obj.client_id);
                        }else{
                            addQueueMessages(obj);
                        }
                    }else{
                        annadeRespuesta(obj.client_name,             obj.body.replace(/[+]/g,             "
"),'der',obj.id,obj.client_id);
                    }
                }else{
                    if (testMode){
                        if (step == 1 && obj.item == item){
                            annadeRespuesta(obj.client_name,           obj.body.replace(/[+]/g,           "
"),'der',obj.id,obj.client_id);
                        }else{
                            addQueueMessages(obj);
                        }
                    }else{
                        annadeRespuesta(obj.client_name,             obj.body.replace(/[+]/g,             "
"),'der',obj.id,obj.client_id);
                    }
                }
                break;
            case REMOVE_USER:
                removeUserFromList(obj.client_name, obj.client_id);
                break;
            case WAIT_RESPONSES:
                addUserTo(obj.client_id);
                break;
            case ACK:
                manejoACK(obj);
                break;
            case NACK:
                manejoNACK(obj);
                break;
            case UPDATE_USER_STATUS:
                manejoUpdateUserStatus(obj);
                break;
        }
}
```