



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di laurea in informatica

ELABORATO FINALE

Development of a Web-based Interface for the Orchestration of Machine Learning Components

SUPERVISORE

Prof. Brunato Mauro

LAUREANDO

Berlato Stefano

Anno accademico 2016 - 2017



UNIVERSITA' DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di laurea in informatica

ELABORATO FINALE

Development of a Web-based Interface for the Orchestration of Machine Learning Components

SUPERVISORE

Brunato Mauro

LAUREANDO

Berlato Stefano

Anno accademico 2016 - 2017

Index

[Index](#)

[Summary](#)

[Chapter 1 - Interface Presentation](#)

[1.1 - Choose Session](#)

[1.2 - Workbench](#)

[Chapter 2 - Thesis Object](#)

[Chapter 3 - Related Work](#)

[3.1 - Front-end](#)

[3.2 - Back-end](#)

[Chapter 4 - Requirements Collection](#)

[Chapter 5 - Designing and architecture](#)

[5.1 - Server architecture](#)

[5.2 - Message Exchanging Definition](#)

[5.3 - Message Exchanging Timing](#)

[Chapter 6 - Development](#)

[6.1 - Websocket code development](#)

[6.2 - File system code development](#)

[6.3 - ChooseSession.tpl development](#)

[6.4 - JSPlumb© interface code development](#)

[Chapter 7 - Javascript Classes](#)

[7.1 - Mother class development](#)

[7.2 - Dynamic class instantiation](#)

[Chapter 8 - Example Class Implementation](#)

[Chapter 9 - Conclusions](#)

[Sitography](#)

[Attachments](#)

[attachment A - Interface Screenshots](#)

[Choose Session](#)

[Workbench](#)

[attachment B - Design and Architecture](#)

[Message Exchanging Definition](#)

[User Connection Flow](#)

[First Connection message exchanging Activity Diagram](#)

[addObject message Activity Diagram](#)

[addConnection message Activity Diagram](#)

Summary

This thesis wells from the plenty of resources that the Machine Learning environment has reached nowadays, and the difficulty of integrating them efficiently by using different means. In fact, most of them are available online but, even though they are accessible and reachable very easily, the user has to navigate through all these platforms and manually handle the interaction between them to achieve his purposes.

I.e. download databases and query the data, implement algorithms and run them via the shell, use external tools to plot the results, et cetera.

The aim is to develop a web based, multiple-client server interface for giving the possibility to make these elements interoperate with each other in a quick and simple way.

The idea is to have several work sessions where more clients can operate together synchronously, instantiating Machine Learning related modules such as databases, algorithms, neural networks and so on. Then these elements can be opportunely configured and connected together so to give them a semantic meaning.

I.e. link a database to an algorithm makes the latter analyze the data provided by the first.

Every change made by a user is reflected on the others, from a simple movement of an object to the utter deletion of another one; besides, it's also saved on the file system of the server.

The purpose is to build the foundations for such system, in order to provide all the needed basic instruments. These will be subsequently enriched with modules, some of them mentioned before, that should just take care of their own functionalities exploiting the system ones.

The starting point is the internship I have done last summer at **HEAS srl**, an office whose job regards hydroelectric plants designing and management. My work involved the integration of a pre-existent commercial SCADA system named **ATWISE®**, implemented using web standards like *HTML5* and *SVG*.

The client interface received via browser was composed of several objects that report noticeable values concerning the plant, such as turbine speed, water pressure, et cetera. Since the nature of these objects was a static one, my time was dedicated to improving this aspect of the interface by making possible to dynamically create these elements from a toolbar. Then the final user would be able to configure them and place them wherever he wanted thanks to a drag and drop functionality.

Each of these functionalities was implemented with *Javascript* and *SVG*.

In the first meeting with the supervisor professor, I showed him my internship labor, and consequently we decided the object of the thesis. Then I began exactly from that project, first of all trying to estrapolate the code from the **ATVISE**® SCADA system in order to make it stand alone.

After that I had a point from which I could start, I continued looking for some related work, thus libraries and small projects, that could help me in what I was doing. Among several possibilities, I decided to use the community MIT edition of a commercial JS library, **JSPlumb**®, that provides APIs for make HTML elements draggable and interactive. For the back-end side I picked out **Node.js**®, and for editing the code I chose **Brackets** text editor.

In another meeting with the supervisor professor we proceeded specifying requirements, answering arisen questions and detailing different aspects of the interface. Then it was time for the modeling and designing part; for UML activity diagrams and back-end architecture I exploited **yEd**® graph editor, a simple and rapid tool. Since a synchronously inter clients communication was requested, I planned a format for exchanging messages as JSON. For the implementation I preferred to follow a bottom-up approach: I built firstly the basic functionalities such as websocket and file system management, and lastly the front-end side of the application, thus the client interface code joint with the **JSPlumb**® library. The HTML files the clients receive are generated out of TPL template files, properly elaborated by the **Bind** module of **Node.js**®.

The only development took about two weeks of work, at the end of which all the requirements were achieved. In addition to the interface requirements, I have also implemented, for illustrative purposes, a SQL database module. This module can be properly configured from the special panel, and makes possible to connect to a given server hosting the database and retrieving the data that match a certain SQL query.

Chapter 1 - Interface Presentation

The interface is by now composed of two different pages. The first lets the user choose the desired working session, also giving him the list of already existent ones, whereas the second wraps all the functionalities mentioned before, and it's the actual concern of the thesis work.

1.1 - Choose Session

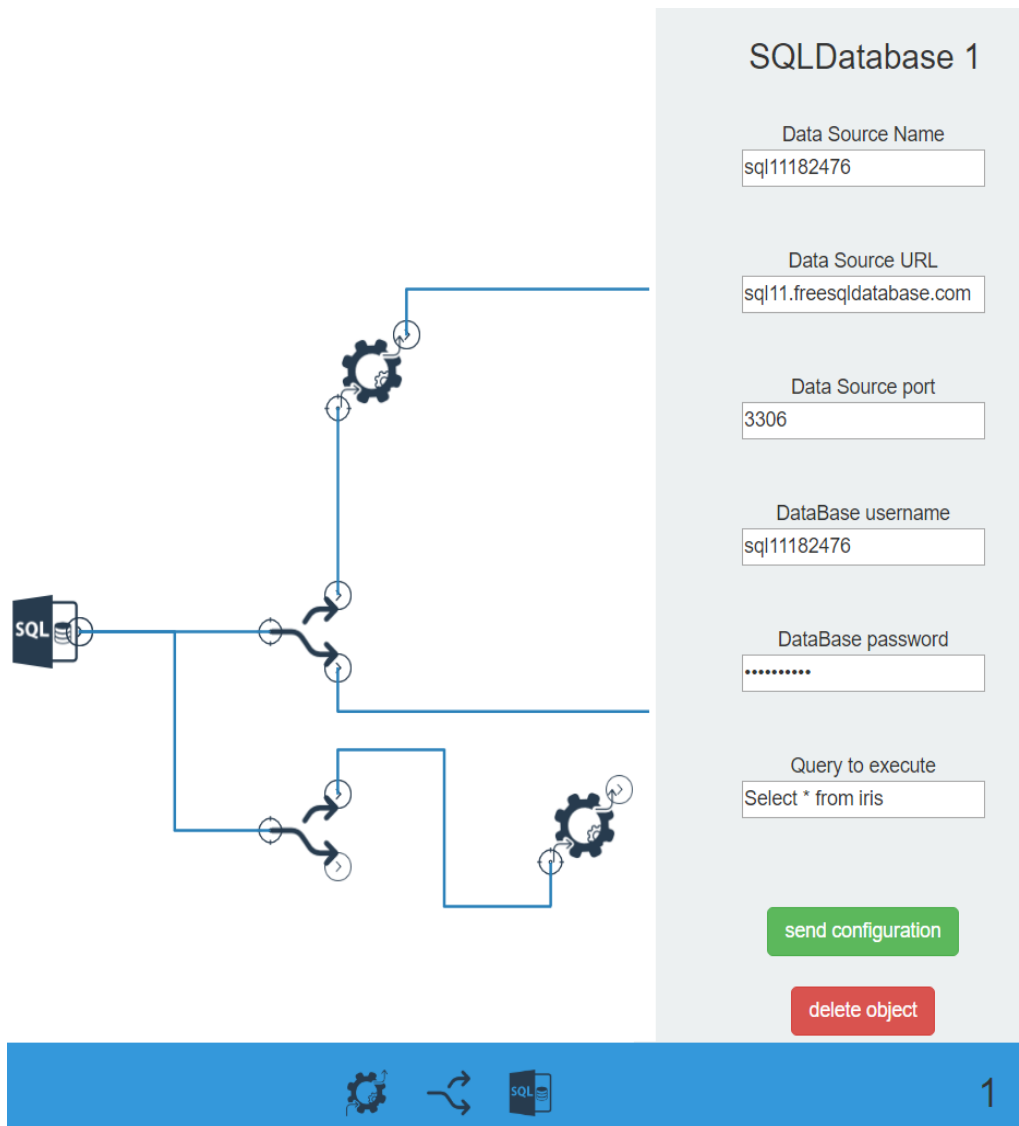
The *ChooseSession.tpl* page presents a flat design form, that allows the user to select a session from the list of the existent ones. Otherwise, it is possible to create a brand new one by just typing the name in the input field. If the name happens to be the same as an already extant session, the user will be instead redirected to that content.

Note: Since the requirements doesn't include it, a password is not requested.

1.2 - Workbench

Once the user picks out a session, the server will respond with the proper content saved from the previous work, if any. If it is the case, the old page status will be restored.

In *Workbench.tpl* there are three macro areas:



Toolbar

A flat blue strip at the bottom of the page. There will be a representative icon for each available Machine Learning related module. By just clicking onto one of these icons, a new object of the chosen type will be instantiated in the graph area. If the number of

modules is big enough to overflow the width of the blue strip with the icons, the user will be able to perform a horizontal scroll hitting the `L. shift` key and using the mouse wheel. This is styled in CSS so that the scrollbar does not appear. On the right, there's also a number that tells how many clients are connected to that particular session in that moment.

Graph

It's the biggest area, located on the top-left part of the page, that holds the whole content. It allows the user to drag and drop elements and creates new connections from the special areas of the objects. Since the graph may grow too big to fit into the screen, there is the possibility to pan the graph by just click and drag on the white space.

Configuration Panel

A grey column at the right side of the page. By clicking onto an element in the graph, this column will be filled with the predefined inputs for that specific class. Besides, the header of the panel will be replaced with the class name of the elements plus its ID. The user will be able to configure the object parameters through these fields, save the configuration and, lastly, remove the object through the *delete* red button.

Chapter 2 - Thesis Object

In the first meeting with the supervisor professor, I showed him my internship code, mostly based on *Javascript* and *SVG*. Then, considering the affinity between the two projects, there was deliberated the subject of my thesis. As mentioned before, the theme is just the interface.

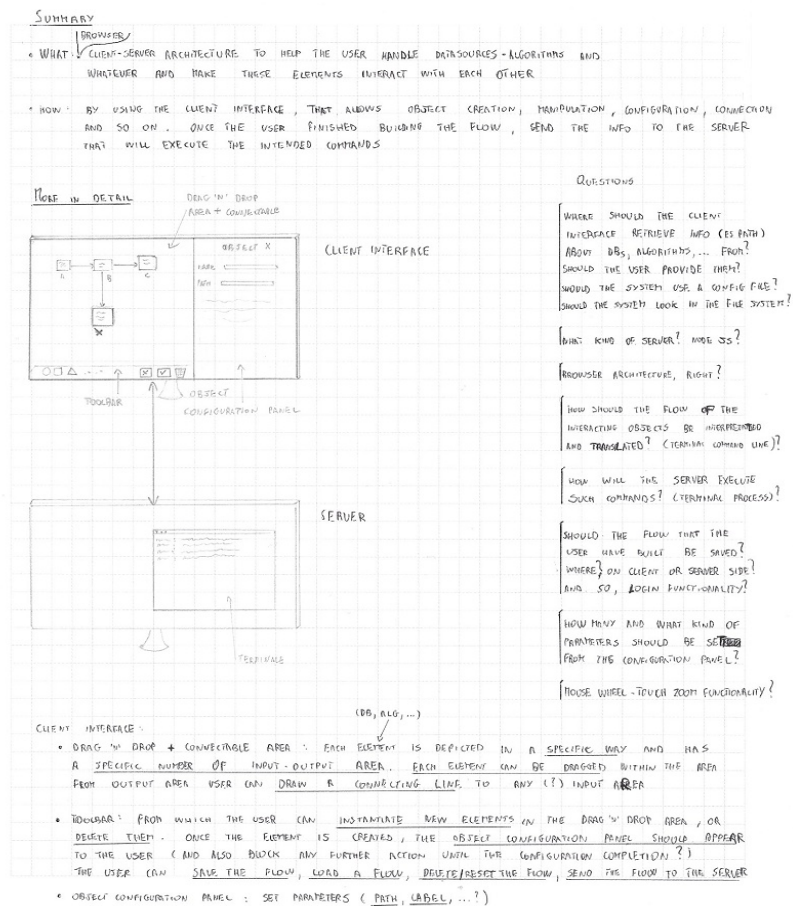
Note: the implementation of the Machine Learning related modules is to be done afterwards.

The proposal was to implement a browser-based client-server architecture that aims to help the user handle Machine Learning elements interaction; these were the initial requirements:

- The user should be able to instantiate and delete objects.
- They should be depicted with a significative icon
- Between the objects, the user should be given the possibility to drag a connecting line.

The discussion regarded the main purposes of the system, trying to understand its usefulness and worthiness. So the details were left for later.

After our bout, I tried to put on paper what I understood of the requirements, annotating questions and doubts.









Chapter 3 - Related Work

Meanwhile, I extrapolated the code I wrote during the internship at **HEAS srl** in order to have a point from which I could start. That was not an easy task to perform, since the platform builder strongly influenced the architecture of the system. Besides, because **Atvise®** is a commercial SCADA, few or none documentation is available on how it internally treats its resources. Then I began to seek some related work that could help me during the development. I searched free-to-use libraries, **Github®** repositories, **Node.js®** projects and so on. The supervisor professor helped me during this phase, linking me the library for the interface that I eventually chose.

3.1 - Front-end

For the front-end side, I collected several options and compared them:

library	icon	license	clearAPI	eventHandlers	drag	connectability
<i>BonsaiJS</i>		MIT	no	yes	yes	no
<i>Snap.js</i>		Apache2	yes	yes	yes	no
<i>D3.js</i>		BSD	yes	yes	yes	no
<i>SVG.js</i>		MIT	yes	yes	yes	yes
<i>cytoscape.js</i>		MIT	no	yes	yes	no
<i>jsPlumb</i>		MIT	yes	yes	yes	yes

After have studied those libraries and have seen how they work, eventually I choose **JSPlumb**®. The main reason is that it has the most clear and complete functionalities. Also **SVG.js** library filled all the requirements, but they were achieved using several plugins, not well documented and too low level. In fact, all the libraries allow direct manipulation of SVG files, and I have learned during my internship how difficult could be this vanilla interaction. Only **JSPlumb**® wraps it efficiently, being at the same time more powerful in its features and giving the control that was needed. Since this library has a lot of convenient and well tested functionalities, some of which covering the ones present in my old project code, I started my thesis interface from scratch.

Another library I decided to exploit is **Klass**, a small and very useful library that digests the prototype-based Javascript mechanism making available a easier syntax for defining classes. This would have been useful while working on the Machine Learning related modules. Lastly, I chose **SweetAlert** for HTML alerts, in order to show any errors that may happen to the users in a nicer way.

3.2 - Back-end

For the back-end side I draw some code from an old project of mine, especially for the server setup. Then, aside the standard **Node.js**® modules, I found nothing that could speed up the implementation but some tutorials found on the internet.

Chapter 4 - Requirements Collection

I approached the second meeting with the supervisor professor with these new resources and some questions. They concerned mostly the graphical and logical aspects of the interface client-side, in particular the user interaction with the graph elements.

The result of our discussion was a list of more detailed new requirements regarding the system:

- There should be the possibility to work on different graphs, or sessions, without login
- Users should be able to work together on the same graph, seeing immediately their respective actions
- These graphs should always be instantly updated and saved in the server
- Adding new Machine Learning modules should not involve modifying old code at all
- In the graph it should be implemented a panning functionality, but no zoom
- Each object should must have a specific number of input-output areas from which draw the connections
- Each object should have a personal configuration panel with its input fields

Since the role of the customer was played by the professor, there wasn't any noticeable difficulty of understanding each other during this second requirements collection.

Chapter 5 - Designing and architecture

As written before, I used the **yEd**® graph editor in order to model and design the system architecture. In fact, there were two spheres of designing, front-end and back-end.

5.1 - Server architecture

I loosely used an MVC pattern for organizing the files, but meanwhile I preferred to divide them as the logical resources they are.

I.e: all the JS files went under the same root directory, and then they were categorized.

```
Interface
--> css
--> js
    --> client
    --> server
--> tpl
--> server.js
```

I listed the functionalities that I expected to implement and divided them into files by logical sort. In particular, I would have needed:

- a code for creating, reading and writing files for saving graphs
- a code for managing and sending messages over websockets
- a file to centralize all the needful parameters
- a code for the server setup and handling of the **Node.js**® routes

Instead of using plain HTML files, I opted for TPL templates that would be elaborated server-side by the **Bind** module of **Node.js**®. An application example that is reported later on is in *ChooseSession.tpl*, where the list of extant sessions is filled that way.

5.2 - Message Exchanging Definition

Since a lot of messages would be exchanged between the clients to the server and vice versa, it was proper to have, if not a protocol, at least a predefined way to format the information. I thought about how many message types would I need, and that that list will surely be expanded in the future. So I chose to use JSON over websockets, including among the other information a fundamental attribute named “messageType”. Then I listed all the different possibilities in a parameter file.

```
// list of messages types
var messageTypeHello = "hello";
var messageTypeBye = "bye";
var messageTypeServerJSONResponse = "serverJSONResponse";
var messageTypeServerObjectsResponse = "serverObjectsResponse";
var messageTypeAddObject = "addObject";
var messageTypeModifyObject = "modifyObject";
var messageTypeDeleteObject = "deleteObject";
var messageTypeUndoMessage = "undoMessage";
...
```

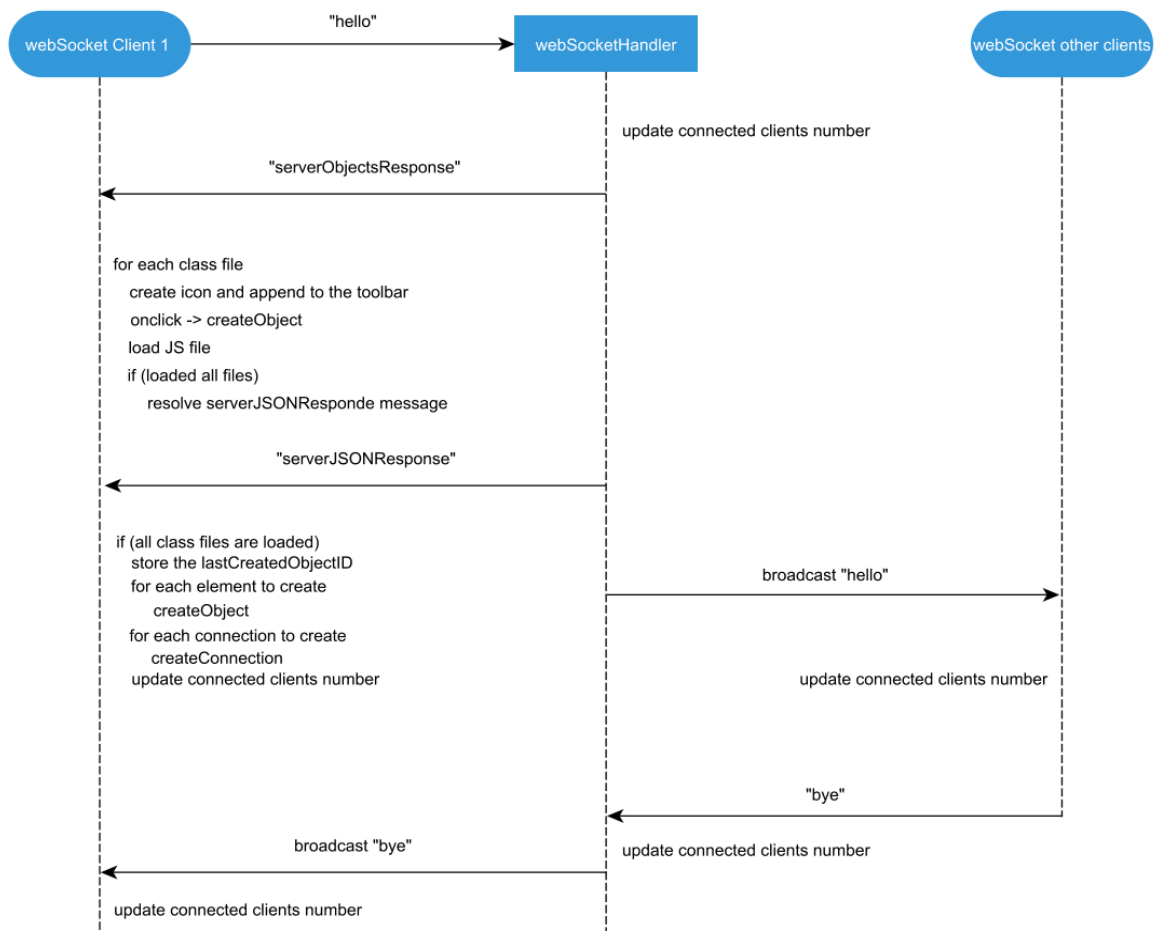
The server and the client, once a message arrives, decide how to treat it basing on this attribute. A complete list with all the relative JSON attributes is reported in the attachments.

5.3 - Message Exchanging Timing

Later on, I worked out how sequences of messages should be managed, creating some UML activity diagrams that helped me during the implementation phase. In the reported scenario, there is represented the ideal flow of messages when an user (client 1) reaches the *workbench.tpl* page of a session where there are already other clients working on. At the end, when one of these clients leaves the page, there is a “bye” message example.

MESSAGES HELLO AND BYE

all is intended to be referred to a specific session equal for all clients



When the user connects and chooses a session, the page he is redirected to initializes a websocket with the server, sending a “hello” message. Then the server responds with the list of Machine Learning related modules available with “serverObjectsResponse”, and the session with its content saved from the previous work, if any, with “serverJSONResponse”. The other activity diagrams will not be discussed here, but they are reported in the attachments.

A considerable aspect was the handling of conflicting messages. Each instance element of a class has a unique incremental ID for avoiding having doubled identifiers. All the clients are

synchronized with that number and, when someone creates something, everyone increase it by one. If two users instantiate an object at the same time, the server will receive two messages “addObject” or “addConnection” that report the same object ID. Only the first one will be accepted, while the other one will be rejected and a “undoMessage” message will be sent to that client.

Note: Other kind of events such as modification or deletion can not be conflicting. At most, a user can modify an object that was already removed, but the server can handle it.

Chapter 6 - Development

As said before, a bottom-up approach was adopted. Firstly, there were build the foundations of the interface, thus websocket code both client and server side. Then the file system, in order to handle the messages sent by a stub module in *Workbench.tpl*. One by one, all the message types were administrated and tested before to proceed to the session management. Once that it was developed, the only thing still missing was the interface. After have styled it with **Bootstrap** CSS, **JSPlumb**® APIs were exploited to complete the system.

As general approach to the development, I have always tried to model and design and then start with the implementation. When writing the code, firstly I composed the documentation and eventually the code.

6.1 - Websocket code development

In browsers, websockets are standardly implemented, and **Node.js**® has a compact and simple module, **ws** that takes that task. Client and server code are supposed to be similar in this case, since their functionalities are quite the same: both have to send and receive messages, and that’s all. Thus, client side *workbench_websocket.js* code and server side *WebSocketHandler.js* code are matter-of-fact composed by a switch statement. This checks the integrity of the message and delegates its handling to the proper class or function basing on the attribute “messageType” after have done a few simple checks.

```
// check the message integrity
var messageJSON = checkMessageIntegrity(message);
...
// swich by message type
switch (messageJSON.messageType) {
    case param.messageTypeHello:
```

```
...  
case param.messageTypeAddObject:  
...
```

For control purposes, every 30 seconds the server sends a “*ping*” message that, if not acknowledged and answered by the client, makes understand that the connection is broken.

```
// for each connected client  
wss.clients.forEach(function each(ws) {  
  // if he is no more connected  
  if (ws.isConnected === false) {  
    // invoke the close connection handling code  
    return closeConnectionHandling(ws);  
  }  
  // otherwise  
  else {  
    // send the ping request  
    ws.ping('', false, true);  
    // the client is not connected until he responds  
    ws.isConnected = false;  
  }  
});
```

6.2 - File system code development

When a client requests a new session, if the name is a proper one, a JSON file of reference is created into a directory that has that session name into a root directory named “sessions”. At creation time, this is what it contains:

```
{"_creationTime":"2017-07-04 12:21:02", "_lastCreatedObjectID":"0"
```

Then the server copies it in an array of cached JSON session files. The final bracket of the JSON is not written; this is done on purpose to allow further additions.

In fact, whenever a new element is created, the cached file is updated and a new pair is appended asynchronously to the file system one. The key is the ID of that object, and the value is a JSON of its attributes, both HTML ones and configuration ones.

```
{"_creationTime":"2017-07-04 12:21:02", "_lastCreatedObjectID":"0",  
"1" : {"objectClass":"SQLDatabase", "id":"1"}
```

The requirements state that the graph of the work session is to be saved as soon as there is any change, updating the server file of reference on the file system.

Therefore, if some attributes of the object are modified, besides the updating of the cache, another pair with the same key is appended to the file, reporting the changes. This approach has multiple vantages:

1. Even if the server crashes, the changes are saved on the file system
2. It's quicker to do this way instead of writing down the whole cached file each time there is a change to save

When the file system file grows too big due to these additions, just write down the cached file as it is to resolve the problem.

```
{"_creationTime":"2017-07-04 12:21:02","_lastCreatedObjectID":"0",  
"1" : {"objectClass":"SQLDatabase","id":"1"}, "1" :  
{"style":["154px","246px"],"databaseSourceName":"myDatabase",  
"username":user07}
```

Also, when all clients are disconnect from the session, there is to write down the cached file. Given the nature of the incremental assigning of IDs to the objects, a “hole” in the IDs sequence is left if an element is deleted. The *writeJSONFile* function in the file system manager code takes care of this, re-assigning and compacting the IDs.

```
// for each ID in session JSON  
for (var id in data){  
  
    // increment cont new ID  
    contNewID++;  
  
    // map the old id with the new one  
    idMap[id] = contNewID;  
}
```

6.3 - ChooseSession.tpl development

Firstly, there was to familiarize again with **Bootstrap**, in order to create an appealing and nice page for letting the users choose their working sessions.

Once this was done, the **Bind** module was exploited for generating HTML pages out of TPL templates. One of the most relevant use was to fill the `<select>` tag with an `<option>` tag for each already extant session available.

```

<select id="sessionsList" class="form-control"
onChange="OnSelectedIndexChange()">
<option style="display:none;" disabled selected value> select a
session </option>

(:data ~ <option value="[:sessionName:]">[:sessionName:]</option>:)

</select>

```

data is a JSON of the available session names under the “sessionName” key. These data are retrieved by a function in the file system manager code, *getNamesFromDir*. This will be also used when there will be the need of getting the list of available Machine Learning related modules.

6.4 - JSPlumb© interface code development

After a plain coding of the interface with **Bootstrap** CSS, it was time to actually develop its functionalities. This was done integrating **JSPlumb©** in order to make elements draggable and connectable. The easy syntax, joint with my previous study of the APIs, allowed me to create some HTML example pages, and that deepens my knowledge of the library. Still, I spent some time familiarizing with the several options available, that let the developer customize quite each graphical aspect.

```

// to make an element draggable
jsPlumb.draggable("element_id", options);

// to allow the user drag connections between elements
jsPlumb.makeSource("element_id", options);
jsPlumb.makeTarget("element_id", options);

// to programmatically connect two elements
jsPlumb.connect({
    source:"source_id",
    target:"target_id",
    options
});

```

JSPlumb© provides a listener trigger system that covers quite all the possible events regarding the interface. In particular, I exploited handlers for these three events:

- “connection”, triggered whenever a connection is created

- “connectionDetached”, triggered whenever a connection is detached
- “connectionMoved”, triggered whenever a connection is moved

Each of these distinguish when the event is user-made or programmatically-made, thus knowing whether the event should be reported to the server or it comes from the server itself.

Since I planned to implement the dynamic instantiation of objects afterwards, I used a static class element. This was supposed to be, once I have finished its development, the mother class from which all the other Machine Learning related classes would inherit. This is properly explained in the chapter that follows

Chapter 7 - Javascript Classes

The already mentioned **klass** library provides a easier way to declare classes in Javascript. Is it possible to have inheritance, static attributes and methods, overriding, et cetera.

7.1 - Mother class development

Since the requirements include the possibility to enrich the list of Machine Learning related modules, I decided to implement a common superclass, *objectClass*, that would take care of changes in the attributes, draggability and connectability, configuration panel and so on. These are just some of the attributes and methods this class has:

attributes

DOMObject - the reference to the element appended in the HTML
configurationPanel - an array of input fields

methods

modify - modifies the given attributes and, if it is the case, send info to the server
setValueOfDataProperty - invoked when a data property is modified
connectedAsSource - triggers when this instance is connected as a source

As the requirements state, I made as simple as I could to add new Machine Learning related

modules. I have prepared some example classes where I showed what was to configure (i.e. configuration data parameters, the number, type and position of input and output areas). Moreover, the inheriting classes can take advantage of the methods of the mother class, such as *connectedAsSource* or any of the other functions by just overriding them. Here one of those classes is reported.

```
// Class DataSplitter
// split the received data into "training" | "test" dataset
var DataSplitter = objectClass.extend(function (parameters) {

    // here the JSON of object parameters
    // just add more pair "paramID":"attributes" if need to
    this.configurationData =
        {"percentage":{"label":"Training set percentage", . . .}};

})

.statics({
    // the class name
    className : "DataSplitter",

    // there are at most 9 anchors available:
    // Top - TopRight - Right - BottomRight - . . .
    inputs : [{ anchor:"Left", maxConnections:1 }],
    outputs : [
        { anchor:"TopRight", maxConnections:MAX_INTEGER },
        { anchor:"BottomRight", maxConnections:MAX_INTEGER } ]
})
```

7.2 - Dynamic class instantiation

The blue toolbar at the bottom of the page contains one icon for each Machine Learning module available. This list is received within the “serverObjectsResponse” message as a JSON containing the names of these classes. Then there’s a loop, and for each element its ** is created, bounded to a click event listener and lastly appended to the DOM with ID equal to its class name.

The icon is contained in the *classes/cons* directory, and has the same name as the class of reference.

Eventually the JS class file of reference is loaded.

```

// create icon through the utility method "shapeHTMLElement"
var objectIcon = shapeHTMLElement(
    'img', {"id": className,
    "src": "classesIcons/" + objectClassName + ".png"
});

// click event listener, in order to create the object
objectIcon.addEventListener("click", function() {
    createObject(this.id, {"objectClass" : this.id}, 2) });

// append the element to the DOM Toolbar
DOMToolbar.appendChild(objectIcon);

// load the relative js file
loadJSFile(("classes/" + objectClassName), function() {

    // another file has been loaded
    loadedFile++;

    // if all files has been loaded
    if (loadedFile == objectClassArray.length)

        // set the boolean to true
        loadedAllJSClassesFiles = true;

        // restart restoring the previous status
        resolveMessage(storedMessage);

});

```

As said before, a click listener is bounded to each icon, and allows the dynamic instantiation of objects. This is achieved by calling a util function, *createObject*, with only one parameter, the ID of the clicked element. The function updates some state variables and simply calls the selected class constructor method.

```
var obj = new window[objectClassID]();
```

Lastly, it sends the notification that a object has been created to the server.

Note: Only when all the JS classes are loaded, the interface can proceed to restore its previous status, if any. This is achieved by stopping the “serverJSONResponse” message handling if these files are not ready yet, and by storing it aside. Then, when the loading is completed, just call the “resolveMessage” function.

Chapter 8 - Example Class Implementation

Lastly, for illustrative purposes, I implemented the code and logic for a SQL database module. This is the procedure I followed:

1. I created a class named *SQLDatabase* and set its needed parameters in order to successfully connect to a such online resource and fetch the data:
 - database host url
 - host application port
 - username
 - password
 - database name
 - query to perform

Then added its icon in the *classes/cons* folder

2. I invented in the documentation two other types of messages, one from client to server (*querySQLDatabase*) and one from server to client (*queriedSQLDatabase*), and added the case in the switch statements
3. Server side, I implemented the code that actually connects to a SQL remote database and fetches the data
4. Client side, I overrode the function that handles the configuration panel in order to show the data received from the servers.


Discursively, once the user completes the configuration of the object, he can send the parameters by using the proper button. The information is encapsulated into the message of type “querySQLDatabase”. The server receives the data, that are then used to perform a connection using the **mysql** module of **Node.js**®. Since the server can receive multiple requests, a queue of connections is prepared to contain them.

```
// create a connection with the given parameters
var connQueue[session][objectID] = mysql.createConnection({
  host      : url,
  user      : username,
  password  : password,
  port      : port,
  database  : dbName
});
```

After the handshake, the query is executed on the remote hosting server, and matching data are returned to our server. Finally, either there was an error along the process or not, it will send the

result to the client through the “queriedSQLDatabase” message. The client will alert the proper message and, in case of success, will append the received data at the bottom of the configuration panel.

The server will store the retrieved data until either the SQL object of reference is deleted or the last client disconnects from the session.



SQLDatabase 1




Query to execute

```
SELECT * FROM iris
```

send configuration

delete object

5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa

1

I prepared an online SQL database hosted by [freesqldatabase](#) and created a table “iris” with data from [UCI](#) Machine Learning repository.

Chapter 9 - Conclusions

At the end of the process, all the previously agreed requirements were achieved: the interface allows multiple users to design together graphs, creating objects and connections between them. Most important thing is that all was designed to be as much as modular was possible, so that it would not be difficult to add new classes, message types and whatever there will be to be integrated.

Note: obviously, even if it's easy to add new Machine Learning related modules to the interface, their functionalities server side have to be implemented, like as the message types that will wrap their parameters.

Thanks to this modularity, the interface can now be the base for creating a complete system that I strongly hope might be a serviceable tool for orchestrate Machine Learning elements. But, even if this was the purpose driving the development, there must be considered the fact that the foundations of the interface are independent from the context of use. Thus, they should not be limited only to this purpose, but they can provide a base from which build any kind of system based on graphs, flows of execution and user intercommunication in real time. The mechanisms that are implemented aren't matter-of-fact bounded to anything, they just fulfill the requirements mentioned and discussed all along this thesis.

I have used three MIT libraries that other developers implemented and released for free. I hope that from the basis I built can rear something good to repay and thank the open-source environment.

To summarize the thesis with numbers:

- 4 activity diagrams
- 2 sheets of architecture and designing
- 3 MIT libraries used
- 2 CSS files
- 2 TPL files
- 17 JS files
- 3500~ lines of code, including comments

Sitography

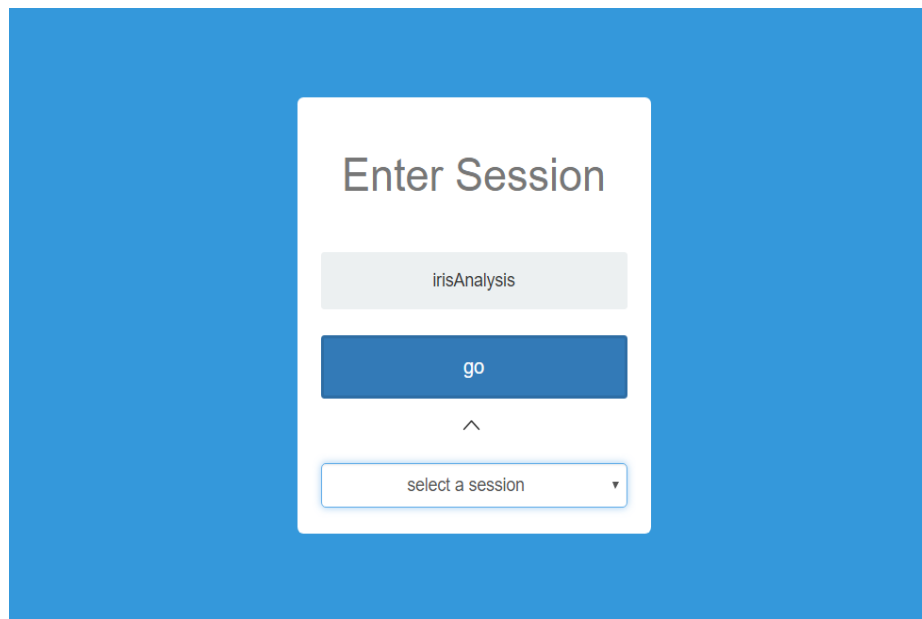
Note: since this thesis is not much of a research, but instead a software development, the only resources I exploited were my university notes and the websites related to the libraries API and tutorials, besides some excursions to Stack Overflow or W3Schools .

Site name	URL
<i>JSPlumb</i>	https://jsplumbtoolkit.com/
<i>klass</i>	https://github.com/ded/klass
<i>Node.js</i>	https://nodejs.org/it
<i>SweetAlert</i>	http://t4t5.github.io/sweetalert/
<i>Stack Overflow</i>	https://stackoverflow.com
<i>W3Schools Online Tutorials</i>	https://www.w3schools.com

Attachments

attachment A - Interface Screenshots

Choose Session

A screenshot of a web interface titled "Enter Session" centered on a blue background. The interface is a white card containing a text input field with "irisAnalysis", a blue "go" button, an upward arrow, and a dropdown menu labeled "select a session".

Enter Session

irisAnalysis

go

^

select a session ▼

Workbench

The screenshot displays the SQL Workbench interface. On the left, a diagram shows a data source icon connected to a central node, which then branches into two paths, each leading to a gear icon representing a database connection. On the right, a configuration panel for 'SQLDatabase 1' is visible. It contains several input fields for database connection details, followed by buttons for 'send configuration' and 'delete object'. At the bottom, a blue toolbar contains icons for a gear, a double-headed arrow, and a document icon, along with a page number '1'.

SQLDatabase 1

Data Source Name
sql11182476

Data Source URL
sql11.freesqldatabase.com

Data Source port
3306

DataBase username
sql11182476

DataBase password

Query to execute
Select * from iris

send configuration

delete object

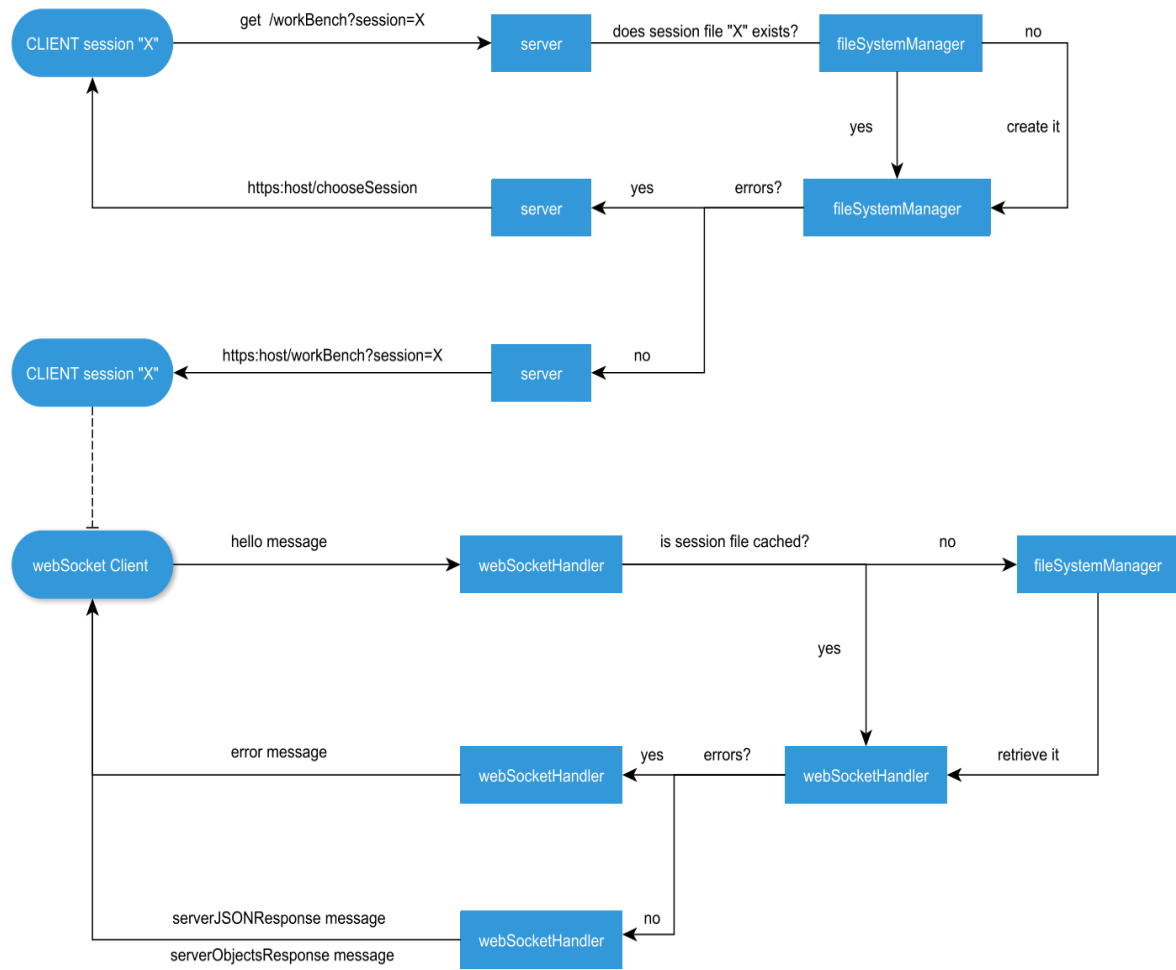
1

attachment B - Design and Architecture

Message Exchanging Definition

SOURCE	BROADCAST	MESSAGE TYPE	MEANING AND AFTERMATHS	JSON MESSAGE PARAMETERS
CLIENT	Y	hello	notify the client connection and initialize session	{sessionName : currentClientSession, messageType : messageType}
SERVER	Y	bye	notify a client disconnection	{sessionName : currentClientSession, messageType : messageType}
SERVER	N	serverJSONResponse	respond with the JSON file the client that said "hello"	{messageType : messageType, JSONFile : JSONFile, clientsConnected : numberOfClientsConnected}
SERVER	N	serverObjectsResponse	respond with the list of available drag'n'drop objects	{messageType : messageType, objectList : drag'n'dropObjectList}
CLIENT	Y	addObject	add an element with the given parameters	{sessionName : currentClientSession, messageType : messageType, objectClass : objectClass, objectID : createdObjectID, DOMParameters : { objectClass : objectClass, id : createdObjectID, x : xValue, y : yValue, ... }}
CLIENT	Y	modifyObject	modify the object attributes	{sessionName : currentClientSession, messageType : messageType, objectID : modifiedObjectID, DOMParameters : { attr1 : attr1Value, ... }}
CLIENT	Y	deleteObject	delete the element and all it's connections	{sessionName : currentClientSession, messageType : messageType, objectID : deletedObjectID }
CLIENT	Y	addConnection	add a connection to the given elements	{sessionName : currentClientSession, messageType : messageType, objectClass : objectClass, objectID : createdObjectID, DOMParameters : { objectClass : objectClass, id, createdObjectID, sourceObjID : sourceID, sourceAnchorType : anchorType, targetObjID : targetID, targetAnchorType : anchorType }}
CLIENT	Y	deleteConnection	delete a connection	{sessionName : currentClientSession, messageType : messageType, objectID : toDeleteConnID }
SERVER	N	errorMessage	error message server sended	{messageType : messageType, errorMessage : errorMessage }
SERVER	N	undoMessage	force the client to undo the action	{messageType : messageType, actionToUndo : actionToUndo, ObjectID : relatedObjectID }
CLIENT	N	querySQLDatabase	request to query a database and get the data	{messageType : messageType, databaseSourceURL : databaseSourceURL, databaseSourceName : databaseSourceName, databaseSourcePort : databaseSourcePort, databaseUsername : databaseUsername, databasePassword : databasePassword, queryToExecute : queryToExecute }
SERVER	N	queriedSQLDatabase	wraps the requested SQL database data	{messageType : messageType, applicantObjectID : applicantObjectID, data : data }

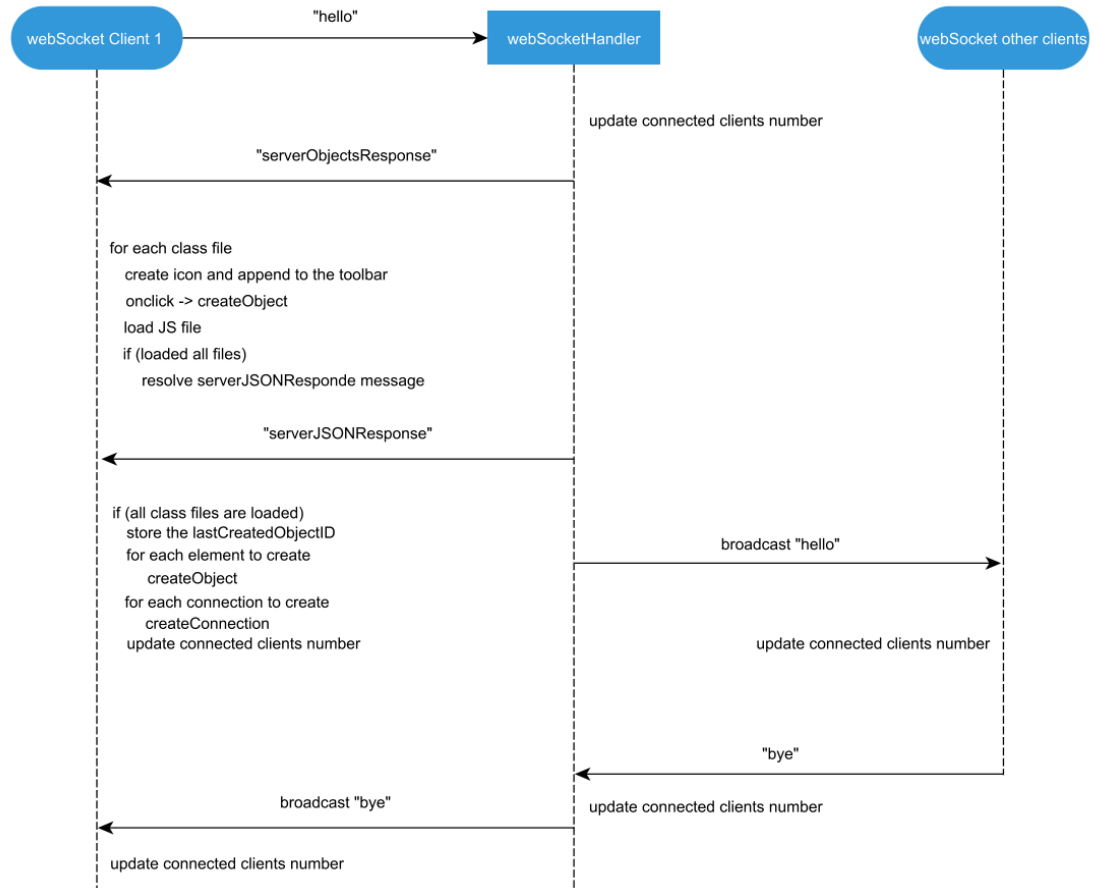
User Connection Flow



First Connection message exchanging Activity Diagram

MESSAGES HELLO AND BYE

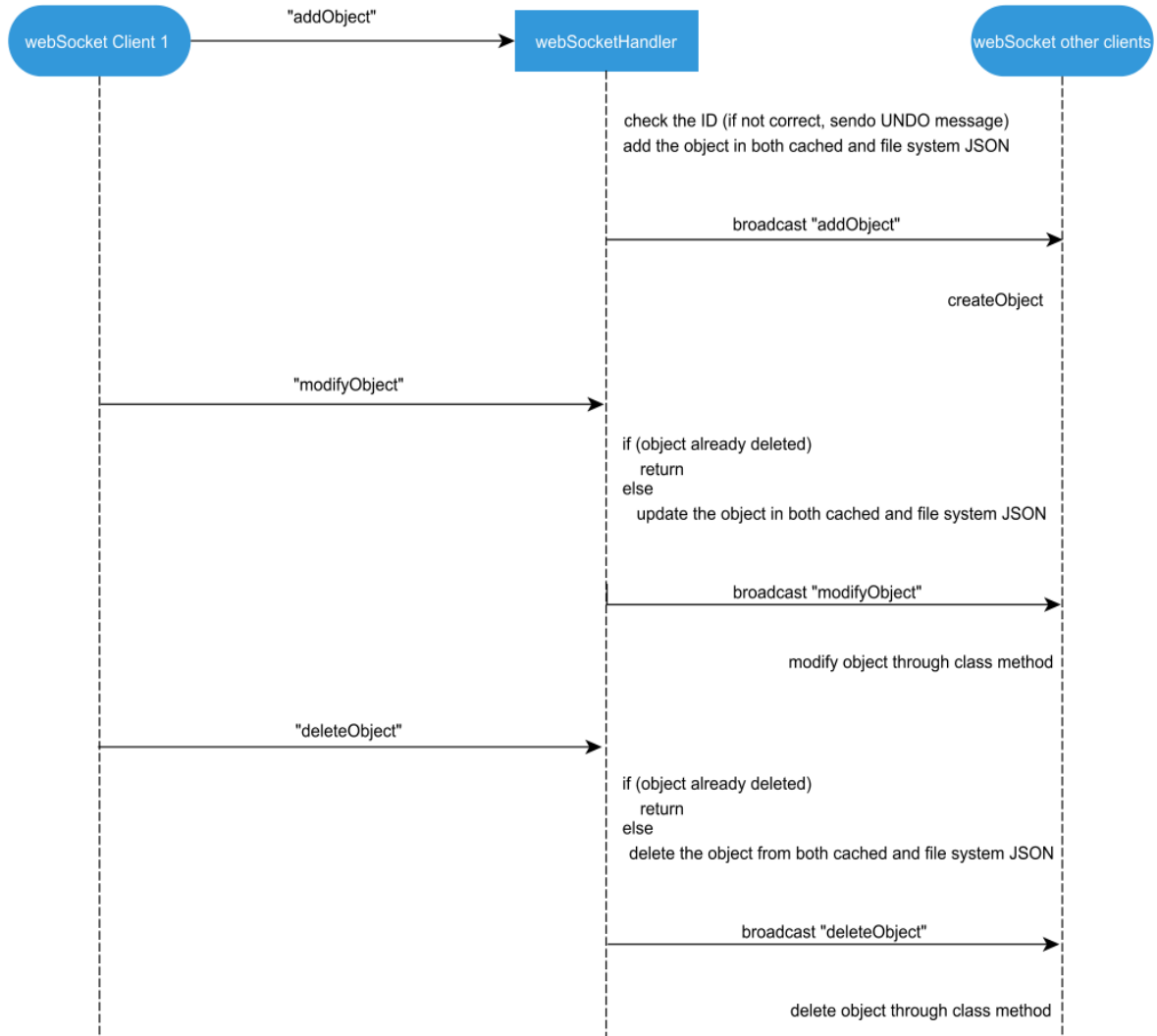
all is intended to be referred to a specific session equal for all clients



addObject message Activity Diagram

MESSAGES REGARDIN OBJECT

all is intended to be referred to a specific session equal for all clients



addConnection message Activity Diagram

MESSAGES REGARDING CONNECTIONS

all is intended to be referred to a specific session equal for all clients

