Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2014

Diogo Santos Pimentel

SOLUÇÃO WEB PARA SUPORTE DE SISTEMAS EM TEMPO REAL

WEB SOLUTION TO SUPPORT REAL TIME SYSTEMS

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2014

Diogo Santos Pimentel

SOLUÇÃO WEB PARA SUPORTE DE SISTEMAS EM TEMPO REAL

WEB SOLUTION TO SUPPORT REAL TIME SYSTEMS

"*The greatest challenge to any thinker is stating the problem in a way that will allow a solution*"

— Bertrand Russell

Universidade de Aveiro Departamento de Eletrónica,
2014 Telecomunicações e Informática

**Diogo
Santos Pimentel**

SOLUÇÃO WEB PARA SUPORTE DE SISTEMAS EM TEMPO REAL

WEB SOLUTION TO SUPPORT REAL TIME SYSTEMS

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Helder Troca Zagalo, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor José Alberto Fonseca, Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus pais e irmãos pelo apoio incondicional e dedicação. Dedico também aos meus tios pelo suporte dado diariamente.

agradecimentos /
acknowledgements

Agradeço a todos os meus colegas na Micro I/O em especial ao Paulo Rocha e Carlos Franco pela orientação disponibilizada. Agradeço também aos meus orientadores pela oportunidade e pela disponibilidade para ajudar.

Palavras Chave

Comunicação em Tempo Real (RTC), Aplicação Web, Websockets, Integração Browser e Hardware.

Resumo

Cada vez mais qualquer terminal se comporta como uma janela para a web, seja computadores, *smartphones*, *tablets*, etc. Com o advento das tecnologias web e da globalização da mesma, cada vez mais os fornecedores de serviços se têm de adaptar à nova realidade e acompanhar a tendência para não perder clientes e mesmo a qualidade de serviço. Cada vez mais se tem verificado nas ferramentas que se utilizam no dia-a-dia, essa evolução no sentido de se tornarem aplicações web. Quando empresas líderes no seu segmento de mercado se propõem a fazer esta evolução nos seus produtos e serviços, vários fatores têm de ser estudados e tidos em conta. Tratando-se de soluções com comunicações em tempo real integradas com *hardware* próprio, várias soluções se propõem a resolver o problema, cada uma com os seus prós e contras, dando prioridade às tecnologias de ponta, que, de preferência sejam desenhadas de origem para responder às questões que se levantam no contexto do produto. Daí ser prioritário um estudo que responda às questões relevantes para o produto bem como os necessários testes que validem a teoria proposta.

Keywords

Real Time Communication (RTC), Web Application, WebSockets, Browser and Hardware Integration.

Abstract

More and more every terminal behaves as a window to the web, being computers, smartphones, tablets, etc. With the advent of web technologies and its globalization, more and more service providers have to adapt to the new reality and follow the trend in order to not loose clients and even the quality of service. This evolution towards web applications has been increasingly observed in the tools used in day-to-day tasks. When leading companies in their market segment propose to make this technological evolution in its products and services, several factors must be studied and taken into account. In the case of solutions using real time communications integrated with its own hardware, several solutions propose to solve the problem, each with its pros and cons, but giving priority to the cutting-edge technologies, which, preferably are originally designed to answer the matters in the product context. Hence a priority to a study to answer the relevant matters for the product as well as the necessary tests to validate the proposed theory.

Contents

List of Figures

List of Tables

Acronyms

HTTP	HyperText Transfer Protocol
TCP	Transmition Control Protocol
SOAP	Simple Object Access Protocol
CAN	Controlled Area Network
HTML	HyperText Markup Language
DOM	Document Object Model
URL	Uniform Resource Locator
NAT	Network Address Translation
SSL	Secure Socket Layer
TLS	Transport Layer Security
JSON	JavaScript Object Notation
XML	Extensible Markup Language
SQL	Structured Query Language
IIS	Internet Information Services
JDK	Java Development Kit
JRE	Java Runtime Environment
SPA	Single Page Application
API	Application Programming Interface
RPC	Remote Procedure Call
SMS	Short Message Service
PoS	Point of Sale
DB	Database
IP	Internet Protocol
UI	User Interface
USB	Universal Serial Bus
MVC	Model View Controller
RFID	Radio-Frequency Identification
CPU	Central Processing Unit
COM	Communication Port
RAM	Random-Access Memory
UART	Universal Asynchronous Receiver/Transmitter

CHAPTER 1

INTRODUCTION

It has been a trend throughout the years, as the internet growth becomes a never ending phenomenon that most applications are redesigned to be web applications and new applications are built as web applications from the start. Some of this trend has been pushed by some big names in the technology business (such as Google) when they announce operating systems that are built to be web client operating systems, working almost like a web browser and connecting the user to its various web applications [1] - later this approach was revised to allow the use of some native applications and the possibility of some of the web applications to work offline seeing that constant internet access is still an issue. This trend has some very logical advantages and foundations both to client and service providers that are the root of its own existence.

1.1 MICRO I/O

From the company website[2]: *"Micro I/O - Serviços de Electrónica, Lda. is an enterprise that creates integrated software and hardware solutions for its own products or for specific custom applications. With a strong R&D and innovation activity, in connection with Universities in Portugal and Europe, Micro I/O uses cutting-edge technology in the development of products and services, fitting the customer's needs."*

One of the products developed by Micro I/O, is a desktop application for school management, that has the largest client quota in its segment. When a product in the market is established as one of the best in its segment and its usage is growing, it takes a lot of careful thought to decide the next technological step to take. With the growing market in mobile applications and users of mobile devices with online connection almost all the time, it is a possibility that the next step should be towards the mobile segment. But mostly because of mobile operating systems constraints it was dropped and the web application took its place with the advantage that it can be used in mobile devices as well desktops and other equipment that was already running the current versions of the software.

For this step to be taken by Micro I/O, it took serious consideration to the market trend discussed previously and other really critical aspects to this particular product and its clients.

1.2 MOTIVATION

In the client perspective one of the main advantages is data safe storage. The failure of a hard drive, Universal Serial Bus (USB) drive or other means of physical storage has always been a problem that a user encounters once in a while and has few options regarding its solution. If it can be recovered, it is an expensive procedure that is only relevant in the case of highly sensitive or important data, or if it can be reproduced the user will have to spend the time to redo the work that was lost, but sometimes it is not possible to reproduce and the cost of recovering is not affordable. But if the user is working on-line, their data has reduced risk of being lost due to hardware failure (or user lack of skills), because now their data is being created, edited and stored in cloud services where safe mechanisms are one of its main goals.

Another important advantage is hardware requirements and cost savings. The client does not have to be running its applications on high performance hardware to have access to heavy duty software or just a smooth user experience of some applications, which used to have great system requirements that were set when the application would execute locally, and the meaning of all this is that with "less" hardware (less money spent) and a good internet connectivity it is possible to have the benefits of high performance hardware, or even better, now that the machines used to run the web applications are usually clusters with a processing power several orders of magnitude higher than the best single machine in the market. These cost savings are considerably higher if we take into account that in most cases the operating system constraint is no longer an issue, and it is not required to run a certain operating system with its added costs to have access to the same applications – and this is another point of discussion: operating system independence.

Regarding the operating system independence to the service providers and the creators of these applications, the advantages rely mostly on the interoperability that these methods provide. Because now the user only needs a browser to run their software and browsers are available across operating systems, the concern about what target operating system is used to build the software is no longer an issue. This opens the door to a market of users that in the past had the constraint of using an unsupported operating system by their software. The main concern now is if the application will run on most of the web browsers available, or in what versions of them it will be possible to use their software. The need for developers to specific operating systems and the possible division in developer's teams is not an issue, having the teams efforts focused to the same target.

Another point that is a controversial topic in some areas where web applications are widely used – social networks for example – is that the service providers will know what the users are using their software to, and in some cases this type of information has been used to make profit by selling it to third parties. This type of knowledge is also important when the service provider uses it to provide a better service, upgrading the software to make up for the lack of some features that come up as the users interact with their software, and thus becoming a much competitive service always innovating. When this information is available it can be used in recommender systems where the information is compared throughout users to find similarities and patterns allowing the recommendation of some product or feature to its users. And finally there is one great point in favour that is not always recognized but took a great weight in the decision making for this platform to be developed: the possibility that all clients are using the latest version of the software. It may seem a trivial point but it will be discussed why it was such a big weight in the process.

One of the concerns about the clients is that the terminals used to operate are not up to date as expected and that means that some are still using old operating systems that have no more support and to upgrade them means that most of these terminals will most likely not be able to because of the minimum requirements, and new equipment is not always a possibility. Another issue is the software licensing, seen that the current versions run on Microsoft© Windows© and that every terminal has to have licensed software takes the cost of the system up just to be able to use the software. With this step, the clients can now use other operating systems namely free operating systems, reducing the cost for every machine that is upgraded. This gives the possibility of offering a better service for a lower cost to the customer who can now choose to use free software in old machines and with it “bring to life” some terminals that can now run the new software.

1.3 GOALS

The possibility to have all clients running the same version of the software – referred above as one great weight in this decision – is of great importance to Micro I/O in terms of customer support, staff organization and resource savings. The current version of the product is installed in over 500 clients (schools) and each one is responsible for the update of the software as it’s released in the client web page. The process of updating the software is not a complex task but it is often neglected, because either there is no person responsible for the task, or the person responsible does not have the skills to perform it, or simply forgets it, and when there is a problem with the system (or seemingly in the system) the client contacts the support line who has to check which version of the software is running in that particular client, and perform an update remotely if possible or guide the user through the operation. The resources needed to have this support available are great and it is time consuming to the employees that would provide a better service in less time if they knew that the user calling is using the latest version of the software. With this change in technology it would be possible to have all servers update from the same location automatically, keeping schools up to date for as long as the update contract is valid, saving time and resources to really work on customer support that now mostly consists on updating clients’ software.

Combining this possibility to redesign the technical support system of the company (providing a better customer service) and saving costs on new products to sell by reducing on operating systems licensing, the technological advance towards this direction became not only important but the logical next step to take.

So the proposed goals for this thesis were to design and develop a modular application, independent of the operating system on the client side that could support communications between application and possible hardware connected to the terminal. This coupled with the design and development of a web application – that makes use of a predetermined Single Page Application (SPA) architecture – to support real time operations, working side by side with the application referred above to interact with possible integrated hardware.

CHAPTER 2

State of the art

In the current web scenario, real time communications is taken for granted by the users who chat with each other, video-conference, watch live streaming, and so on and so forth. This seamless real time communication is based on various techniques, evolving through time as new technologies unveil. Tracking these evolving techniques is a challenging task and to this point, the conclusion is to point out the most used and known to the developers.

2.1 "real time" for web applications

There wasn't always the notion of "real time" for the web. As the use for the internet evolved to what is known today, many applications and uses were brought up, and so was the need for new types of communications aside from the usual client-server context, where the client asks for information whenever it wants. There was then a demand for a way to send information to the client without him having to ask for it, as in chat applications.

Because the protocol "for the web" and the base for all communications mentioned is HyperText Transfer Protocol (HTTP) and it does not offer – from the beginning – a possibility to make the type of interaction needed to address these issues – that had the first documented version in 1991[3] – so there were developed methods and techniques to work with the protocol and with its capabilities to make it work.

The first approach to this problem was in 1996[4] when it was possible to create and sustain a persistent connection with the web browser with the help of a *Java applet*. This was made by having Java applets embedded into the browser to give the ability of a real time communication using *Transmition Control Protocol (TCP) sockets* which stays open as long as the client stays in the document hosting the applet, and the server would use this socket to send notifications translated later by the applet. This solution did not go on its full potential forward both by the need of a plugin installed in the browser, as well as the lawsuit opposing Sun to Microsoft that created a division among developers who then had their software built for Microsoft's© Internet Explorer® 4.0, or built with Sun's Java Development Kit (JDK) 1.1 that would run on any other browser[5].

Of course that by using HTTP - which is inherently a request-response protocol - lead to the use of its communication capabilities: by forcing the client application to send requests with a defined time interval it would get the server to respond with updated data, i.e., the client application asks for new data for as long as it was needed (*Polling*). This is the most simple and most common technique, used to have "real time" in web applications. It had many evolutions with a few tweaks always trying to get data to the user more frequently with less effort to the servers, network, and less complexity for the developers.

2.1.1 POLLING VERSUS LONG POLLING

One of the first techniques used to create the likeness of real time communication, and easy to implement is *Polling*. It can be simply put as sending *HTTP requests* to the server in fixed intervals in order to obtain a response.

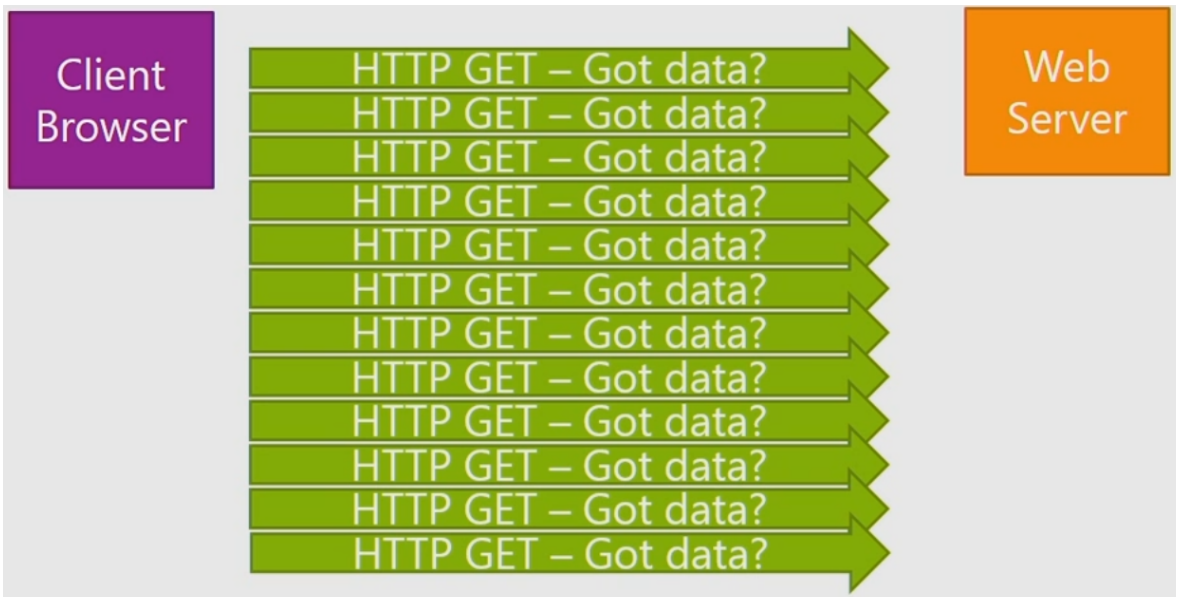


Figure 2.1: Basic Polling scheme[6]

To create the effect of real time, the time interval must be decreased to be really small and thus getting data as fast as possible. This has an important disadvantage: it creates large amounts of requests to the server and network load (*HTTP overhead*).

```
function doPoll() {
    $.post('ajax/newData.php', function (data) {
        alert(data); //process new data here
        setTimeout(doPoll, 1000);
    });
}
```

In the snippet above, it is shown how to make a simple poll to an Application Programming Interface (API) on the server, requesting new data from *ajax/newData.php*, with an interval of 1 second (1000 milliseconds). The code is embedded in the webpage HyperText Markup Language (HTML) as a JavaScript script and processed on the client side.

One improvement to this approach is *Long Polling* which consists on sending the same *HTTP requests* less frequently, but then the server will hold the request and only send the response when it has new data available, completing the open *HTTP request*[7] – new requests will be sent if timeout is reached before new data is available. After a response is returned, a new request must be sent in order to keep the connection "alive". This reduces the network load compared to the previous and data is received by the client just as it is available[7] – reduced latency.

```
function doLongPoll(){
    $.ajax({
        type: ``GET``,
        url: ``newData.php``,
        async: true,
        timeout: 5000
        success: function(data){
            //process new data here
            setTimeout(``doLongPoll()``,1000);
        },
        error: function(XMLHttpRequest,textStatus,errorThrown) {
            setTimeout(``doLongPoll()``,1000);
        }
    });
}
```

Much like the previous sample of simple polling, this code sample shows how it is possible to make a *HTTP GET* request from the server, but with the small change that this time the request has a timeout of 5 seconds (5000 milliseconds), meaning, it is alive for 5 seconds awaiting for new data to be available. If the request timeout expires, the error function is invoked and it simply calls the polling function again, and restart the cycle. This is once more on the client side and whit this small change in approach, it is simple to understand the advantage of having few requests to the server and how the networks load can decrease significantly just by extending the *HTTP request timeout*.

With the long polling technique, it's more likely that new data will be sent to the client just when it is available, by having unanswered requests on the server, waiting to be replied to. This clearly improves the real-time perception of the communication, and allied with its implementation simplicity makes it a strong choice of technique to use in most cases, and justifies why it was broadly used.

Of course this is just one more tweak to a technique that was originally flawed and has as base a protocol that was not designed with this intent, and with the growth of the number of client connections to a server this is unsustainable, because for each client there will be a thread running on the server waiting for response.

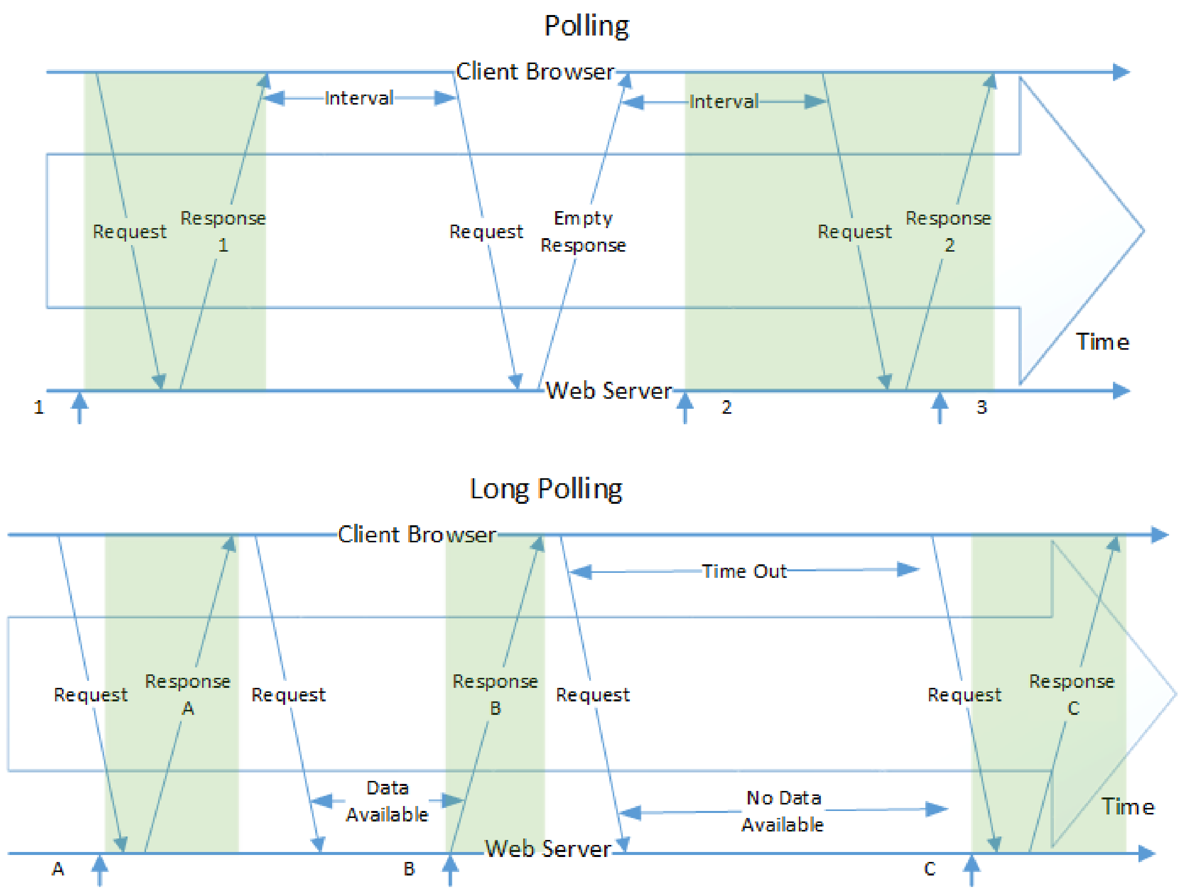


Figure 2.2: Traditional Polling versus Long Polling

The Figure 2.2 allows for a visual perception of what has been explained before, it shows how the two techniques would behave for an example of interactions (the labeled up arrows represent new data available at the server side ready to be sent). Having that the start of communications begins with the time arrow representation, data “1” and “A” have almost the same interval from availability to delivery (green areas) because both techniques would send the same request in the beginning. Only after the first data delivered to the client the differences start to show, on traditional Polling the client will wait the time interval to send a new request and if no data is available the empty response will follow; during the next time interval data “2” is available to send but there is no request for a while and when data “2” is delivered to the client there is data “3” already waiting for a request that will not occur in another time interval. For the events represented in “B” and “C” the time elapsed, starting when data is ready and ending on its delivery, is almost the same compared to data “A” even for distinct phases in the process: in data “B” the request arrived some time before and is still possible to send the response, so data will be delivered immediately; when data “C” is ready, a time out has occurred on the previous request but a new request will arrive soon after and the data will be delivered with a small delay.

2.1.2 FOREVER FRAMES

Used in Internet Explorer® for a long time, this technique works by creating a hidden *Iframe* in the page that requests a "document" from the server but what the document is, is a collection of scripts that the browser will execute as they arrive. This document is "never ending" (*long lived HTTP connection*), the request is kept alive and the server will keep sending script tags creating a seamless persistent connection[6].

This technique is possible because of a feature in the HTTP 1.1 specification, *Chunked Encoding*[8], that allows a server to divide the response in a series of chunks each with its size associated, and the total size of the response might be unknown during this process. It was made use of this specification to have servers sending data without revealing its total size and thus keeping the connection alive allowing the push of chunks of data to the client.

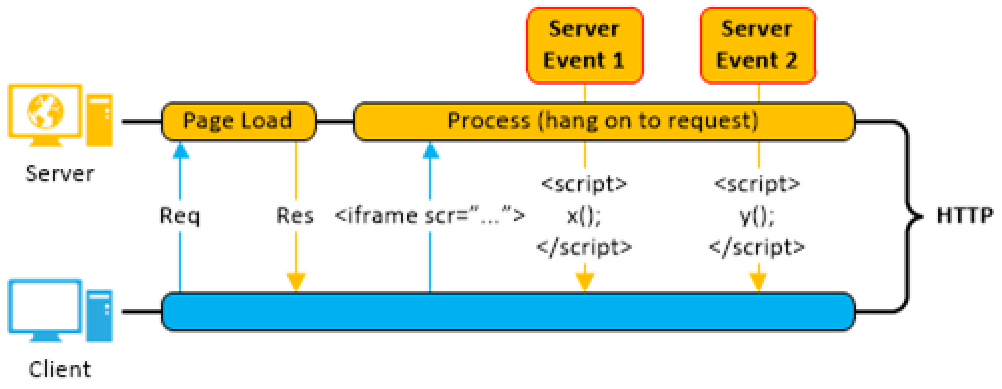


Figure 2.3: Forever Frame Example[9]

The *Iframe* element starts the connection with the server as soon as the browser finishes loading the page. Then the chunks would begin to arrive – as new data is available at the server – and the script contained would be executed with the use of a function call from a library in the parent document.

Data chunks received are rendered incrementally by the browser and added as nodes to the *Iframe* by the Document Object Model (DOM) causing the document size to grow. There could be some memory issues with the non-ending responses that would make the *Iframe* increment its size, but as the chunks finish rendering they are typically removed. Not all versions of the browser react the same way to this technique and some tweaks are required to have it working with the various targeted[10].

2.1.3 SERVER-SENT EVENTS

Supported by almost every modern browser – other than Internet Explorer® – this standard was designed to be efficient, and it stands out because the server can now send notifications to the clients whenever it wants without the need of the initial request. The client now subscribes to an event stream defined by the *Event Source* API which opens a HTTP connection to receive push notifications from the web server[11]. These push notifications are received in the form of *DOM events*, and that's why it is said that the client subscribes to an event stream, because by using this API and connecting to the server the client is actively a recipient of the series of events published by the server.

```
var sourceStream = new EventSource(``example_sse.php``);

sourceStream.onmessage = function(event) {
   //sample process of an envent
   document.getElementById(``result``).innerHTML += event.data + ``<br
       >``;
};
```

This is the most basic example of how to make the subscription to an Event Source and handle the events push by the server. The first line is how the subscription takes place, creating a source stream from the target location. Then it's needed to tell the browser how to deal with the new events (*onmessage* signal), and that's what's been done in the definition of the handle function that takes the event received in the stream and – as an example – adds it to the HTML element *result* as a new line.

2.1.4 WEBSOCKETS

The real "game changer" in terms of real time connections for the web is WebSockets: persistent full-duplex channel over a TCP connection. It's not just another tweak or workaround to HTTP, it's a protocol[12] designed specifically to allow asynchronous and bi-directional communications between the server and client applications[13].

To supersede the mentioned technologies, WebSockets had to be designed to escape the HTTP request-response premise. In order for that to be accomplished a simple solution would be to create a TCP connection and use it to bidirectional traffic, which was the solution implemented in the WebSockets protocol. HTTP is used for the handshake negotiation which is interpreted as an *Upgrade Request*[14] by the server and the TCP communications are made over the ports 80 or 433 to make use of existing HTTP infrastructures – this is in itself an advantage, having traffic over those ports ensures that the traffic will be treated as HTTP[6].

The improvements are a big step towards the ultimate goal: sending data to the client whenever there is new data, wherever the client is (independent of network, device or browser) and with as low latency as possible. Because of the use of a TCP persistent connection, server and client can send data at any point in time, and with the use of the ports 80 and 433 to communicate traffic will be having the same reach of HTTP traffic passing through firewalls and proxies which block "non-web" traffic[6] (traffic other than HTTP). By not being exclusively implemented in web browsers and servers, it can be used in other applications and so this broadens the scope of uses for this protocol. Noteworthy is the fact that the data sent back and forth is really just data, it's not scripts or HTML elements, but the raw data that meant to be delivered.

```
var connectionWS = new WebSocket('ws://example.org/ping');

connectionWS.onopen = function () {
   //send a string through the socket
   connectionWS.send('ping');
};
```

```
connectionWS.onmessage = function (e) {
    //process received data
    console.log('Server: `` + e.data);
};
```

This sample code shows two important things about the protocol: the simplicity of creating and use this connection and the direction of the data flow. On the first line the connection is established by creating an instance of a WebSocket pointing to the server location – the *Uniform Resource Locator (URL) schema* for WebSockets is "*ws*" as shown or in the case of a secure connection using Secure Socket Layer (SSL), "*wss*". Then in the second code block, it's established that when the connection opens the client will send a message represented by the *string* "ping" just as an example of how upstream connections are processed. Then whenever a new message is received, the signal *onmessage* is emitted and the data is processed – logged in this case.

Because this is a point to point connection, if the application has multiple pages (very common) then each page represents a different stream of data, and it's the server job to manage these transfers: there must be a collection of connections (each one representing a client) and the knowledge of which data to send to the client (what page is he connected to). And so the server must iterate through the collection and send data to the respective clients[10].

Another scenario in which WebSockets scale-out becomes complex is when the application allows the clients to interact with each other. In this case the same iteration through the collection of connections has to be repeated, in order to send data each time a client "talks" to another.

Scaling this technology is the difficult part; the last two points show simple examples of how this starts to get complex as the application grows or the number of client grows, but there is another issue with a common situation these days: multiple servers, or *web-farms*. In these configurations clients of the same application can be connected to multiple servers and at the time of distributing data, the servers must keep an updated list of active connections to them and convey that information to other servers so that their clients get the up to date data as is expected. To implement these type of configurations, all the logic behind the data exchange and the clients connections is usually implemented by an extra entity. That entity can be a server, a variety of services implemented in the application servers, or both. It contains a list of all clients connected to the application servers and their location, which the servers constantly update, meaning extra tasks to the application servers and increased overall complexity.

2.2 WEBSOCKETS FRAMEWORKS

As discussed WebSocket is the latest technology regarding real time communications for web applications, and so it will be presented the best frameworks to aid the development, regarding browser compatibility, operating systems supported, methods to solve the proposed goals, and documentation with active development if possible. The presented frameworks are all implementations for the .NET framework. To better understand each framework in the developer perspective, a simple chat was built for each presented framework, and the important elements of its code can be seen in Appendix B.1

2.2.1 POKEIN FRAMEWORK

It's a framework developed by Zondig[15] with support and documentation depending on contract and an yearly fee. The free version of this framework cannot be used with commercial means. Has a variety of features including message level encryption (even in an open HTTP connection it is possible to cipher the message transmission), "clone message attack" protection mechanisms (detects and discards cloned or repeated messages) and session cloning detection (prevents attackers from using established sessions to be reused)[16]. The range of browsers and software support is great but at a high cost (only available in paid versions). Because it also has *Mono* support, it was considered a possibility, but some issues such as the need for PokeIn developers" team to intervene in some type of implementation scenarios, it was ruled out in the end.

2.2.2 SUPERWEBSOCKETS FRAMEWORK

It's an open source framework, built over the SuperSocket framework by the same community – which is a widely used framework within its type of implementations – and with *Mono* support as well as SSL/Transport Layer Security (TLS). Has tools for scale out scenarios and support for Microsoft© Windows© environment hosting. Although it seemed to work in the test application (chat application), the lack of documentation and community support, added the fact that it never released a stable version (the latest being 0.8 beta) caused it to be left out of the solution.

2.2.3 ALCHEMY WEBSOCKET FRAMEWORK

Another open source framework, with community and Olivine Labs[17] team support, with an extensive and complete documentation. It has *Mono* support as well as all frameworks considered, and can be downloaded directly from *NuGet* into the Microsoft© Visual Studio© project. There were some problems when it came to disclosure on what features it had implemented and was not possible to have a list of that features unless they were tested.

2.2.4 ASP.NET SIGNALR FRAMEWORK

Started as a side project from its creators[10] – who are part of the ASP.NET developing team – soon became popular and was added to the ASP.NET platform. It is a framework used for real time communication on web applications. The concept goes beyond the last discussed frameworks that are exclusively WebSockets frameworks, because it relies on several technologies to push data from the server to the browser in real time. It has a more diverse set of features and capabilities.

Although it belongs to the ASP.NET platform and has the support behind the team, it is still an open source project as it was created[18]; the community can still contribute to its development, make suggestions to ASP.NET developers for changes or improvements; it has *Mono* support even after the adoption by ASP.NET allowing its usage outside Microsoft operating systems. It is available for download at *NuGet*, facilitating its integration in the Microsoft© Visual Studio© projects, along with samples and documentation.

Like all the other frameworks it's asynchronous and event driven. A signal is given when an event takes place. It uses several technologies to transport data, being WebSockets the priority, and then rolling back to other technologies until the client and network has support for that type of transport – this feature is completely transparent – making sure that data is delivered in the best possible time frame. Has capability for connection management: it is possible to know when a client connects to the server, when it disconnects, or for how long it was disconnected, so the clients aren't just sockets connected to a server, it's possible to identify and track their usage. Another great advantage is the abstraction available to the developer from the complexity of iterating through connections to push content to the browser. It is possible to broadcast to groups of clients, depending on their subscription, rather than targeting one by one (which is possible).

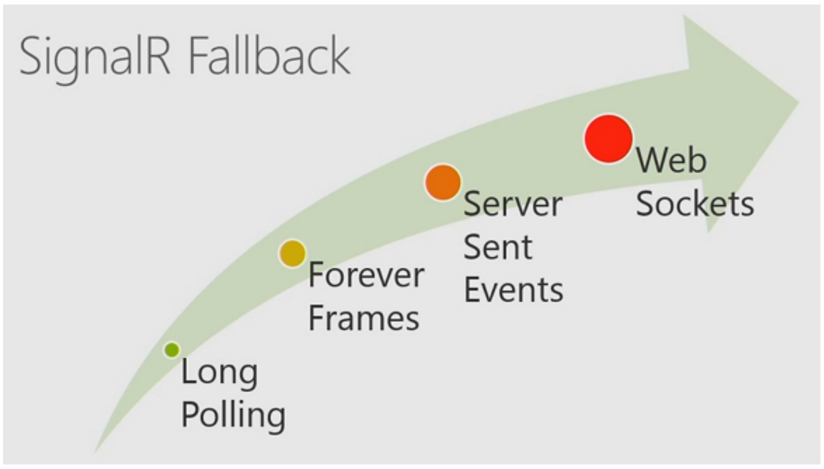


Figure 2.4: SignalR Fallback Technologies[6]

SignalR establishes a persistent connection between the web application and the server, using one of the following technologies:

- **WebSockets** - this is the preferred technology, if both client and server support the technology, it will be used.
- **Server-Sent Events** - if the client or server have no support for WebSockets the framework will try to use Server-Sent Events (if the browser is not Internet Explorer®).
- **Forever Frame** - if the browser is Internet Explorer® and there is no support for WebSockets, this technique will be used to establish a persistent connection.
- **Long Polling** - the last resort is Long Polling, used if the browser is unknown or is from an older version not supported by any of the preferred transport methods.

This transport negotiation is automatic and established at the web application startup, and from the developer point of view the procedure always provides the same result – a persistent connection over HTTP.

It's also possible to use Remote Procedure Call (RPC) via SignalR. it has an API to create this type of connections so the server can, for example, call JavaScript functions in the client application's browser, and the client can also perform function calls on the server.

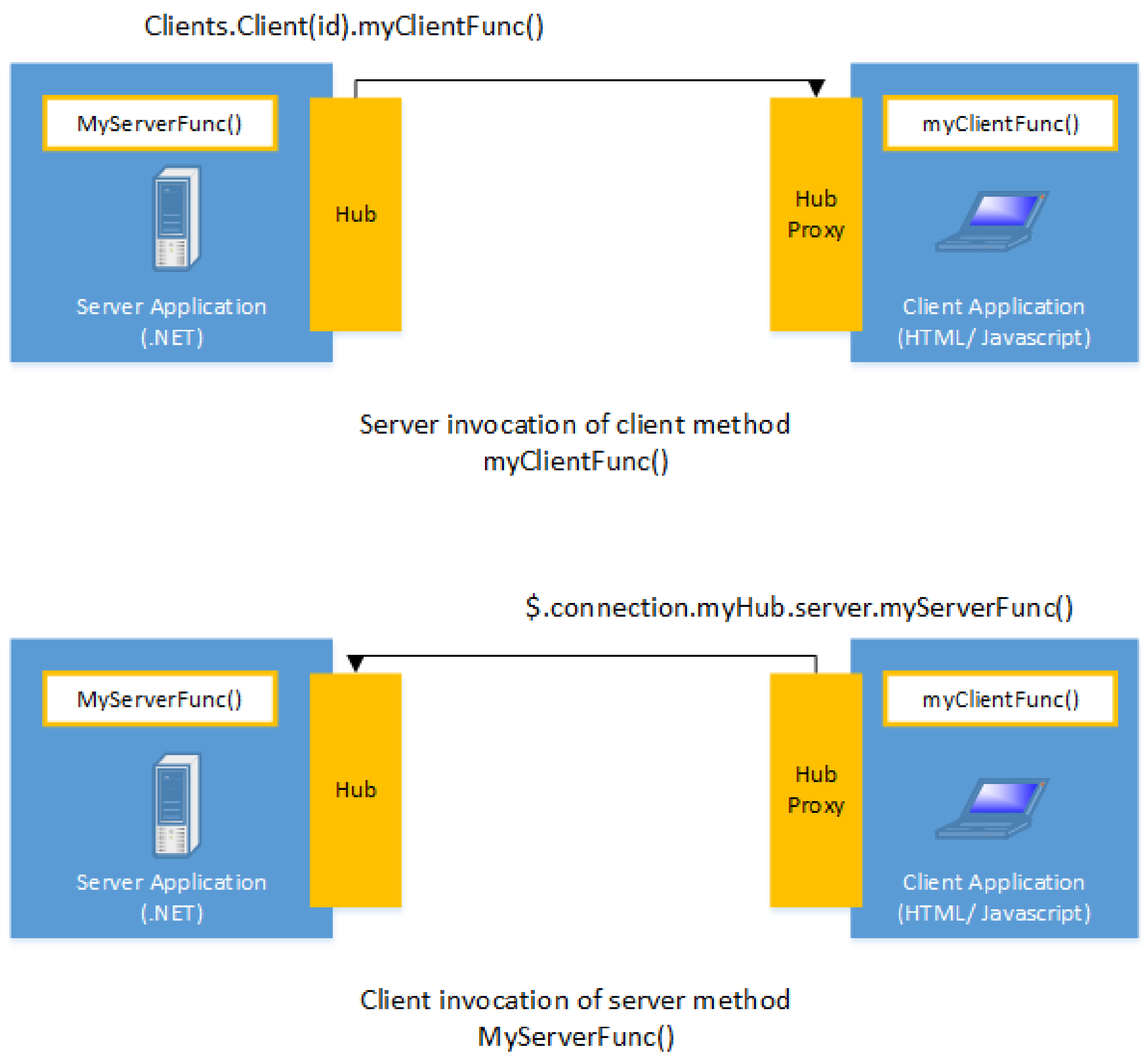


Figure 2.5: SignalR RPC Examples[19]

The server can call the JavaScript function at *C#* code just as it can call a function from an object – the clients are instantiated objects that can be accessed by *id* – as well as the opposite (bottom image) seen when the client application makes a call for a server function. SignalR allows not only RPC but sending objects through the connection as well, and the serialization is handled by the framework - objects are sent as JavaScript Object Notation (JSON). The Figure 2.5 shows an example of RPC from both server and client. "Hub" on the server and "Hub Proxy" on the client, are the names of the framework components used for the communication. They refer to the *Hubs* API of the framework, discussed bellow.

For applications where concurrency is an issue – collaborative tools with multiple users accessing and editing the same data – this framework offers solutions. So if there is a web page or document with many fields that multiple users are editing, is possible to let all users know what fields are being edited, show the changes in real time and in the case of two users editing the same field there is concurrency management.

When the server is in fact a cluster of servers and there is this scenario of a web farm where some clients are connected to one server and another set of clients is connected to other servers, there is the need for communication among them to keep the functionalities running. To solve this, SignalR framework uses a component, named *backplane*, which connects multiple servers and replicates messages so they are redirected to the target clients.

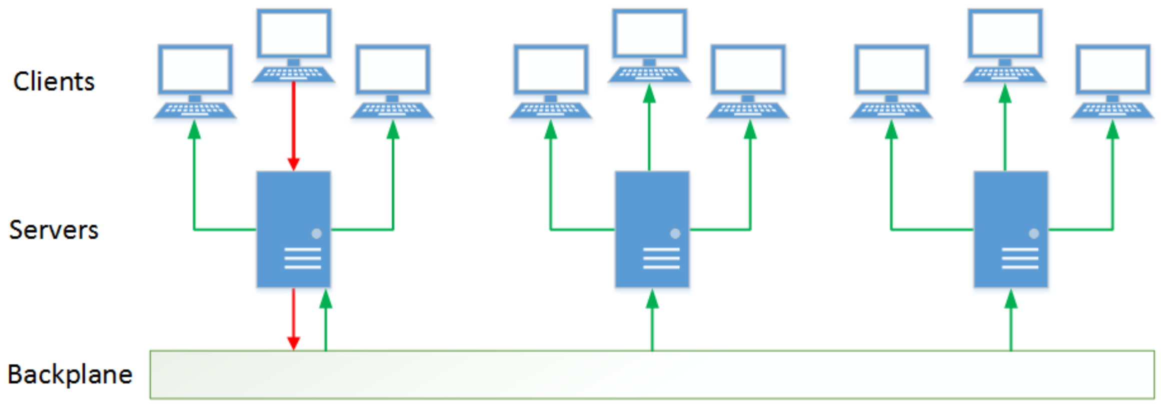


Figure 2.6: SignalR scaleout representation[20]

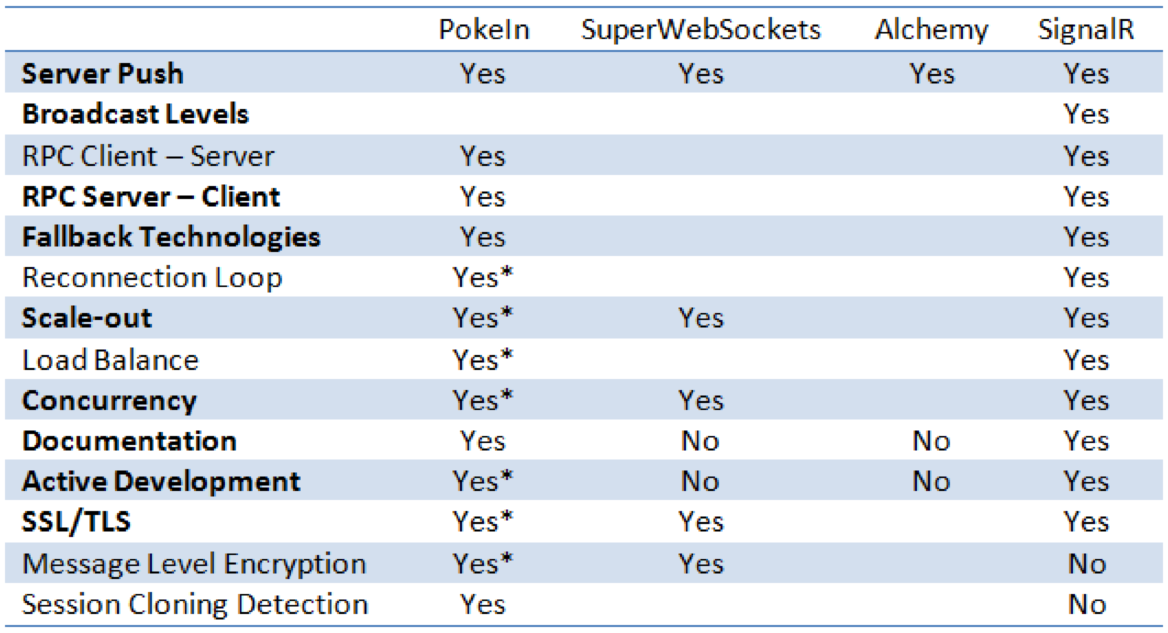
The way these scale out operations are handled is by enabling the backplane and every message generated by a client to broadcast is sent directly to the backplane which replicates the message distributing it to the application servers that will forward it to the target clients. Since every message is sent using a message bus (*IMessageBus*) when the backplane is enabled it will replace the default message bus with the appropriate bus for the backplane – there are three types of backplane and each one has a different message bus implementation. The Figure 2.6 is a graphical example of how the backplane works: the red lines represent the message uploaded by the client to broadcast, and the green lines represent the message sent to all connected clients.

The framework is composed of a two level APIs being the low level the *Connections* API which is a simple API to manage connections, send raw data in both directions, send to groups or individual clients and has tools to know about the client status. Then there is the higher level *Hubs* API which abstracts from *Connections* and give the developer access to all the other features such as RPC.

All these features are very simple to use, there is active support from ASP.NET and the community at GitHub where the source is available, and there is a great focus from the ASP.NET team to always improve on performance.

2.2.5 FRAMEWORK COMPARISON

To put in perspective the features available in each framework this next table will show some of the features that were considered important – being the highlighted ones more important – in the selection of the framework to use.



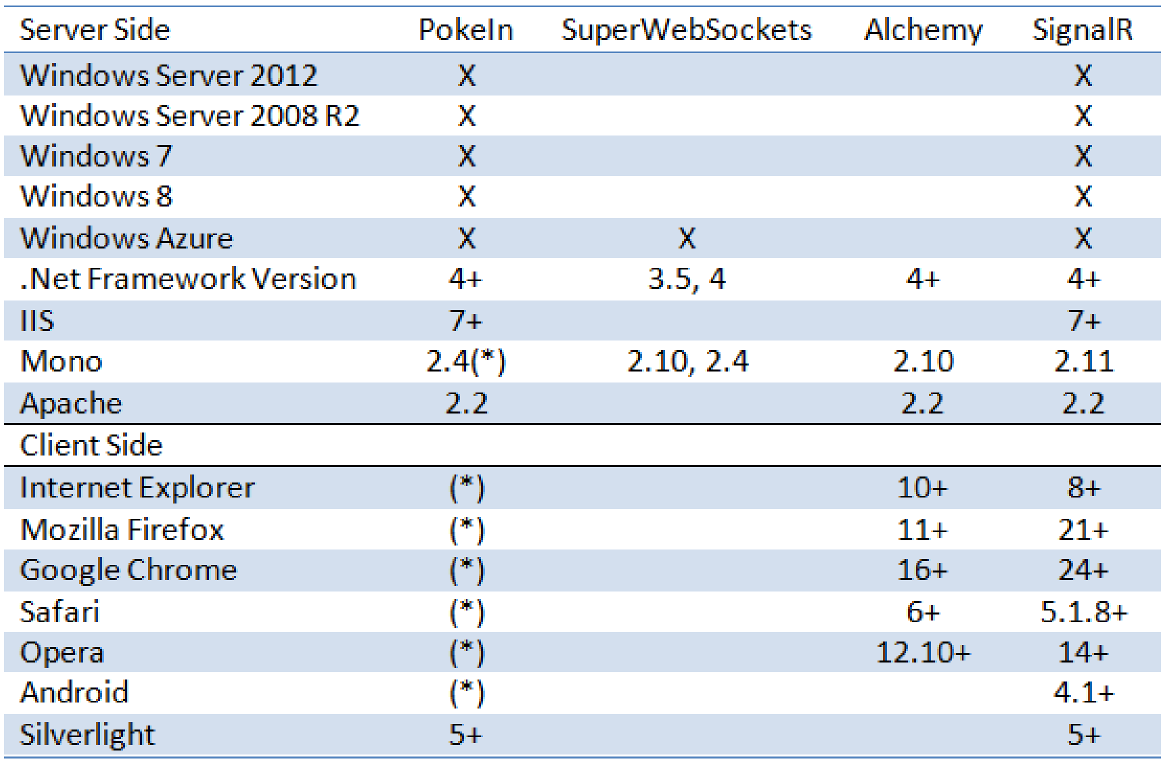
	PokeIn	SuperWebSockets	Alchemy	SignalR
Server Push	Yes	Yes	Yes	Yes
Broadcast Levels				Yes
RPC Client – Server	Yes			Yes
RPC Server – Client	Yes			Yes
Fallback Technologies	Yes			Yes
Reconnection Loop	Yes*			Yes
Scale-out	Yes*	Yes		Yes
Load Balance	Yes*			Yes
Concurrency	Yes*	Yes		Yes
Documentation	Yes	No	No	Yes
Active Development	Yes*	No	No	Yes
SSL/TLS	Yes*	Yes		Yes
Message Level Encryption	Yes*	Yes		No
Session Cloning Detection	Yes			No

Table 2.1: WebSocket frameworks feature comparison

The missing answers in Table 2.1 are due to the lack of information or documentation in the framework official site. Some of the answers were found after small tests (a chat application was built using each framework to determine its usability). The "*" represent features that are available on the paid version of the framework (PokeIn framework only).

Server push is a feature brought by the use of WebSockets (which all of them implement); broadcast levels is the ability to send a message to multiple clients or groups of clients without the iteration through all client connections; reconnection loop represent the implementation of the logic necessary for a client to try and reconnect to the server after the loss of a connection, keeping the context of the lost connection after the reconnection; load balance is used in case of having multiple servers distributing the clients for the available servers; message level encryption allows the messages to be encrypted even if the connection is not.

It's easy to highlight the two frameworks with the most features available at the moment, but in the case of PokeIn some of the features were only available when purchased a license and in the case of a scale-out operation there has to be an intervention of the PokeIn team so it can be accomplished.



Server Side	PokeIn	SuperWebSockets	Alchemy	SignalR
Windows Server 2012	X			X
Windows Server 2008 R2	X			X
Windows 7	X			X
Windows 8	X			X
Windows Azure	X	X		X
.Net Framework Version	4+	3.5, 4	4+	4+
IIS	7+			7+
Mono	2.4(*)	2.10, 2.4	2.10	2.11
Apache	2.2		2.2	2.2
Client Side				
Internet Explorer	(*)		10+	8+
Mozilla Firefox	(*)		11+	21+
Google Chrome	(*)		16+	24+
Safari	(*)		6+	5.1.8+
Opera	(*)		12.10+	14+
Android	(*)			4.1+
Silverlight	5+			5+

Table 2.2: WebSocket frameworks supported platforms

To know the reach of each framework is necessary to know what versions of software they support – which is almost the same for the server side platforms – and how many clients will that represent. Table 2.2 contains the platforms supported by each framework, including browsers that ultimately translate into users. PokeIn supported browsers depend on the subscription level, and to support older versions for each browser will bring a higher price on the subscription (hence the "*").

According to *StatCounter Global Stats*[21] these are the market shares for each browser and versions: Internet Explorer® 8+ (up to Internet Explorer® 11) represented 26.84% of the clients on-line worldwide(from January to December of 2013); Google Chrome 24 (up to version 31) represented 35.78% of browsers used during the same period; Safari 5.1 has a market share of 1.53% and versions 6 and above have 2.43%; Opera 12.1 has a market share of 0.7% and versions 14 and above 0.15%; Mozilla Firefox® has a market share of 18.96% for versions 5 and above not detailing per version status, above the mentioned version. These number show that the supported browsers by the frameworks, represent most of the browsers on-line, and older versions are almost irrelevant.

2.3 BROWSER AND HARDWARE INTEGRATION

For obvious security reasons, browsers have no capabilities to give web applications direct access to *hardware interfaces* in the system they are running. This *sandbox* ensures that if the applications are malicious or if they (both browser and web applications) are compromised, there are limitations to what damage can result from it. This security measure makes it difficult to design in the simplest manner web applications that interact with specific hardware and some solutions where designed to address this issue[22].

JAVA APPLETS

Java applets are one solution to this: by adding a Java applet to the client application it is possible to have it interact with a Java application running on the system, which serves as an interface to the local hardware. When the user enters the client application on the browser, is asked for permission to execute the applet, meaning that the user can give access to its system outside the browser's *sandbox*; this way it's possible for the applet to interact with a previous installed and running Java application that handles the hardware communication.

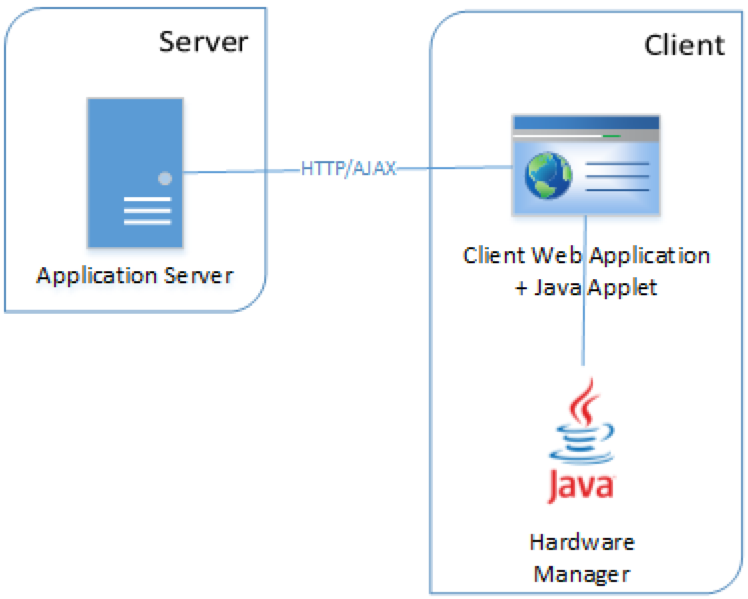


Figure 2.7: Using Java Applets for hardware communication

The figure 2.7 is a simple representation of what the general process and communications would look like. To this general principle described above there are two ways (the most common) of having the applet communicate with the hardware interface Java application. The first would be by using the technology *LiveConnect* which is an API that allows the Java applet to communicate with the Javascript in the web application and with Java applications on the system – through the Java Runtime Environment (JRE).

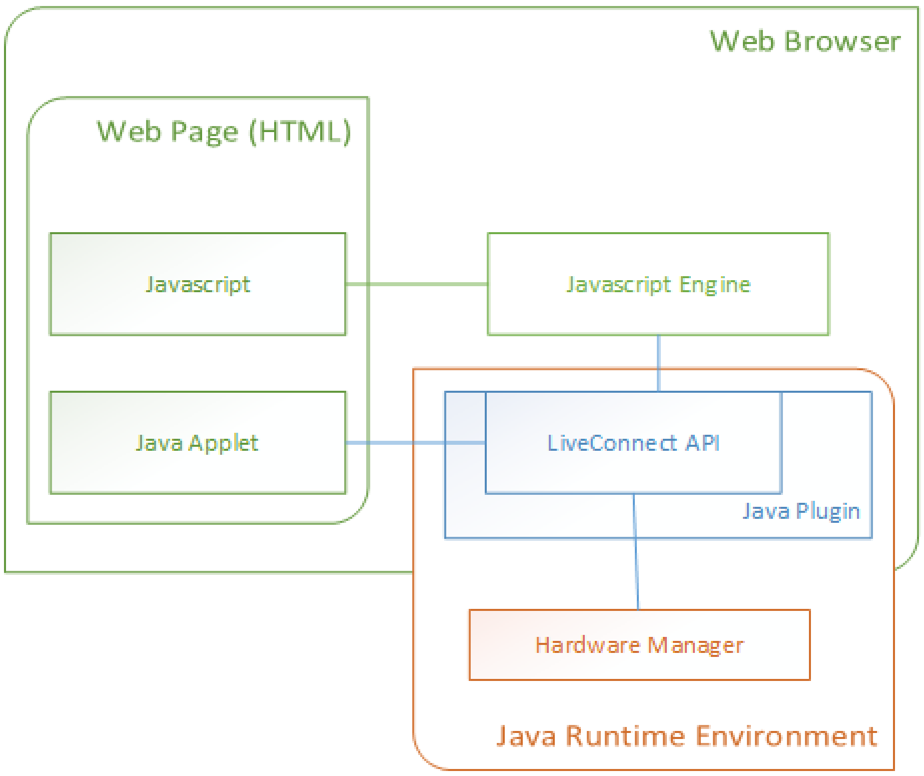


Figure 2.8: Interaction Scheme with LiveConnnect API[23]

As described above the Figure 2.8 shows the overview schematics of the interaction among intervenient parts. This is not an “out-of-the-box” solution even using the *LiveConnect* API. It has some great disadvantages: the fact that there’s a need to add complexity to the client application (by adding the applet); the need to have a Java plugin installed for the respective browser; and that the system user must have the right privileges to run the applet.

The *Hardware Manager* is a local server that handles connections between the Java applet and the hardware device drivers. It serves as an interface with the needed functions for the web application abstracting from the device driver functions.

Instead of using the *LiveConnect* API there is the possibility of using sockets. The Java applet used would be designed to create a socket to a local server – running on the *Hardware Manager* application – and send/receive commands through it. The advantage of doing this is that there is no constraint of using function calls – to which the *LiveConnect* API restricts the applications to – and the socket can be used to whatever ends necessary.

COMMUNICATION USING THE APPLICATION SERVER

Another possibility that uses the same hardware manager of the previous solution is to have it connect directly to the web server (see Figure 2.9), this way the user interacts with client application on the browser and the server will send or receive data directly from the Java application running on the client system.

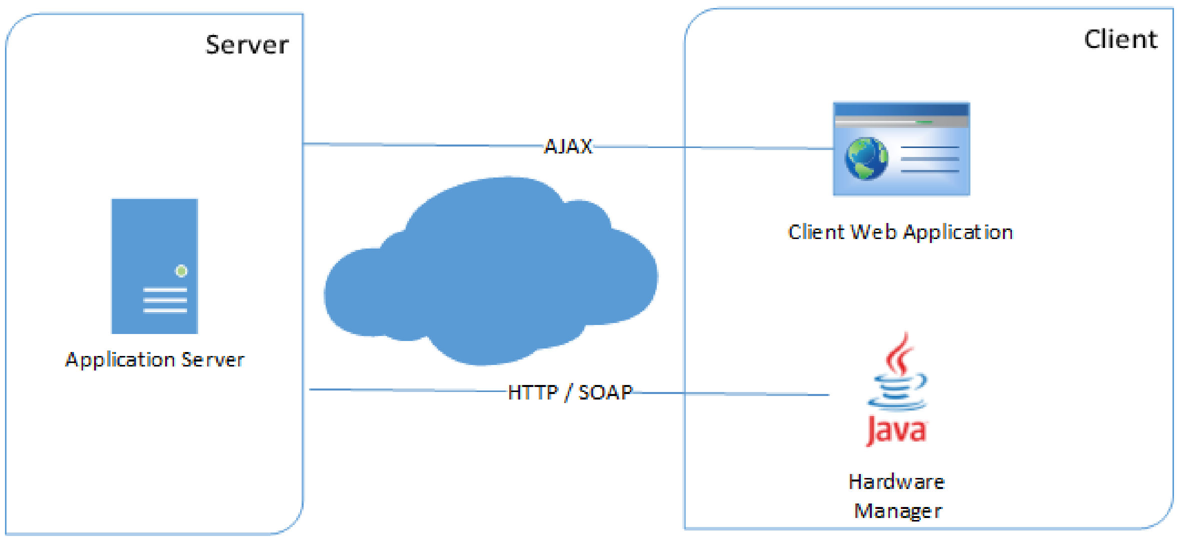


Figure 2.9: Communication using the application server

The *Hardware Manager* application would connect directly to the server using HTTP or Simple Object Access Protocol (SOAP) and the server would then apply the logic that was defined in the Java applet of the previous solution for the hardware. This is a more viable solution seeing that the client application has no knowledge of the hardware, removing the complexity and there is no need for a plugin on the browser. The problem relies on the delay between the time commands on the client application in the browser are made and when the hardware receives that information. Another issue might be the knowledge in the server side to what hardware connection matches the client application, because of the use of Network Address Translation (NAT) this is an issue that too needs addressing and more complexity in the end solution.

2.4 REAL TIME LOGGING

For the standard desktop applications that were distributed and that clients had no contact later with the manufacturer of the software, logging and debugging tools were mainly used in the development or testing processes, but the software was rarely deployed with logger or debugger tools. With the transition from desktop applications to web applications, clients are now in permanent contact with the servers and it is possible for whoever who has access to server applications to consult and keep a log of the interactions the users make on the web applications, if the right tools are implemented in the software.

This has great benefits for the manufacturer in a variety of processes and software development stages (even for other purposes aside from the software engineering perspective):

- the users are now part of the software testing team.
- the tests become real world usages of the software.
- the developer can access directly to information gathered by the loggers.
- statistics can be made about the real world interactions.

- it is possible to assert what are needs that the software doesn't provide yet.
- software can now be redesigned to fit the users interactions patterns with the software.

Software testing is an important aspect of software development, assuring its quality before delivery to the client, and to make sure it is fulfilling its purpose. This is usually a process conducted by a team that is part of the manufacturer team in the software chain of development, or by an outsourced team, or even by specialized software. Any of those responsible for the testing can have multiple types of knowledge about the target software: can be a blind test (when not knowing the purpose and testing its robustness); it can be a test to its capabilities as considering the tasks that it is supposed to perform, etc. The testing processes are logged and then delivered to whom is responsible for revising it and rectify the wrongs (usually the developers). Considering that the software users have their actions logged, it's as if they are part of the testing team (the user is the ultimate tester) and tests are not a scripted task but a real usage of the software.

The information given by the loggers – depending on what is set to be logged – can have multiple purposes outside the software development scope. There can be statistics applied to them in order to transform logged data into knowledge about the system, usage, users" interactions or activities, etc., which can then be used to redesign the product and offer a better service for the client, to create other products based on that statistics – such as report tools – and sell them separately or add to the solution package and upgrade the offer to the client.

Security is a possibility with real time logging, as there are tools to monitor the system's usage, it's possible to know what type of operations are being made and if the users are performing typical operations that the software was designed to perform or are exploiting it beyond their permissions. If there are levels of privileges it is possible to identify possible malfunctions or software bugs if certain users have actions logged that were not possible under normal conditions.

There are several tools/libraries available to make this type of logging (*NLog*[24], and *log4net*[25] are two popular examples for .NET framework applications) but the premise of them all is to save the log messages into files or databases for later access. After the logging files or databases are created and the logging messages are programmed into the software, there are several applications that can be used to access the log files and visualize them in real time – i.e. they are constantly accessing the log file or database searching for new entries.

LogFusion[26] is one of the applications to facilitate the visualization of log files – as described before. It works by monitoring text files, event logs (remote or local), event channels (remote or local) and output debug strings (if the target application writes log to the console). It highlights message rows, has text filtering capabilities and some other tweaks to help find information easier and with less clutter. A similar popular application and freeware as well is Trace Log[27] with the same main features.

There are other applications that have a free subscription but with limited capabilities, either by limited time, of by limited number of log messages per day, or even limited number of hosts under monitoring. Some applications store the information gathered from the log files in the cloud, which is a plus when there are multiple hosts being logged but for the task in hands it's not worth the price and a custom built solution is the way to go.

CHAPTER 3

Background Platform

SIGE© (stands for *Sistema Integrado de Gestão de Escolas*, i.e., integrated management school system) is a system developed by Micro I/O for school management, from the basic day to day operations (such as selling meals, control attendance) to schedule or transport management.

The original system is being developed by Micro I/O since 2001, in order to provide support to school boards for managing schools and to facilitate day to day operations, starting as an application to control students access to school, and then gradually becoming a complex platform with new functionalities being added to fill the needs of the clients (schools) – some features added were client specific, such as "bus pass control" and "bus routes" which were developed per client suggestion but then were offered as part of the main platform to all clients.

As the number of clients grew, and its requests were met, new applications began to take form and by the third version of the software (SIGE© 3) these are its main components:

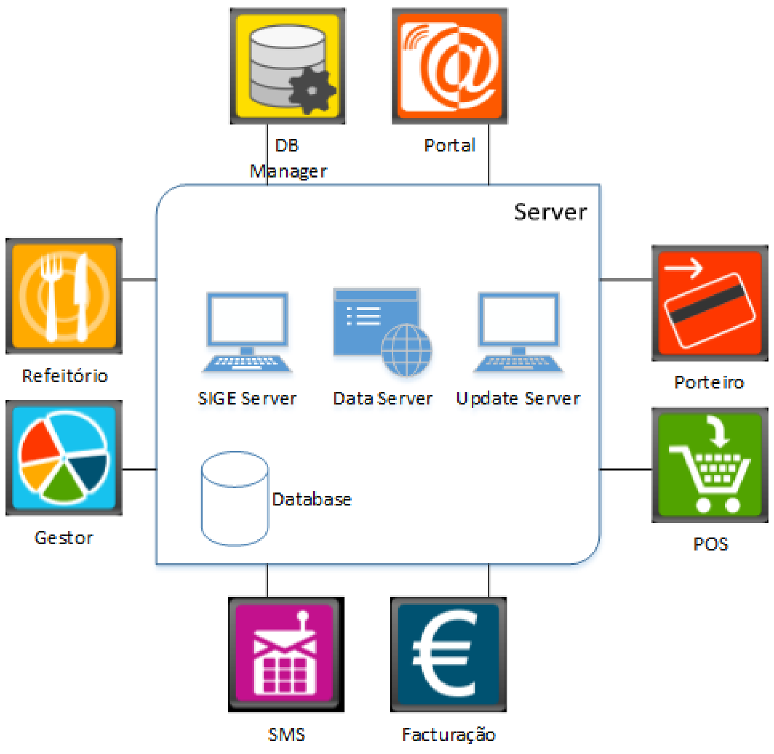


Figure 3.1: Platform main components

The center container in Figure 3.1 represents the server side applications and the surrounding images are the logos of each client application available in the platform.

3.1 ARCHITECTURE

The general architecture can be described as a traditional client-server architecture, operating typically on a private and dedicated network within the school perimeter, but it's not unusual that sometimes it is part of the school network, and more recently in wide areas connecting many schools through the web. All applications and services work exclusively on Microsoft© Windows© machines.

Part of this network is also the hardware created and manufactured by Micro I/O to perform specific tasks – this hardware can be a card reader on a tourniquet with network connection that communicates directly with the server validating a user entrance or exit from school unlocking the tourniquet, or a vending machine that requests the user balance to allow some purchase.

In this platform, server has different meanings, i.e., for different installations and configurations there can be one global server that performs all tasks needed by the system, or it can be several machines each one responsible for a service to provide the required functionalities. In the case of having multiple schools as one client (a school cluster) there is at least one server per school, being that one of them is defined as the synchronization server and has one more task to perform, which is to keep every school database up to date with the others at all times.

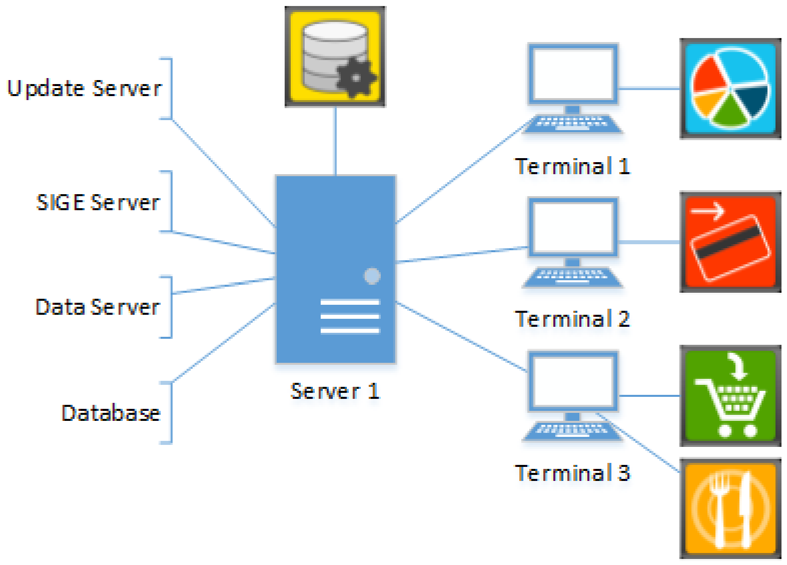


Figure 3.2: Common installation general architecture

The image (Figure 3.2) represents one of the simplest configurations the platform can have: a single machine responsible for all server features. This configuration was most common when schools lacked the resources to have additional hardware responsible for the services, something that as changed in the course of the years and with the advance of technology. The importance of the purchase of new hardware became evident any time the system would go down due to hardware fault in the server side. With more school staff using the software and the software replacing other mechanisms for performing some tasks, the load over the server increased and with it, the importance of having it running by having more services depending on it.

3.1.1 SERVER COMPONENTS

As in any server-client architecture, the server represents the center piece of the architecture, providing the many services needed for the platform to perform the designed functionalities. In this case the services provided, and therefore, corresponding servers are:

- Database Server - responsible for the database storage and Structured Query Language (SQL) server capabilities, in which the application database has one instance dedicated to it.
- *SIGE Server* - the server to which every application on the system points to, and has to have access to, at all times to be running. This service has the location to the database server, and proxies every application to it. When a new installed application runs for the first time, it is asked the Internet Protocol (IP) address for the *SIGE server* machine, and if the connection is lost at any time, the application will not allow for any tasks to be performed.
- *DATA Server* - this service provides an interface to the database to be used by apache. Usually the machine running this service is called the *Web Server*.
- *Update Server* - checks the update directory for a new version of the software, and sends the files to the clients upon request.

- *Hardware Server* - communicates with specific devices/hardware providing information from the database so they can perform their intended tasks. This service is not deployed in every client because it's for specific hardware, that is not used in most systems.

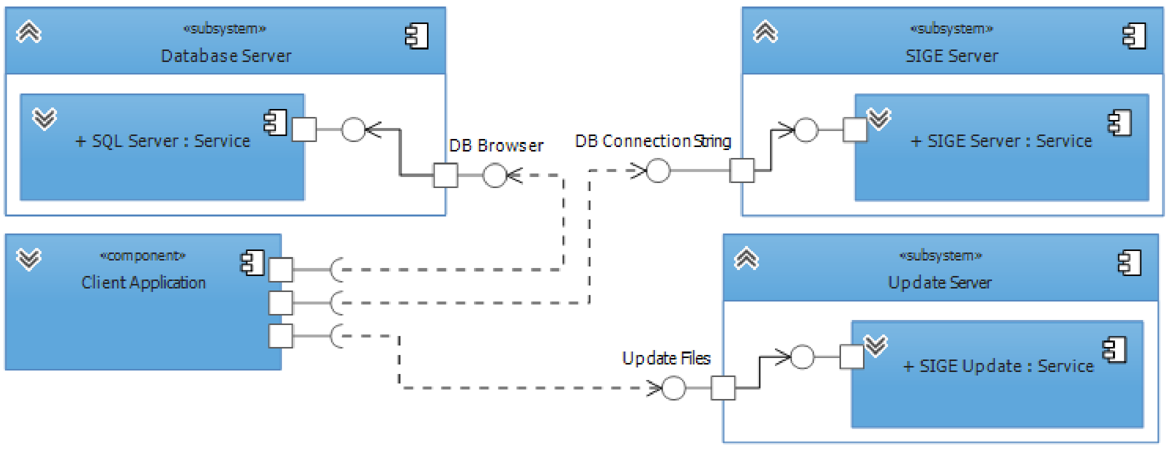


Figure 3.3: Component diagram for components involved in client startup

Whenever a client application initiates it performs two main tasks which are to check if there are updates and ask for the database connection string from the *SIGE Server*. The location of the *Update Server* and *SIGE Server* is configured on the first time an application starts so these tasks can be performed. The *SIGE Server* and *Update Server* don't interact with each other and can be installed in different machines if that is required.

The web server is where the *DATA server* service is running – alongside apache – providing web access to some operations for students or parents such as meal purchase, grades and schedules consult, among others. These operations are made on the web application *Portal*.

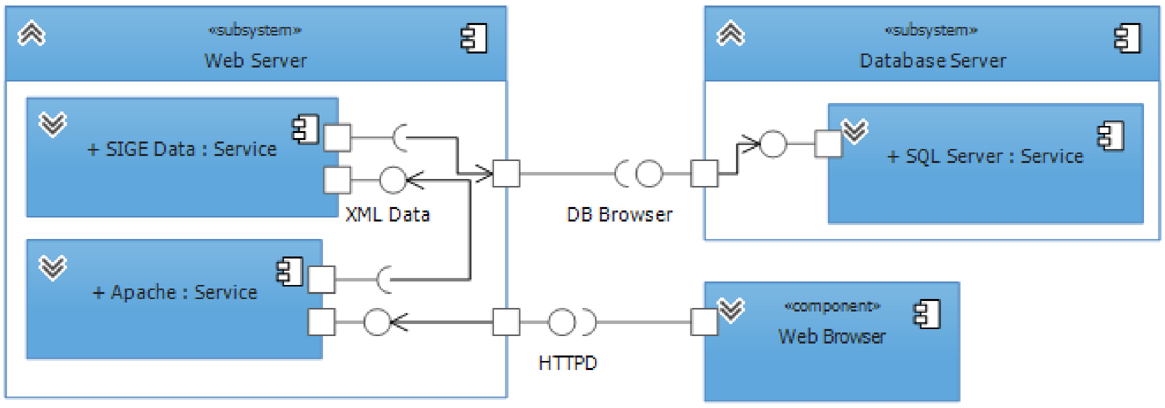


Figure 3.4: Data server component diagram

The *Data Server* connects directly to the database to run the necessary queries and translate the result to Extensible Markup Language (XML) to then be used by *Apache* to display in the web application. There are no other interactions with other services on the platform.

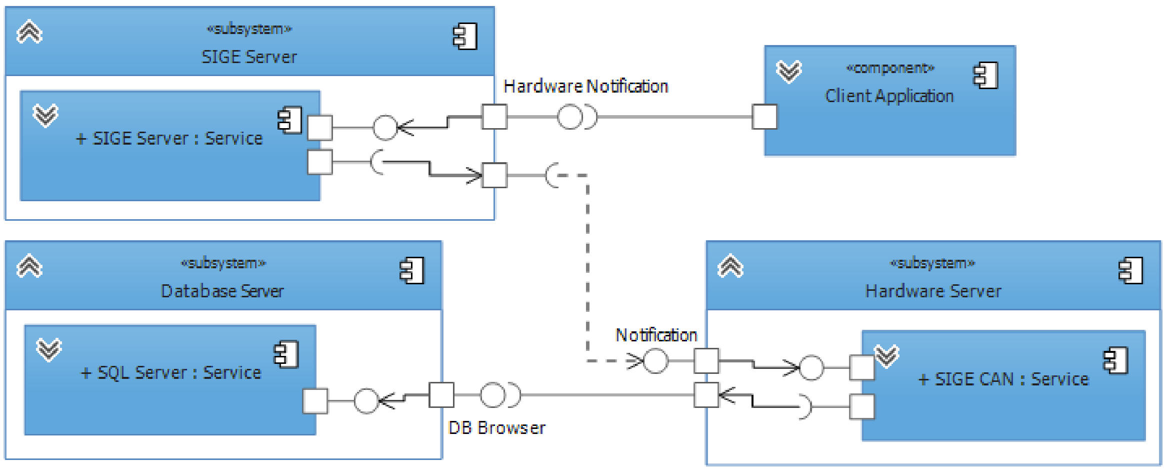


Figure 3.5: Hardware notification component diagram

The platform allows some operations to be handled by hardware without a client application to interact or supervise it. One example is access control using card readers connected to a Controlled Area Network (CAN) network in which there is usually a light or sound alarm to warn about a denied access. When a card is read the *SIGE CAN* service will pick up the information on the CAN network, connect to the database and apply the logic necessary to validate the access and send a signal to the alarm either to sound the alarm or give the green light. This can be accomplished without the client application, which works just as explained, but it can also have the client application to monitor the accesses just has it would with a card reader attached to the terminal. To allow this *visualization* – this is the name of the process in which every client application (*Porteiro*) can be configured to monitor multiple access points in the school perimeter – after the answer about the access validity is known by the *SIGE CAN server*, it's generated a notification sent to the *SIGE Server* which will redirect it to the connected clients on the monitoring list.

3.1.2 CLIENT APPLICATIONS

The set of client applications provided with the platform is composed of four main applications that represent the majority of functionalities of the system, and another four applications which aid/add features and simplify the use of the overall system.

Three of the main applications are easy to identify and recognize its functionalities, they are:

- ***Porteiro*** - (as in doorman) is where the users have first contact with the system. They use it when entering or leaving the school perimeter, and it's where the access control first begins. It's usually composed by a single terminal with many card readers connected, and it allows the supervisor – responsible for monitoring the students' accesses – to consult the users schedule and activities, allow a visitor in, and get alerted when a student tries to exit without permission. The access log is shown in a timeline type interface with basic information regarding the user and its picture[28].
- ***POS*** - with functionalities similar to other Point of Sale (PoS) applications, selling cafeteria products, school books, meals, and any other items needed. It lets the staff member to choose one of the users account (users can have multiple accounts) and purchase items allowed to that

account and user. Since there are many uses for a POS in a school environment – cafeteria, library, charging accounts, etc. – it is possible to have a different set of product pages configured to each terminal, allowing the same application to have multiple purposes. It has also a kiosk mode, in which the users are the students themselves, and there is no staff supervision. It is mostly used in this mode to work as a money accepting terminal where students charge their accounts, and/or schedule meals[28].

- ***Refeitório*** - is the canteen manager application, it has a "gate" where students use their cards to check in and validate the meal, purchased earlier, and can show the information on a screen with a timeline such as *Porteiro* warning staff when there is some unauthorized entrance[28].

Finally the main application of the group and the platform itself is *Gestor* – meaning manager. It is used by the school staff, and the school management to manage parameters, users, products, classes, school activities, schedules etc; generate reports about many subjects such as account movements, entries and exits from school perimeter, grades, products sales, etc. It is a really complex application, and the number of features has never stop growing, both by client request or to respond to a missing feature or new technology now used in the system. Because this application (and platform in general) was not designed to have a plug-in system in which features can be added or removed by client request, the further the development goes, the more complexity it has, as it is adding more and more features to the original design, and so, it takes more resources to provide training to the school staff and resources in customer support with its associated costs, to answer questions about how some operation is made and clear doubts that would not exist if the level of complexity lowered[28].

There are other complementary applications:

- ***SMS*** - which is used to configure and manage all Short Message Service (SMS) related tasks. Feature provided by the system that allows the platform to send informative SMS messages to both students and parents: it can be a message alerting the parent for the fact that the student did not entered the school perimeter in the schedule time; or an informative message to all users about an event taking place in school.

- ***Facturação*** - handles all billing tasks, aiding school staff to deal with billing reports, regarding the usage of the system. It is usually used to print reports and billing information to be checked by accountants later.

- ***Database (DB) Manager*** - used to backup or restore databases, and run update scripts when a new version is available. As anyone would guess this application is used exclusively by the responsible for system maintenance at school or by a Micro I/O agent during a system update. It's generally located in the server containing the database.

- ***Portal*** - to give some more information to the students' parents and allow for small operations to students themselves, *Portal* was created and now students can consult their schedules at home, purchase meals, among other operations. Parents can check the students' grades, classes' attendance, what products are being bought, etc.

3.2 SYSTEM UPDATES

To update the system there are two main steps required, the first is to get the update files and the other to restart the system allowing the applications to update themselves. For this process there is the service *SIGE Update* that is usually located at the server running *SIGE Server* – but it can be changed and the proper configurations have to be made – that is responsible for answering the question *"is there a new version?"* whenever an application starts.

The files needed to update a system are available to customers at the clients' web page, and it can be downloaded if the customer has a valid *update contract*. Then the client has to extract the file in the predefined update folder on the *SIGE Update* server location. For the new files to be delivered to all terminals they have to be restarted (all applications), starting with the *SIGE Server* that will auto update and restart again, and then restarting the client applications one by one so that they ask the *Update Server* if there are new versions and download the files needed[28].

If the update represents a major version, the database may need to be updated itself, and for that there is another task to make: that is to run the application *DB Manager* and run the SQL script given in the extracted update files.

It may not be a complex task, but for the average user – that is the usual user of the system – it can be a tricky task to perform and it is sometimes neglected and some clients will go many versions (in the course of months) without an update.

There are some recurrent issues with this process besides not being done at all, and one of them is when the update files are not downloaded to the terminal where the client application is running, either by network failure or more commonly because of the lack of privileges to write data in the system folder, leading to applications not starting and some basic functions not being able to run on school: one example of this is when a school has a parameter set that only users that logged into the school perimeter using *Porteiro* to be able to use their cards, either to buy food or meals, and the application *Porteiro* fails to start then these users won't have access to the cafeteria. Other issue that is less recurrent but critical is when the person responsible for the update does not update the database or tries to update it using a wrong script version causing multiple malfunctions throughout the services.

3.3 SYSTEM LOGGING AND DEBUGGING

To give better response to customer support and to track some known problems to help identify patterns that lead to a certain error and thus bringing to light new bugs or flaws in design, it was developed and deployed only recently mechanisms to log using the Microsoft© Windows'© event log service. The type of information logged is mostly error messages, and service status updates – logging the start and stop. When there is a support call in progress and the assistant cannot solve the problem or it is identified as a critical problem, the support call goes to the next line of help and is made using remote access to the school server (or the terminal where the problem resides) and the first action to perform is to try to identify known error codes in the system log under the application tag.

This process of remote access and problem solving is heavy time consuming and sometimes the system as to be stopped or the support has to be made when there are no users interacting with the system. When the error does not clarify its origin, the process leading to it has to be repeated in order to understand its origin, and in that case a more thorough debugging would be of great impact. There are cases of bugs that were only found after the client's database was cloned by the assistant to be provided to the developers so that they could try to recreate the process and evaluate where the bug occurs.

The remote access is the only way the assistant can understand the user's interactions with the application that lead to the error, or if it isn't an error at all, and because sometimes connections are so slow a member of the support team has to visit to the school in order to perform the required assistance.

CHAPTER 4

Solution and Case Study

To move forward into the design and development of a possible solution, it was necessary to establish solid goals for the platform and determine the pre-requisites needed for each one of them to work. And so, the main goal asked was to provide the clients of the current system an alternative to the desktop applications, with the same features, that can be used by the same users intuitively, and with as little effort as possible – hardware specifications and installations procedures – to both Micro I/O and the clients. This was proposed to allow a smooth transition for the clients into the next version of the software so that it would be welcome instead of being dropped and clients opt out of adopting it. For the main goal to be reached the application must work in real time, in interaction with existing hardware, support to the same operating systems they are already using, and keeping network configurations as they exist, maintaining this way the same characteristics of the current version.

To improve on the current version other requisites were added so that the technological advance would justify the change: the client applications should run independently of the operating system; mobile devices are to be taken into account in this next version, with methods of accessing the applications; it should be possible for the system to auto update; the system must be able to track user interactions in order to boost client support service and bug report/solution; and it was proposed to solve an existing problem with the current system which is concurrency – it is possible that two users try to edit the same record at the same time and the solution has been that the last to save overrides the information.

Given the facts the best way to go would be to implement a web application with client to server and client to client real time connections and capabilities of communicating with local hardware. This should be built using the SignalR framework for the real time communications, given all its fail safe mechanisms and the fact that it has ASP.NET support assuring continuity in its development. For the interface (SPA) it will be used the Kendo User Interface (UI)[29] from Telerik – this choice was made by Micro I/O developers and so it was purchased a license for the software. This SPA framework allows developers to build responsive interfaces[30] using HTML 5 and JavaScript that adapts to the browser in use, screen size and more without much intervention from the developers giving the web application the necessary tools to handle all types of devices and especially mobile devices.

4.1 CASE STUDY

Because of the large amount of features and applications in the current platform, there was a need to choose one of the main applications and implement the technologies discussed to it as a case study. The application chosen was *Porteiro* because it has the elements needed to prove the concept of this thesis: it has the least amount of features from the main applications; it has hardware integration; and the main focus for its implementation is the real time aspect, making it a solid proof of concept.

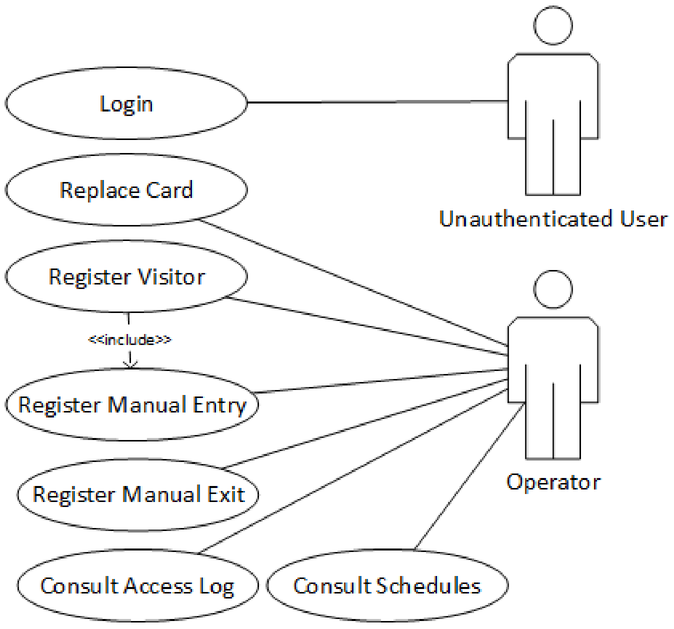


Figure 4.1: Use case diagram for *Porteiro*

As expected the number of use cases for this application is very low, and very simple to understand, as shown in Figure 4.1. If the user is not logged in there are no operations to do other than log into the system. Each operation presented depends on the operator privileges so it is possible that an authenticated operator may not be able to perform some of the use cases identified. These use cases derive from the features presented in the SIGE© 3 product presentation[31]:

- *Login*: the application can only be used by an operator with login credentials, and specific permissions to login at that terminal.
- *Replace Card*: if any user needs card replacement, there are usually several replacement cards to this purpose, so the user can still use other applications, such as *POS*, even if it lost its card.
- *Register Manual Entry*: if the user doesn't have the card to access the school and there are no replacement cards, the access can be registered manually.
- *Register Visitor*: process of registering visitors information and automatic register an entry access to the school.
- *Register Manual Exit*: same process for the manual entry but the access direction is the opposite. To register a visitor exit access, a list of visitors inside school perimeter is displayed during this operation.
- *Consult Access Log*: the operator can consult the access log for a certain user, class, or specific date.

- *Consult Schedule*: the operator can consult the schedule for a certain user, class or classroom.

4.1.1 HARDWARE INTEGRATION

After the study of the best approaches on how to deal with this issue, it was decided that the integration would be made using the real time tools that were going to be used on the web application. It would be created a service to run on the client side – on the clients that had hardware connected named *Hardware Interface* – to interact directly with the server letting it decide what to do with the information and when to deliver it to the client.

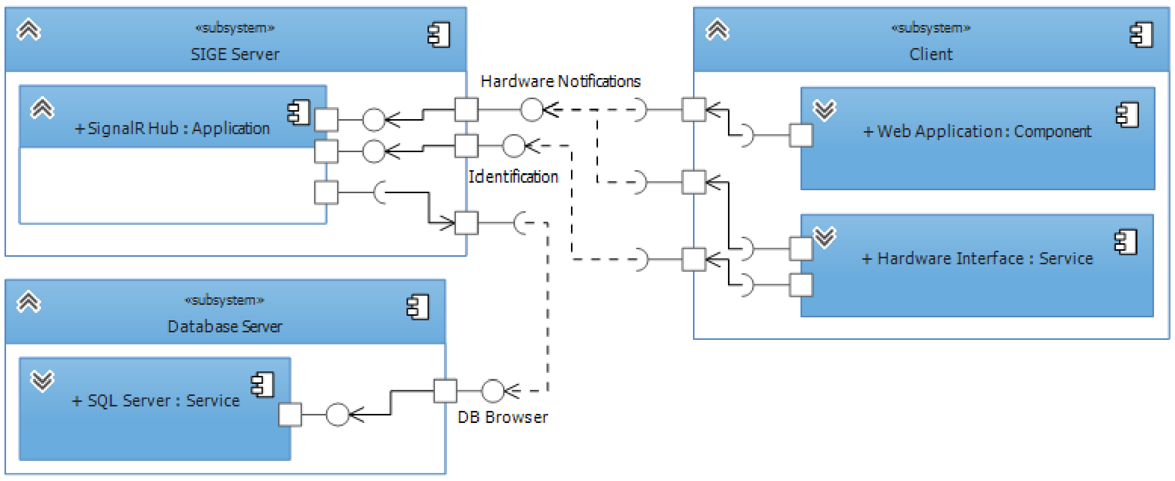


Figure 4.2: Hardware integration component

The application server provides an interface for the client's web application (in this case *Porteiro*) and *Hardware Interface*, for receiving and sending notifications about hardware interactions (see Figure 4.3). The *Hardware Interface* also needs to be identified so the server can know to which client it belongs (terminal identification) and what hardware is connected to it. The server application will query the database for the information regarding what hardware is configured in the terminal to which the *Hardware Interface* belongs to.

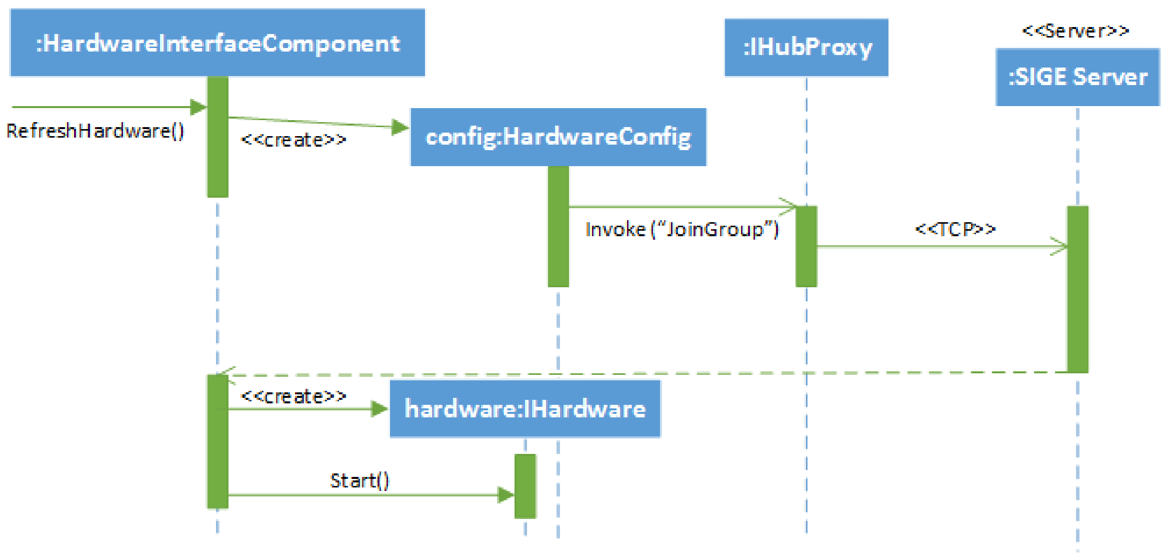


Figure 4.3: Hardware service identification sequence

The *HardwareInterfaceComponent* (Figure 4.3) represents the *Hardware Interface* in the client service. When the *Hardware Interface* service starts it will run the method *RefreshHardware* that is responsible for the identification of the terminal where the service is running, and creating the necessary configurations for the hardware interface to start. It will create a hardware configuration (*HardwareConfig*) that will identify the host terminal on the first attempt to connect to the server that will join the service connection in the same group as the web client of the identified host – the client connections in the server are grouped so that when there is new information, the server will broadcast it to a group of clients. After all configurations are made the *HardwareInterfaceComponent* will create the *Hardware Interface* that will interact with the hardware (one instance per configuration).

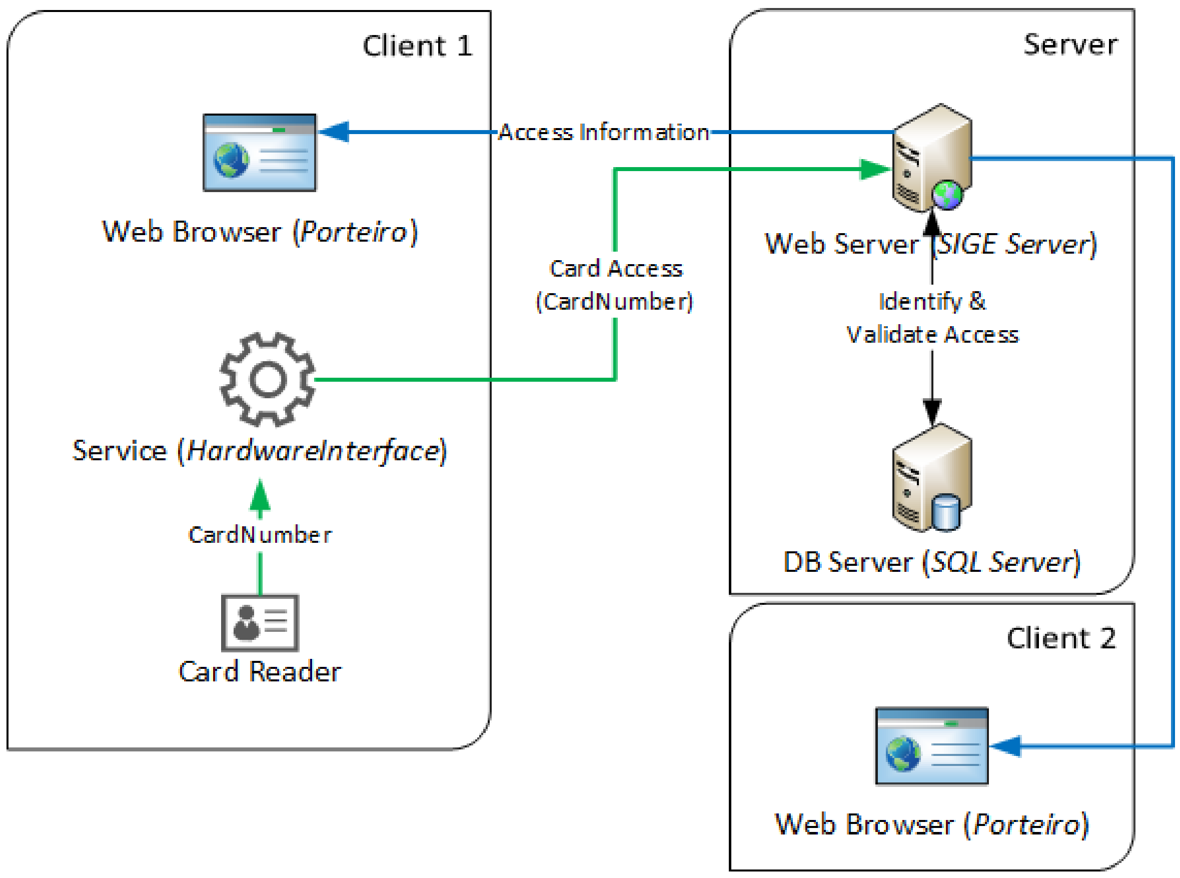


Figure 4.4: Hardware notification example

As an example, in the scenario represented on Figure 4.4, the *HardwareInterface* service is connected to the hardware via serial (either USB or RS232) and when there is a read on a card reader the service will send the tag (*CardNumber*) that was read to the server using the SignalR connection established earlier. It is the server responsibility to interpret the tag and identify the card and the user it belongs to, as well as all operations needed to validate the access by the user – consult access points the user has permissions to use; consult its schedule and other rules. Then the server sends the information about the validity of the access to all clients that have a visualization of that access point: in this example there are two clients with the application *Porteiro* running and if both have configured to have a visualization of that access point, they will receive the same message.

Most of the implementation has the server resolving this type of logic and the clients act as a visualization of the result. It was built as such so that the client side has less complexity and could run smoothly, the alternative would have the clients make many connections to the database through the *SIGE Server* thus increasing the number of connections drastically, and the implementation of such logic in JavaScript was in many cases unfeasable. With this, the server will have more tasks "in hands" to resolve, but having the database (in most cases) running in the same machine speeding the operations by a large margin.

4.1.2 LOG VIEWER

After the experience developing the current version of the software, the importance of logging was well recognized in the Micro I/O developers' team. So this time building a new version was required that it had logging from the beginning. It was asked that all operations in the system should be logged, and it was asked to search for the best tool or library to deal with the issue. Two of the available tools found – and mentioned in the chapter *State Of The Art* – are NLog and Log4net, being that the choice in the end was NLog because of a simple feature it had over Log4net. Both frameworks presented the same basic concepts and they were easy to work with but NLog has a feature that allows the user to choose targets other than the typical database or file[32]. This difference allowed for the experience of having it target a function from SignalR Hub on the project and the result is that now any log message can be sent to a connected client allowing him to receive log messages in real time.

With this possibility the logging tool had now a different weight in the development process and on the future of the web application in general. Combined with the fact that it is possible to change the level of debugging and how deep it can be made in runtime it gave it a new meaning – by having multiple logging levels it is possible to have log messages from every line of code to "high level" messages such as information about when a *Porteiro* supervisor logs into its working station. It was defined that there were four main types of log levels:

- **Debug** - "low level" messages to help developers in the development stage and in the future to find possible malfunctions within the software, they are mostly messages reporting when a routine has started or ended, what function is running, exception errors, etc.
- **Info** - these are "high level" messages to be viewed by a system supervisor or even school staff. It gives information about users in the system, operators and what their interactions an example would be "Alice charged her account with 1 euro" or "Bob consulted Alice's schedule".
- **Warning** - when a user tries to perform an action that is not allowed by the system it is logged as a *warning*. Like in the *info* level these messages can be viewed by all operators with the right permissions.
- **Error** - as it seems, it is used to log errors, possible errors that might occur and the system can handle are logged under this tag.

The *Log Viewer* – name of the tool to implement – is simply a grid showing log messages with filters for date, name of the application that generated it, log level, or even strings in the message, so it's possible to look for a specific username and find the logs associated with it.

Because of the amount of messages generated at any time, and to help the user to consult the log, the messages are not sent when they are generated – as it was firstly designed – because the grid would grow and it was not possible to read the messages. The implementation was changed so when there is a new message to show, the SignalR Hub will send not the message but a simple notification alerting the *Log Viewer* that a new message is available, giving the user the possibility of refreshing the grid whenever he wants.

4.1.3 USER INTERFACE

MVC was the architecture of choice for the user interface development. The web application development is divided into three main components that give a more intuitive way of creating the web application and its interface:

- **Model -** implements the logic of the application, has means of accessing and managing data and responds to commands sent from the user (through the *Controller*).
- **View -** it's the representation of the interface and the output to the user's interactions with the application. The updates to the *View* are sent from the *Model*.
- **Controller -** responsible for the interaction between user and *Model* which turns it into the means for the user to interact with the application. The input is manipulated in this component so it can be converted into commands that the *Model* can process.

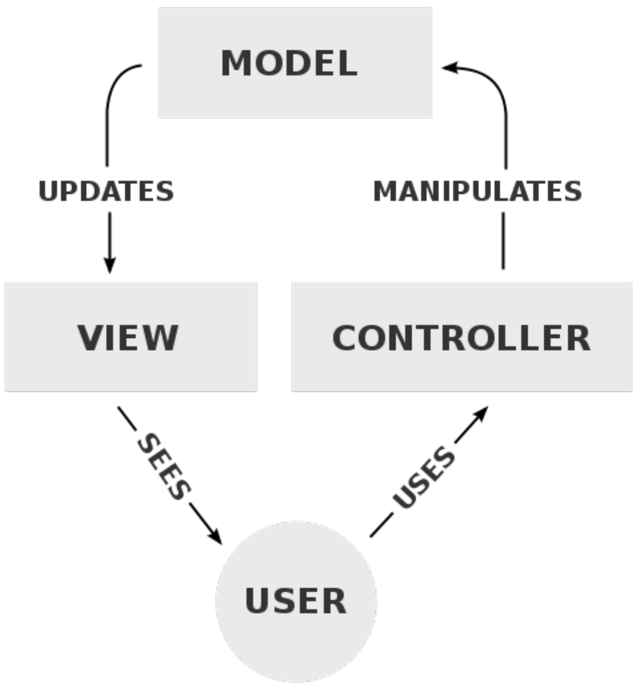


Figure 4.5: MVC components[33]

The Figure 4.5 represents how the different components – and user – interact to provide the end result. It's a simple cycle, in which the user is provided with the interface formed by the views components and the controllers for each view to interact with the application[34]. Because the interactions with the interface do not correspond to the exact commands used in the business layer of the application, the controllers transform those interactions into commands for the models. The Kendo UI framework provides tools[35] to help developing the views and controllers for the interface facilitating the implementation of the MVC architecture.

CHAPTER 5

Implementation

Because every installation is a different scenario, and for each client there's a different set of configurations, an hypothetical scenario was thought to show how an implementation would be. It is only considered the components needed to the case study.

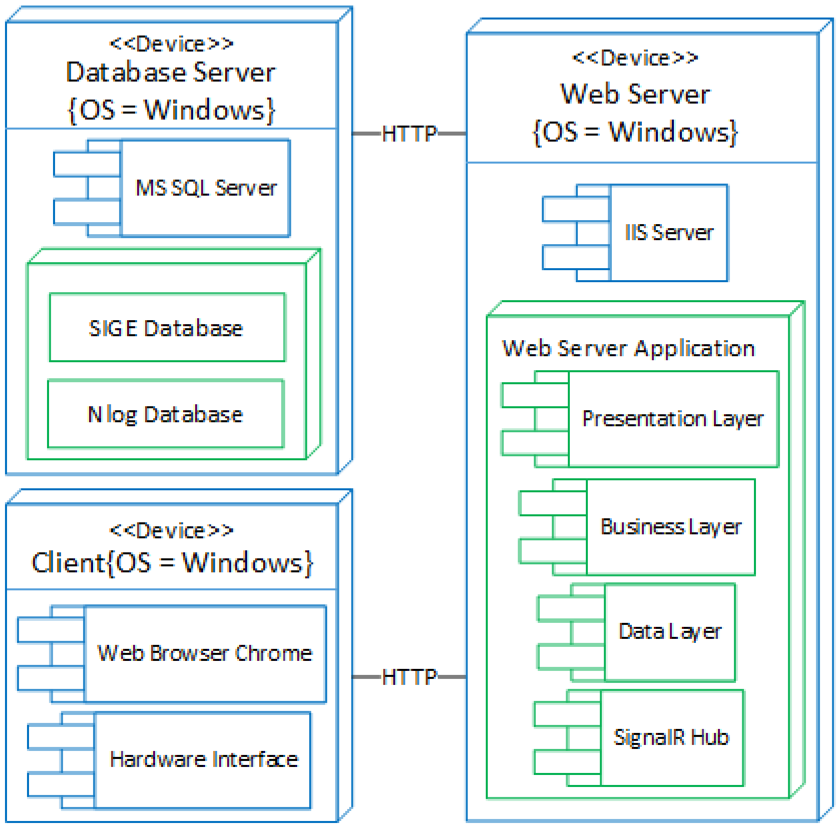


Figure 5.1: Deployment diagram (hypothetical scenario)

The diagram - Figure 5.1 - exposes the main components that for the solution architecture. Of course that even for our case study the database server and web server could share the same machine but this is a more realistic approach – based on the trend on recent installations – and the client could run on a Linux operating system and therefore the component for *Mono* would have to be shown too. In a real case deployment, more devices are added with operating systems such as Mac OS X, and Linux based systems, as well as the most used browsers supported by the SignalR framework – displayed previously.

To better understand the work behind some of the key components, it will be shown some code examples of how it was developed - it will only be shown the relevant code to the study. The Data Layer and Business Layer will not be shown since they don't represent the goals of this dissertation.

5.1 SIGNALR COMMUNICATIONS

```
public class HardwareHub : Hub
{
    public void NewCard(long hardwareID, string card, bool serial)
    {
        cartao = serial ? DOCartao.GetBySerial(card) : DOCartao.
            GetByCardNumber(long.Parse(card));
        DOPostoHardware ph = App.DB.FindByKey<DOPostoHardware>(hardwareID);
        AccessResult resultAcesso = new AccessResult();
        resultAcesso = GetAcessoResultByPontoAcesso(ph.AcessoPonto.
            AcessoPontoID, card,  cartao, (AcessoTipos)ph.InOut.Value);
        Clients.OthersInGroup(StartGroupNamePontoAcesso + ph.AcessoPonto.
            AcessoPontoID).NewAcess(resultAcesso);
    }
    public Task JoinGroupHardwareService(string hardwareIdOrComputerName,
        NotificacaoCartaoTipos notificacaoCartaoTipo)
    {
        var groupName = StartGroupHardwareHub + hardwareIdOrComputerName;
        return Groups.Add(Context.ConnectionId, groupName);
    }
}
```

The code above represents one method (*JoinGroupHardwareService*) used by the clients' *HardwareInterface* services to join a client group – thus allowing the server to send notifications to groups – belonging to a *Porteiro* visualization. And in case of a new card read the *HardwareInterface* will invoke the method *NewCard* on the server which validates the access and returns the result (*AccessResult*) to the clients on the broadcast group. These methods are part of the *HardwareHub* (SignalR Hub) responsible for connections between hardware and web application.

```
$(document).ready(function ()
{
    var microioHubProxy = $.connection.hardwareHub;
    microioHubProxy.client.NewAcess = function (cardInfo)
    {
        if (options.newTimelineAccessCallback)
            options.newTimelineAccessCallback(cardInfo);
        else
            microio.timeline.newTimelinePass(cardInfo);
    };
});
```

This is the JavaScript function used to connect to the *HardwareHub* on the client side and handle the the *NewAccess* function calls from the server.

5.2 HARDWARE SERVICE

```
public static class SignalRClient
{
    static HubConnection hub;
    public static IHubProxy hardwareHub;
    public static Action<List<HardwareConfig>,IHubProxy> RefreshHardware;

    public static void Start(string url)
    {
        var querystringData = new Dictionary<string, string>();
        querystringData.Add("machineName", Environment.MachineName);
        hub = new HubConnection(url, querystringData);
        hardwareHub = hub.CreateHubProxy("HardwareHub");
        hub.StateChanged += hub_StateChanged;
        hub.Start().Wait(1000);
    }
    static void hub_StateChanged(Microsoft.AspNet.SignalR.Client.
        StateChange obj)
    {
        if (!HardwareLoaded) RequestHardwareList();
    }
    public static void RequestHardwareList()
    {
        hardwareHub.Invoke<List<HardwareConfig>>("RequestHardwareList",
            Environment.MachineName);
        RefreshHardware(j.Result, hardwareHub);
    }
}
```

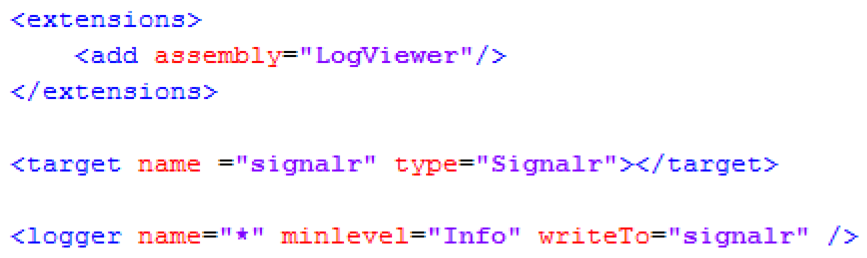
This is an extract of the code from *SignalRClient* implemented on the *HardwareInterface* service – it is not complete and only shows key parts of the service. When it's started, the client will connect to the server using a SignalR Hub and use that connection to create a proxy on the local machine (*HardwareHub*). That proxy will be used to communicate with the server to request the list of hardware

(*RequestHardwareList*) and for each configuration on the list it will use the function *RefreshHardware* that will be shown next.

```
public class HardwareInterfaceComponent : IServiceComponent
{
    public void Start()
    {
        SignalRClient.RefreshHardware = SignalRClient_RefreshHardware;
        SignalRClient.Start(ConfigurationManager.AppSettings["Server"]);
    }
    private void SignalRClient_RefreshHardware(List<HardwareConfig> val,
        IHubProxy hub)
    {
        val.ForEach( config => {
            IHardware hardware = AvailableHardware.FirstOrDefault(p => p.
                Tipo == config.Tipo);
            hub.Invoke( "JoinGroupHardwareService", config.ID.ToString(),
                NotificacaoCartaoTipos.Acessos);
            hardware.Start();
        });
    }
}
```

When the service starts it will invoke the *Start* function presented before from the *SignalRClient* and set the function *RefreshHardware* – used on the code mentioned before – to a local function (*SignalRClient_RefreshHardware*). The client will use the *RefreshHardware* as soon as it receives the hardware list to configure it and start it. So for each hardware configuration a new interface will be created and the SignalR Hub proxy (created on the *SignalRClient*) will be used to connect it to the server. When all is finished the hardware interface service will start communications with the server.

5.3 LOG VIEWER



```
<extensions>
    <add assembly="LogViewer"/>
</extensions>

<target name ="signalr" type="Signalr"></target>

<logger name="*" minlevel="Info" writeTo="signalr" />
```

Figure 5.2: NLog configuration for SignalR

The Figure 5.2 shows the NLog feature that allowed for the use of SignalR in the logger: by adding an extension to the project *LogViewer* it's possible to set the target *Signalr* which is implemented in

LogViewer. The minimum level of log messages defined is *Info* meaning that debug messages will not be sent to this target, but are stored in the database – in the database target the minimum level is *Debug*.

```
namespace LogViewer
{
    [Target("Signalr")]
    public class SignalrTarget: TargetWithLayout
    {
        protected override void Write(NLog.LogEventInfo logEvent)
        {
            IHubContext context = GetClients();
            context.Clients.All.newEntry();
        }
        private dynamic GetClients()
        {
            return GlobalHost.ConnectionManager.GetHubContext<LogViewerHub
                >();
        }
    }
}
```

This is the implementation of the target *Signalr* configured previously. NLogs' targets will use the *Write* method to create new log messages so the method is overridden to send a notification to all clients connected to the *LogViewerHub*.

```
[HubName("loggerHub")]
public class LogViewerHub : Hub
{
    public void LogEntry()
    {
        Clients.All.newEntry();
    }
}
```

The *LogViewerHub* is a simple SignalR hub with nothing to add, just allow clients to connect to it and implement all other hub characteristics. It has the method *LogEntry* to allow classes other than NLog to send a notification to the clients - NLog will send it through the target shown above.

```
<script>
    $(function ()
    {
        var hub = $.connection.loggerHub,
        alerts = 0;
        $(".top-menu-bar-alert-container").on("click", function () {
            $("#Grid").data("kendoGrid").dataSource.page(1);
        });

        hub.client.newEntry = function ()
        {
            var $container = $(".top-menu-bar-alert-container"),
                $notif = $container.find("span.alert-counter");
            if (!$notif.length)
            {
                $container.find('ul > li > span.k-link').append( '<span class
                    ="alert-counter badge"> </span>');
                alerts = 0;
                $notif = $container.find("span.alert-counter");
            }
            $container.find("span.alert-counter").html(++alerts);
            $notif.clearQueue().animate({ fontSize: '20px', height: '26px'
                }, 300).animate({ fontSize: '10px', height: '13px' }, 300);
        };
        $.connection.hub.start();
    });
</script>
```

In the view that creates the *LogViewer* interface, this is the script that connects to the *loggerHub* (name of the *LogViewerHub*), resets the notification counter (*alerts*) and creates the logic for the *newEntry* function invoked by the server. It simply increments the counter at every new log message and shows a small animation to alert the user. At last the connection to the hub is started so communications can begin.

```
using NLog;

public static class ExampleClass
{
    static Logger Log = LogManager.GetCurrentClassLogger()

    public static void doSomething()
    {
        Log.Debug("Doing something now!");
        Log.Info("Sample Info message.");
        Log.Warn("Sample Warning message.");
        Log.Error("Sample Error message!");
        Log.Fatal("Fatal Error message!!!");
    }
}
```

To use the logger it's as simple as adding a reference to the NLog project and use its *LogManager* to get the logger in the context that the class belongs to. The logging of messages is really intuitive as shown in the function *doSomething*.

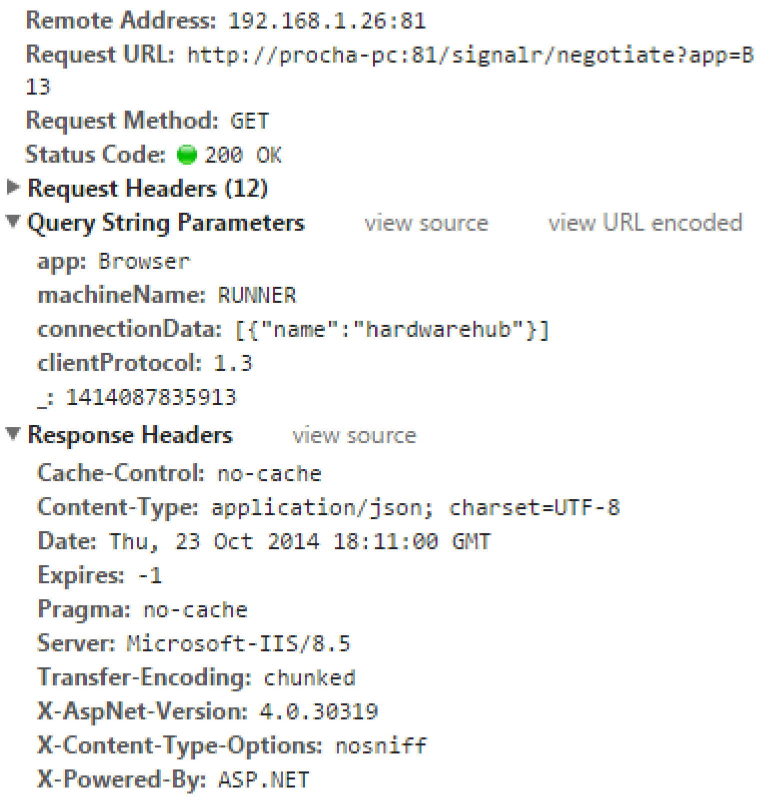
CHAPTER 6

Tests

To prove the concept presented and the implementation architecture, a few tests were made to show how some of the key components work and to stress the communication between server and client using hardware even when the communication is not using WebSockets – considered the worst case scenario.

6.1 Porteiro start with WebSockets support

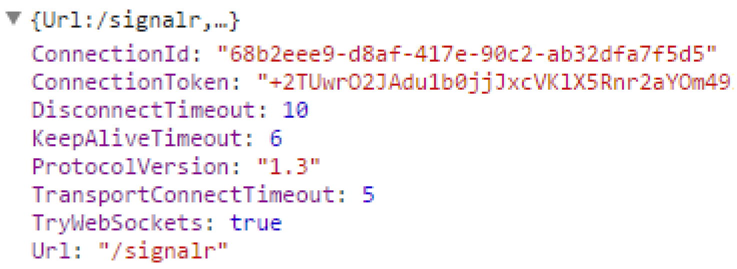
When the web application starts it has to negotiate the communication parameters with the server. In the first case the server supports WebSockets – Microsoft© Windows© 8 with Internet Information Services (IIS) 8.5, .NET Framework 4.5 and Microsoft© SQL Server© 2012.



```
Remote Address: 192.168.1.26:81
Request URL: http://procha-pc:81/signalr/negotiate?app=B
13
Request Method: GET
Status Code: ● 200 OK
▶ Request Headers (12)
▼ Query String Parameters      view source      view URL encoded
   app: Browser
   machineName: RUNNER
   connectionData: [{"name":"hardwarehub"}]
   clientProtocol: 1.3
   _: 1414087835913
▼ Response Headers      view source
   Cache-Control: no-cache
   Content-Type: application/json; charset=UTF-8
   Date: Thu, 23 Oct 2014 18:11:00 GMT
   Expires: -1
   Pragma: no-cache
   Server: Microsoft-IIS/8.5
   Transfer-Encoding: chunked
   X-AspNet-Version: 4.0.30319
   X-Content-Type-Options: nosniff
   X-Powered-By: ASP.NET
```

Figure 6.1: Negotiation request

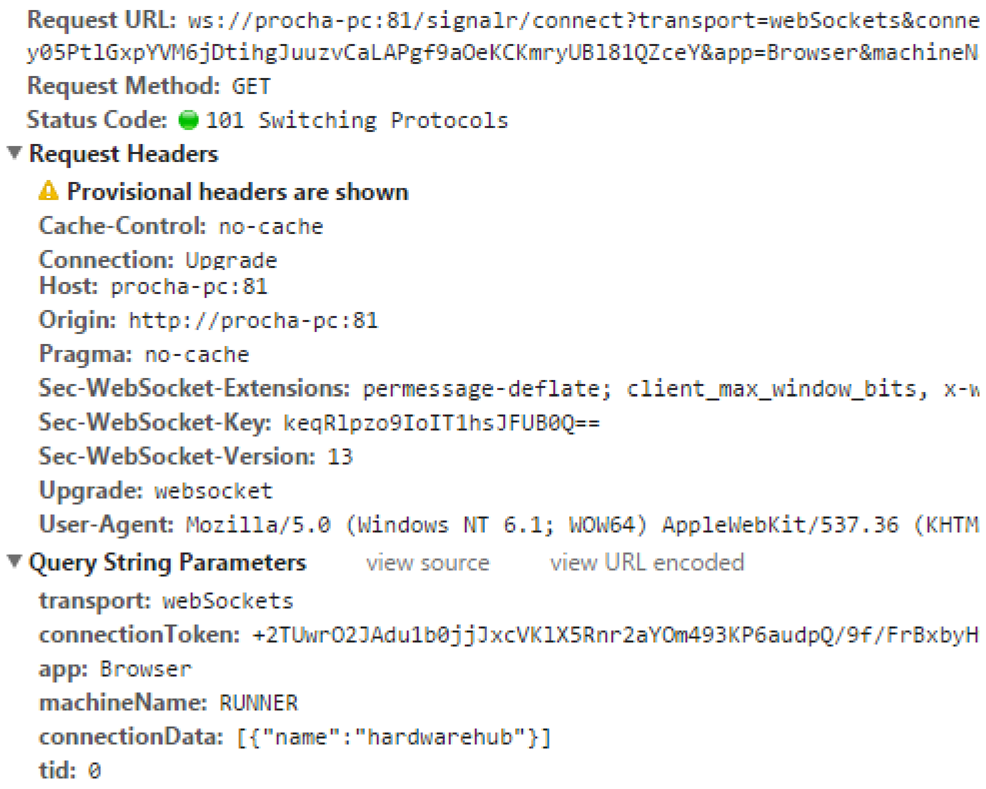
The Figure 6.1 is the negotiation request that the client sends to the server, indicating the machine name and the hub name that it wants to connect to. This is important because the server now knows the client name and in what hub it should be connected so it receives notifications from hardware in the same machine.



```
▼ {Url:/signalr,…}
   ConnectionId: "68b2eee9-d8af-417e-90c2-ab32dfa7f5d5"
   ConnectionToken: "+2TUwrO2JAdu1b0jjJxcVK1X5Rnr2aYOm49
   DisconnectTimeout: 10
   KeepAliveTimeout: 6
   ProtocolVersion: "1.3"
   TransportConnectTimeout: 5
   TryWebSockets: true
   Url: "/signalr"
```

Figure 6.2: Response to negotiation

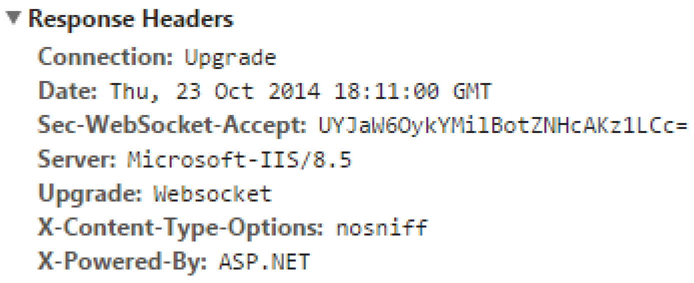
The server responds with a *ConnectionId* indicating the connection unique identification; a *ConnectionToken* (secure token) to be used by the client to perform any operation; a set of other parameters configured in the server application; and the flag which indicates if the server has support for WebSockets (*TryWebSockets*) enabled to the client can upgrade the HTTP connection to use WebSockets in future communications. The response is in Figure 6.2.



```
Request URL: ws://procha-pc:81/signalr/connect?transport=webSockets&conne
y05Ptl GxpYVM6jDtihgJuuzvCaLAPgf9aOeKCKmryUB181QZceY&app=Browser&machineN
Request Method: GET
Status Code: ● 101 Switching Protocols
▼ Request Headers
   ⚠ Provisional headers are shown
   Cache-Control: no-cache
   Connection: Upgrade
   Host: procha-pc:81
   Origin: http://procha-pc:81
   Pragma: no-cache
   Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits, x-w
   Sec-WebSocket-Key: keqRlpzo9IoIT1hsJFUB0Q==
   Sec-WebSocket-Version: 13
   Upgrade: websocket
   User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTM
▼ Query String Parameters      view source      view URL encoded
   transport: webSockets
   connectionToken: +2TUwrO2JAdu1b0jjJxcVKlX5Rnr2aYOm493KP6audpQ/9f/FrBxbyH
   app: Browser
   machineName: RUNNER
   connectionData: [{"name":"hardwarehub"}]
   tid: 0
```

Figure 6.3: Upgrade request using WebSockets.

The request shown in Figure 6.3 made by the client after the negotiation, has the HTTP code "101" meaning that the connection will be upgraded and the protocol will change[36]. This request is made using WebSockets (on the URL the parameter is *transport=webSockets*) and the user will send the token received before to be identified in the server.



```
▼ Response Headers
   Connection: Upgrade
   Date: Thu, 23 Oct 2014 18:11:00 GMT
   Sec-WebSocket-Accept: UYJaW6OykYMilBotZNHcAKz1LCc=
   Server: Microsoft-IIS/8.5
   Upgrade: Websocket
   X-Content-Type-Options: nosniff
   X-Powered-By: ASP.NET
```

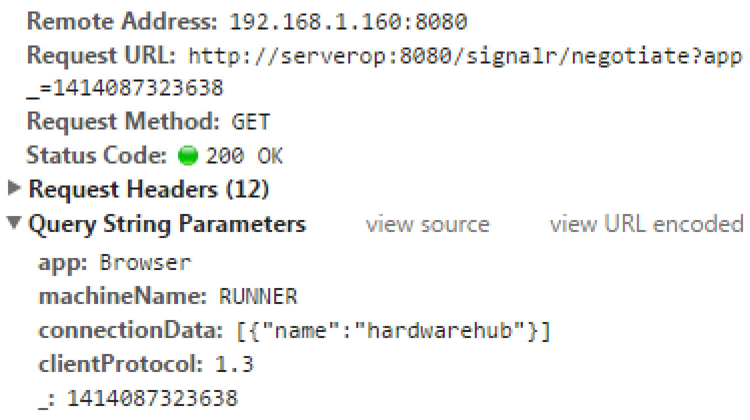
Figure 6.4: Response to upgrade request.

The response to the *Upgrade Request* is a simple acknowledge with an accept token generated in the server to assure the client of the message origin (Figure 6.4). From this point forward the communication are made using WebSockets.

6.2 PORTEIRO START WITHOUT WEBSOCKETS SUPPORT

To perform the tests that assured the minimum quality of the communications using hardware, it was asked to deploy the web server on a machine without WebSockets support and test the fallback methods making sure that client with software not up to date could run the system in what they consider good conditions. It was explained that the server must run without perceivable delay even at bursts of 5 card reads per second (multiple applications mean multiple interactions at the same moment).

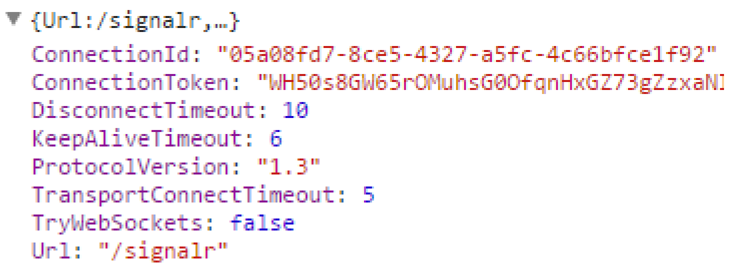
To test this, it was used a machine with the minimum requirements to the system (for the server machine): Intel© Pentium© G2030 (dual core and dual thread), 4GB of Random-Access Memory (RAM), Microsoft© Windows© Server 2008 Standard, IIS 7 with .NET framework version 4 and Microsoft© SQL Server© 2008 R2. It was deployed with a database filled with data identical to a client's database and the network bandwidth limited to 10 Mbps.



```
Remote Address: 192.168.1.160:8080
Request URL: http://serverop:8080/signalr/negotiate?app
_=1414087323638
Request Method: GET
Status Code: ● 200 OK
▶ Request Headers (12)
▼ Query String Parameters      view source      view URL encoded
   app: Browser
   machineName: RUNNER
   connectionData: [{"name":"hardwarehub"}]
   clientProtocol: 1.3
   _: 1414087323638
```

Figure 6.5: Negotiation request without WebSockets support.

The negotiation request in Figure 6.5 does not differ from the previous presented in Figure 6.1, the information is the same and only when the server replies to this request the differences appear.



```
▼ {Url:/signalr,…}
   ConnectionId: "05a08fd7-8ce5-4327-a5fc-4c66bfce1f92"
   ConnectionToken: "WH50s8GW65rOMuhsG0OfqnHxGZ73gZzxaNI
   DisconnectTimeout: 10
   KeepAliveTimeout: 6
   ProtocolVersion: "1.3"
   TransportConnectTimeout: 5
   TryWebSockets: false
   Url: "/signalr"
```

Figure 6.6: Negotiation response without WebSockets support.

The server responds (Figure 6.6) with the same parameters of the previous communication but now the flag *TryWebSockets* is disabled meaning that the client will not send an upgrade request and is the server responsibility to send notifications to the client using a different technology.

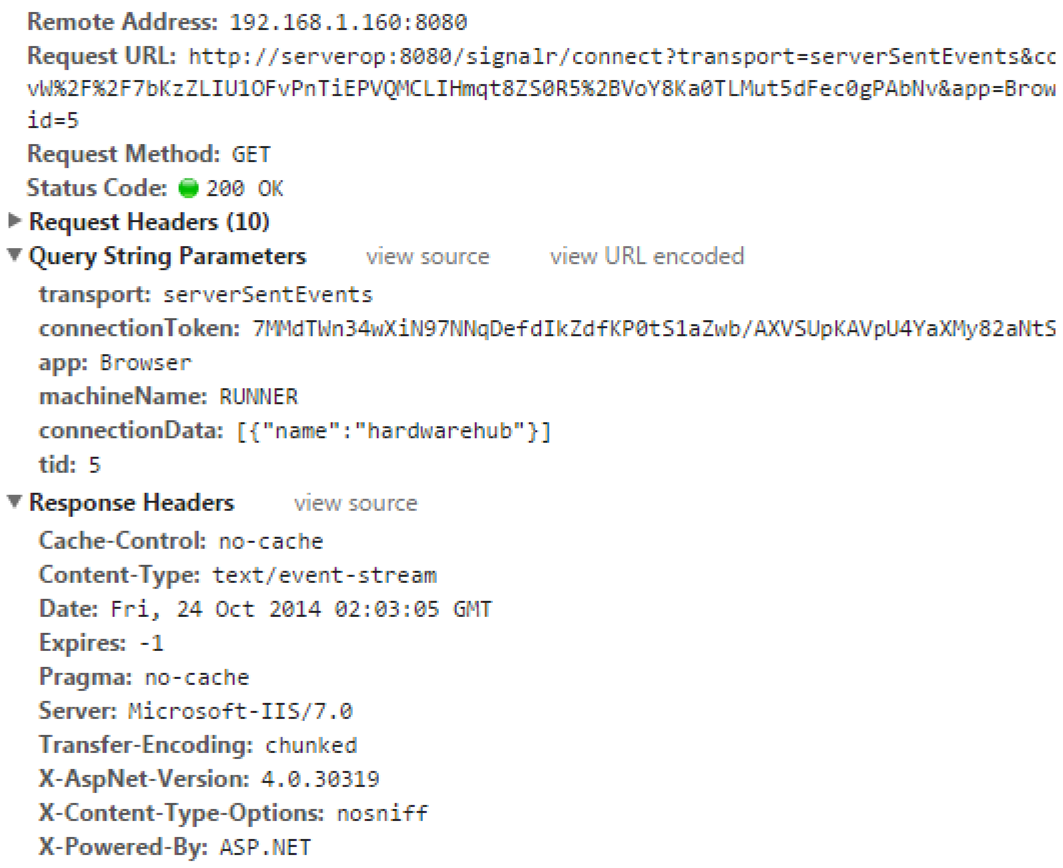

```
Remote Address: 192.168.1.160:8080
Request URL: http://serverop:8080/signalr/connect?transport=serverSentEvents&cc
vW%2F%2F7bKzZLIU1OFvPnTiEPVQMCLIHmqt8ZS0R5%2BVoY8Ka0TLMut5dFec0gPAbNv&app=Brow
id=5
Request Method: GET
Status Code: ● 200 OK
▶ Request Headers (10)
▼ Query String Parameters     view source     view URL encoded
   transport: serverSentEvents
   connectionToken: 7MMdTWn34wXiN97NNqDefdIkZdfKP0tS1aZwb/AXVSUpKAVpU4YaXMy82aNtS
   app: Browser
   machineName: RUNNER
   connectionData: [{"name":"hardwarehub"}]
   tid: 5
▼ Response Headers     view source
   Cache-Control: no-cache
   Content-Type: text/event-stream
   Date: Fri, 24 Oct 2014 02:03:05 GMT
   Expires: -1
   Pragma: no-cache
   Server: Microsoft-IIS/7.0
   Transfer-Encoding: chunked
   X-AspNet-Version: 4.0.30319
   X-Content-Type-Options: nosniff
   X-Powered-By: ASP.NET
```

Figure 6.7: Communication using Server-Sent Events.

Because the browser used was Google Chrome the fallback technology is Server-Sent Events, and in the transport parameter it's shown exactly that (‘*transport: serverSentEvents*’ in Figure 6.7). The server responds specifying format of the messages (*event-stream* the format type for messages sent by servers using Server-Sent Events or any event stream technology).

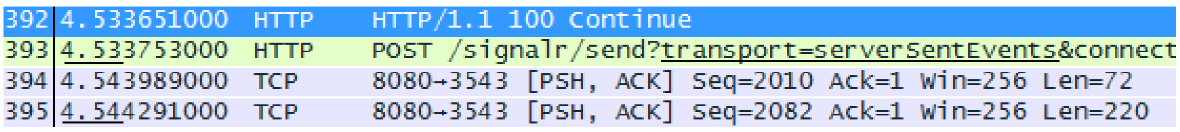
6.3 CONNECTION AND OPERATIONAL STRESS TEST.

To test the capacities of the connection in a case where WebSockets aren't available, the last server was used, and in the client was installed the *HardwareInterface* service to have card read operations multiple times per second.

To perform this test in the client was configured a Radio-Frequency Identification (RFID) card reader on Communication Port (COM)6 – this configuration was entered in the database so the server would send it to the client when the *HardwareInterface* starts – and two additional applications were used (Docklight[37] version 2 and Realterm[38] version 2) to send card serial numbers to port COM7. It was used two USB to Universal Asynchronous Receiver/Transmitter (UART) bridges connected to port COM6 and port COM7 so when information was sent to port COM7 it would be written into port COM6 where the *HardwareInterface* service was listening.

The first test was made sending 10 card serial numbers per second (or one every 0,1 seconds): in the *HardwareInterface* service log was noticed that it would send packets every 0,1092 seconds, meaning that it induces a delay of 0,0092 seconds for every card read which, considering that every card reader has a delay between reads in the order of the tens of milliseconds, is sufficient to process every card read possible up to 100 per second (which the current card readers cannot reach).

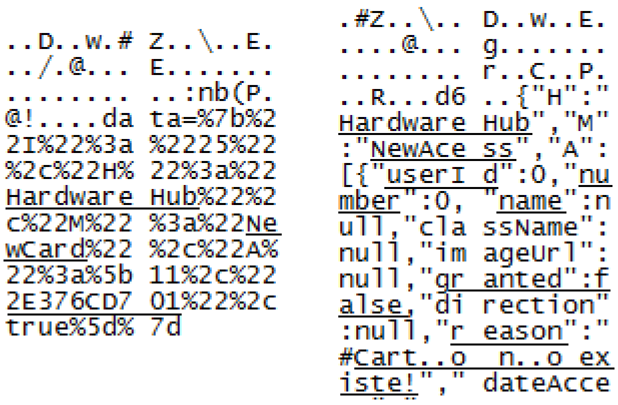
In the server side the user access is registered in the database and sent back to the client, so the software Wireshark[39] was used to measure the time between sending the card read packet to the server and obtaining the response.



```
392 4.533651000  HTTP    HTTP/1.1 100 Continue
393 4.533753000  HTTP    POST /signalr/send?transport=serverSentEvents&connect
394 4.543989000  TCP     8080→3543 [PSH, ACK] Seq=2010 Ack=1 Win=256 Len=72
395 4.544291000  TCP     8080→3543 [PSH, ACK] Seq=2082 Ack=1 Win=256 Len=220
```

Figure 6.8: Wireshark packets capture.

The packet number 393 - shown in Figure 6.8 - corresponds to the client's packet with the card read information, and the 395 is the response with the information regarding the user access. It's highlighted as well the transport flag used by the client. The time between packets to the server is about 0,1 seconds and the response to each takes about 0,011 seconds further indicating that the server has capability of responding close to 100 card reads per second (approximately 90 card reads in the server side).



```
..D..w.# Z..\..E.
../.@... E.......
........ ..:nb(P.
@!....da ta=%7b%2
2I%22%3a %2225%22
%2c%22H% 22%3a%22
Hardware Hub%22%2
c%22M%22 %3a%22Ne
wCard%22 %2c%22A%
22%3a%5b 11%2c%22
2E376CD7 01%22%2c
true%5d% 7d
```

```
.#Z..\.. D..W..E.
....@... g.......
........ r..C..P.
..R...d6 ..{"H":"
Hardware Hub","M"
:"NewAce ss","A":
[{"userI d":0,"nu
mber":0, "name":n
ull,"cla ssName":
null,"im ageUrl":
null,"gr anted":f
alse,"di rection"
:null,"r eason":"
#Cart..o n..o ex
iste!"," dateAcce
```

Figure 6.9: Communication packets payload.

In the left of Figure 6.9 there is the payload corresponding to packet 393 (from client to server) where is possible to read the SignalR hub that it's destined to (*HardwareHub*), the function call being made (*NewCard*) and the arguments of the function, in this case the card serial number ('2E376CD701'). On the right is the response, serialized in JSON to be read by the browser containing the JavaScript method to invoke (*NewAccess*) and information about the access made, in this case the card did not belong to any user so the answer is '*Card doesn't exist!*'.

The delay between card reads and presentation on the browser was imperceptible even at 10 per second, and the resources used on the server were almost irrelevant even in a machine without high performance. This was enough to understand that a minimum spec server can perform in smooth conditions even in the worst case scenario presented.

To test the server and communications even further trying to reach the capacity limits, the same test was made but with a frequency of 100 card reads per second (during 5 seconds) with 4 clients connected to the server with the same visualization of *Porteiro* and *LogViewer* to receive notifications regarding the user accesses (two clients on a Google Chrome browser and two other clients on Firefox®). This is the calculated limit at which the *HardwareInterface* service can operate and it is over the limit at which the server was responding in the last test.

With this test case the log on the *HardwareInterface* service showed that the time elapsed between the first card read and last card read was 8,84 seconds meaning that the limit at which the service can operate was passed. In fact the log showed that the maximum sent per second was 57 and minimum was 55 card reads. On the database the user access table indicates that the time between first and last accesses was 22 seconds. So the operations made to assess the user access (read and write into database and write into the log database), send to all clients (in this case 4 clients) took 22 seconds for 500 card reads. The resources on the server side were not 100% in use: the IIS server had 34 threads using 37% of the Central Processing Unit (CPU) capabilities; the SQL server was using 41 threads and averaging 6% of the CPU usage. This means that the limit is not on the resources but on the frameworks in use as well as some logic constraints regarding concurrent operations (limiting the minimum requirement system to 23 operations per second).

6.4 STRESS TEST WITH WEBSOCKETS

It was not possible to perform the same test on the same machine, due to licensing costs for the Microsoft© Windows© 2012 Server. But the same test was performed using a newer server: Intel© Core™ i7 (4 cores and 8 threads), 16GB of RAM, IIS 8.5, Microsoft© Windows© 8, .NET 4.5 and Microsoft© SQL Server© 2012. With this server the response to 100 card reads per second was almost immediate. The struggle now was on the client side, that had its resources on maximum usage, running the JavaScript necessary to show the accesses in the *Porteiro* application. Because the hardware specs are much higher in this case, most of the performance improvement brought by the use of WebSockets may be unnoticed, but, because the applications and database are optimized to work with IIS 8.5, .Net 4.5 and Microsoft© SQL Server© 2012 and above, such performance improvements are expected.

6.5 TESTS CONCLUSIONS

The first tests shown were made to prove the fail safe mechanisms of the SignalR framework, and to understand how they perform. It is clear that it works for older versions of operating systems and browsers, as it was expected from the framework presentation.

Then a stress test was made with two different goals: to understand what are the minimum requirements of the system; and how the minimum required specs perform under stress. The test shows that a system with the minimum requirements will perform with no problems in a real implementation, given the typical usage that Micro I/O staff as knowledge of. Because every school has different needs, the requirements will be proposed accordingly, but it is safe to set the minimum requirements as they are.

The last test was performed to confirm what was already suspected: that the communications, using WebSockets on a high spec server, would run smoothly, even on situations that are known to be extreme. On a conference where SignalR was presented by the creators[10], its was shown a service[40] running on a similar server, that had over 250 clients connected to it and exchanging information at a rate of 20 updates per second (20Hz)[10], during the conference. The application seemed to run smoothly even with that amount of load, so it was expected that SignalR wouldn't represent the "bottleneck" performance, of our web application, if there is one.

CHAPTER 7

Conclusion

Real time communications is a very broad subject that had in its early days multiple techniques and workarounds to make it work for the expanding web, mostly because of the lack of support from the existing technologies and protocols, but, not less important because the very term "*Real Time*" has different interpretations for two different purposes. With the expansion of services provided and crescent number of devices and clients some of them became obsolete and had to be replaced or transformed to meet the needs of the service providers and users.

These emerging technologies and techniques have so many varieties and are used in such a big scale that on one hand they are well documented and there are numerous sources of information regarding them, but on the other hand they are so many and sometimes have so little differences that studying them all would be very difficult, so the decision of the ones to study and present was not an easy task itself. The choice was mainly based on the most used techniques that were relevant to show an evolution in the field.

Because of the specific nature of this thesis and because of the deadlines proposed, the study of the technologies and testing was very brief before the beginning of the development and that study had to be repeated further into the development to better understand the technologies. The choice of the WebSockets platform was made by the developer's team leader based on this research, and because its part of ASP.NET, using tools already familiar to the developers team.

It wasn't a difficult conclusion about SignalR being the framework to use because it had so many features already working and great support team behind it, whereas the other frameworks were in some cases simple implementations of communications using WebSockets. The framework that stood as a candidate was PokeIn but the fact that isn't free and some development issues had to be solved by their support team made the choice clear.

The testing of the frameworks gave a real look into what it would be like to use them, by building a simple chat application. One of them was not possible to build successfully due to the lack of documentation and samples provided, and even that the others were not a difficult task, SignalR stood out by being extremely easy to use, and by having a really small learning curve.

In the approach to the hardware and browser integration issue, there were many limitations regarding the technologies being used and how to compare them. During the research it was obvious that some software manufacturers have their own implementations and maybe new techniques that aren't available to the public – which is understandable considering that if they have the same issue and develop a solution they're not obliged to provide it. Comparing these techniques was an extensive task and the deadlines of the development of the web application didn't allow for that, instead, it was used a similar approach to an existing implementation in the current version of SIGE$^{©}$, and all it was required was to test it, find possible problems and correct them – which was accomplished.

The LogViewer was a bonus solution, since the requirements were to simply log the system's operations, and the real time aspect of it was not on the table in the beginning then becoming a crucial point. The possibilities to the LogViewer are extensive and not yet all discussed with the possibility of it becoming a product and present it as a feature of the system or to be sold separately. To complement the existing LogViewer there has to be made some changes and implement inside the web applications methods to turn on and off levels of logging – this is thought to be implemented in the first release of the product to help in the client support department.

The product is now on four pilot schools, with some training being given to school staff on how are operations now performed. With the feedback expected from these schools there will be a clearer image of what has to be improved and how does the new web application compares to the current desktop application.

The general opinion about the usability and feel of the application – by people at Micro I/O and in my experience – is that it's somewhat less responsive than the current product. Being the start of the deployment it's understandable that it may not behave as a similar product that the users were accustomed to and that has many years of updates and improvements. The probable cause of this less responsive feel may be related to the browser engine, and JavaScript processing (that is the general belief among the team). To help improve, a plugin feature is under development to add or remove features, providing the clients with a custom system with the necessary features to each, removing unnecessary parts that contribute to the overall performance.

7.1 FUTURE WORK

When the feedback from the pilot schools starts, it will be possible to fix some problems, change some user interactions, to provide a better service.

To improve the web applications interface responsiveness, the plugin platform is underway. Because the development had that consideration form the beginning, it is easier to adapt the components developed for each client application to plugins. One feature that is being discussed to improve the client applications, is to load only the components that the operator has permission to use. Because all operations have permissions constraints that have to be specified for each operator, this can ease the load in the browser, which is thought to be behind the lack of responsiveness.

It's not decided yet the future for *LogViewer*. One certainty is that it's being deployed to the pilot schools to record their interactions. It is possible that it will have further development, to allow changing the level of logging remotely; send critical errors directly to Micro I/O (as a new SignalR target), to be analyzed and possibly corrected before the client calls for support.

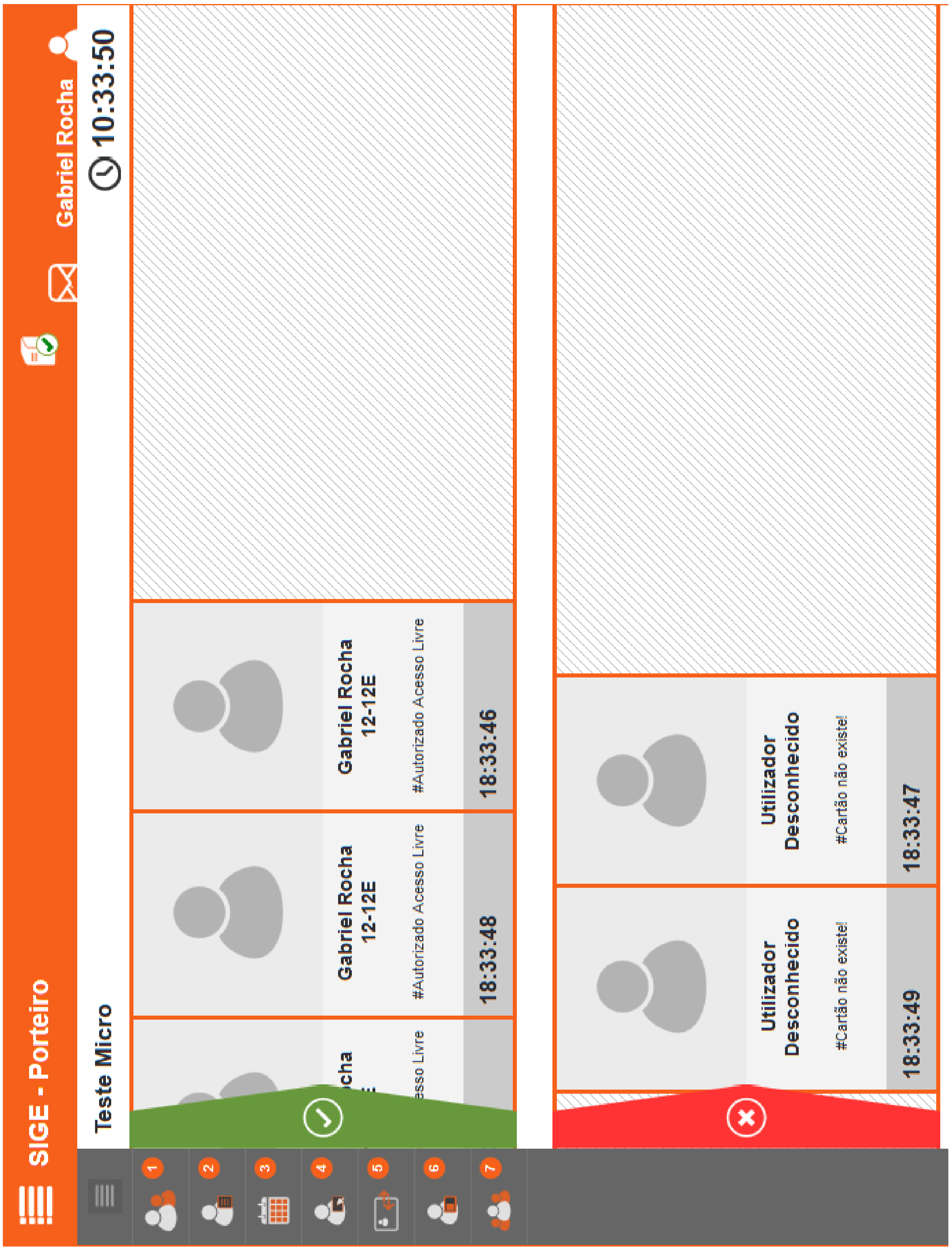
Appendices

APPENDIX A

Application Interface

A.1 *porteiro*

To have a glimpse into the *Porteiro* application's interface, the Figure A.1 is a screen capture taken during the tests. The interface may have some faults due to the styles of the page being under update at the time. It represents the application horizontal "timeline" view, the main view of the application where accesses by users are shown.



Gabriel Rocha
12-12E
#Autorizado Acesso Livre
18:33:46

Gabriel Rocha
12-12E
#Autorizado Acesso Livre
18:33:48

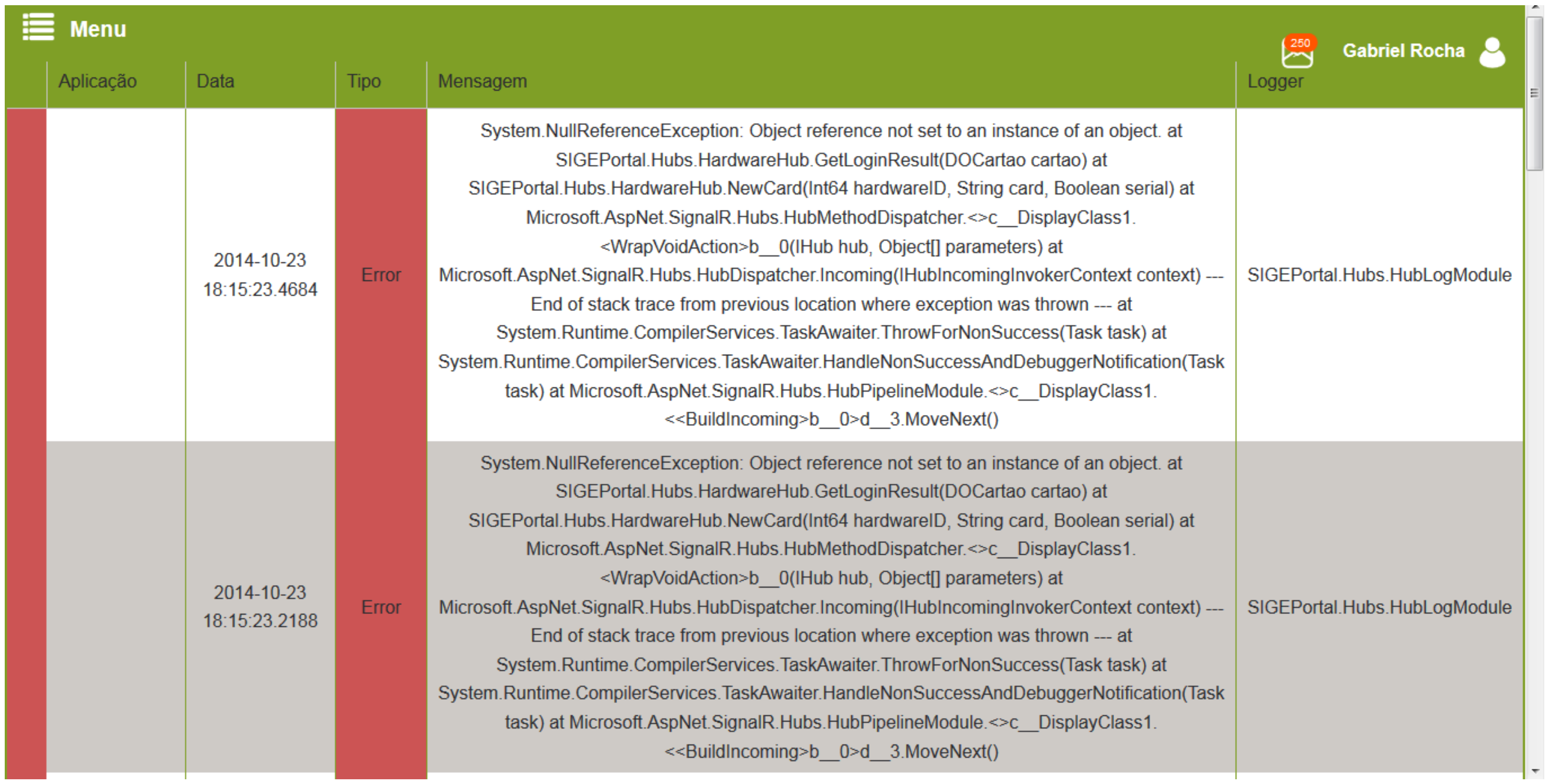
Utilizador Desconhecido
#Cartão não existe!
18:33:47

Utilizador Desconhecido
#Cartão não existe!
18:33:49

Figure A.1: *Porteiro* interface

A.2 *logviewer*

The *LogViewer* application interface is not yet designed, nor planed. It is for now a grid view showing all the log entries in the database, with filter capabilities, and color scheme for easy identification of the log level. It has a counter on the top right, showing how many new log entries were created after opening the page, which is updated in real time as discussed. Because it is still used as a tool by the developers there wasn't much effort into its appearance, as show in Figure A.2.



	Aplicação	Data	Tipo	Mensagem	Logger
🟥		2014-10-23 18:15:23.4684	Error	System.NullReferenceException: Object reference not set to an instance of an object. at SIGEPortal.Hubs.HardwareHub.GetLoginResult(DOCartao cartao) at SIGEPortal.Hubs.HardwareHub.NewCard(Int64 hardwareID, String card, Boolean serial) at Microsoft.AspNet.SignalR.Hubs.HubMethodDispatcher.<>c__DisplayClass1.<WrapVoidAction>b__0(IHub hub, Object[] parameters) at Microsoft.AspNet.SignalR.Hubs.HubDispatcher.Incoming(IHubIncomingInvokerContext context) --- End of stack trace from previous location where exception was thrown --- at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task) at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task) at Microsoft.AspNet.SignalR.Hubs.HubPipelineModule.<>c__DisplayClass1.<<BuildIncoming>b__0>d__3.MoveNext()	SIGEPortal.Hubs.HubLogModule
🟥		2014-10-23 18:15:23.2188	Error	System.NullReferenceException: Object reference not set to an instance of an object. at SIGEPortal.Hubs.HardwareHub.GetLoginResult(DOCartao cartao) at SIGEPortal.Hubs.HardwareHub.NewCard(Int64 hardwareID, String card, Boolean serial) at Microsoft.AspNet.SignalR.Hubs.HubMethodDispatcher.<>c__DisplayClass1.<WrapVoidAction>b__0(IHub hub, Object[] parameters) at Microsoft.AspNet.SignalR.Hubs.HubDispatcher.Incoming(IHubIncomingInvokerContext context) --- End of stack trace from previous location where exception was thrown --- at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task) at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task) at Microsoft.AspNet.SignalR.Hubs.HubPipelineModule.<>c__DisplayClass1.<<BuildIncoming>b__0>d__3.MoveNext()	SIGEPortal.Hubs.HubLogModule

Figure A.2: *LogViewer* interface

APPENDIX B

Chat application code snippets

To understand each WebSocket framework in more detail, it was implemented a simple chat application using each framework. The code in this chapter was selected from the referred projects to show some specifics of each framework.

B.1 SignalR chat application

Client side

```
//this script is created by the framework, it handles all connection
   related tasks.
<script src=''/signalr/hubs''></script>

<script>
  $(function () {
    //connect to the server
    var conn = $.connection.chatHub;

    //define function for when a message is received
    conn.client.newMessage = function (data) {
      $(''#messages'').append(''<li>'' + data + ''</li>'');
    };

    //when the connection starts, functions can be defined here
    $.connection.hub.start().done(function() {
      //define function for Send button.
      $(''#send'').click(function () {
        var msg = $(‘‘#msg’’).val();
        //use connection to make RPC
        conn.server.send(msg);
      });
    });
  });
</script>
```

SERVER SIDE

```
namespace SignalR_test.Chat
{
  public class ChatHub : Hub
  {
    //function used by the clients through RPC
    public void Send(string message)
    {
      //sends the message to all connected clients
      Clients.All.newMessage(message);
    }
  }
}
```

B.2 SUPERWEBSOCKET CHAT APPLICATION

CLIENT SIDE

```
//function to send messages
function sendMessage() {
  //Only supports websockets
  if (ws) {
    //find the messageBox value in the page
    var messageBox = document.getElementById('messageBox');
    //sends the value in the messageBox element
    ws.send(messageBox.value);
    //resets the messageBox
    messageBox.value = "";
  } else {
    alert(noSupportMessage);
  }
}
```

```
//connect client to server
function connectSocketServer() {

  // create a new websocket and connect
  ws = new window[support]('ws://<%= Request.Url.Host %>:8080/sample');

  //when data is received from the server, this metod is called
  ws.onmessage = function (evt) {
    //writes the message on the messageBoard
    messageBoard.append("# " + evt.data + "<br />");
    scrollToBottom(messageBoard);
  };

  // when the connection is established, this method is called
  ws.onopen = function () {
    messageBoard.append('* Connection open<br/>');
  };

  // when the connection is closed, this method is called
  ws.onclose = function () {
    messageBoard.append('* Connection closed<br/>');
  }
}
```

SERVER SIDE

```
public class Global : System.Web.HttpApplication
{
    private IBootstrap m_Bootstrap;
    private WebSocketServer m_WebSocketServer;

    void Application_Start(object sender, EventArgs e) {
        StartSuperWebSocketByConfig();
    }

    void StartSuperWebSocketByConfig() {
        m_Bootstrap = BootstrapFactory.CreateBootstrap();
        var socketServer = m_Bootstrap.AppServers.FirstOrDefault(s => s.
           Name.Equals(``SuperWebSocket'')) as WebSocketServer;

        //define functions to implement in the application
        socketServer.NewMessageReceived += new SessionHandler<
           WebSocketSession, string>(socketServer_NewMessageReceived);
    socketServer.NewSessionConnected +=
        socketServer_NewSessionConnected;
    socketServer.SessionClosed += socketServer_SessionClosed;

    m_WebSocketServer = socketServer;
    m_Bootstrap.Start();
}

void socketServer_NewMessageReceived(WebSocketSession session, string
     e) {
    SendToAll(session.Cookies[``name''] + ``:'' + e);
}

void SendToAll(string message) {
    //there has to be an iteration through sessions to send a message
         to all clients
    foreach (var s in m_WebSocketServer.GetAllSessions())
    {
        s.Send(message);
    }
}

/*Set of functions that have to be handled by the developer.*/

void socketServer_SessionClosed(WebSocketSession session, CloseReason
     reason) {
    //When the socketServer closes the session it has to be handled
        by the developer
}

void Application_End(object sender, EventArgs e) {
    if (m_Bootstrap != null)
        m_Bootstrap.Stop();
}

void Application_Error(object sender, EventArgs e) {
    // Code that runs when an unhandled error occurs
}

void Session_Start(object sender, EventArgs e) {
    // Code that runs when a new session is started
}

void Session_End(object sender, EventArgs e) {
    // Code that runs when a session ends.
    }
}
```

B.3 ALCHEMY WEBSOCKETS CHAT APPLICATION

CLIENT SIDE

```
//parse message sent from the server
function ParseResponse(response) {
    var data = JSON.parse(response);

    if (data.Type == 3) {
    var message = data.Data.Message;
    LogMessage(message);
    }
}

//writes the message parsed on 'results'
function LogMessage(message) {
  var p = $('<p></p>').text(message);
    $('#results').prepend(p);
}

function Connect() {
  // Set up the Alchemy client object
  AlchemyChatServer = new Alchemy({
    Server: $('#server').val(),
    Port: $('#port').val(),
    Action: 'chat',
    DebugMode: true
  });

  $('#status').removeClass('offline').addClass('pending').text('
      Connecting...');

  //when the server connects, this funtion is called
  AlchemyChatServer.Connected = function() {
    $('#status').removeClass('pending').addClass('online').text('Online')
        ;
    $('#connectToServer').hide('fast', function() { $('#registerName').
        show('fast'); });
  };
  AlchemyChatServer.Disconnected = function() {
    LogMessage('Connection closed.');
    $('#status').removeClass('pending').removeClass('online').addClass('
        offline').text('Offline');
    $('#registerName, #sendMessage').hide('fast', function() { $('#
        connectToServer').show('fast'); });
  };

  //when a message is received, it's handled here
  AlchemyChatServer.MessageReceived = function(event) {
    ParseResponse(event.data);
  };

  //after all functions are defined, it is time to start the app.
  AlchemyChatServer.Start();
};
```

SERVER SIDE

```
class Program {
  //list of online users.
  protected static ConcurrentDictionary<User, string> OnlineUsers = new
      ConcurrentDictionary<User, string>();

  // Initialize the application and start the Alchemy Websockets server
  static void Main(string[] args) {
      // Initialize the server on port 81, accept any IPs, and bind
         events.
      var aServer = new WebSocketServer(81, IPAddress.Any)
                       {
                           OnReceive = OnReceive,
                           OnSend = OnSend,
                           OnConnected = OnConnect,
                           OnDisconnect = OnDisconnect,
                           TimeOut = new TimeSpan(0, 5, 0)
                       };

      aServer.Start();
  }

  // Event fired when a client connects to the Alchemy Websockets server
     instance.
  public static void OnConnect(UserContext context)  {
    var me = new User {Context = context};
    OnlineUsers.TryAdd(me, String.Empty);
}

// Event fired when a data is received from the Alchemy Websockets
   server instance.
public static void OnReceive(UserContext context) {
    var json = context.DataFrame.ToString();
    dynamic obj = JsonConvert.DeserializeObject(json);

    switch ((int)obj.Type) {
        case (int)CommandType.Message:
          ChatMessage(obj.Message.Value, context);
          break;
    }
}

/// Event fired when a client disconnects from the Alchemy Websockets
   server instance.
public static void OnDisconnect(UserContext context) {
    var user = OnlineUsers.Keys.Where(o => o.Context.ClientAddress ==
       context.ClientAddress).Single();
    OnlineUsers.TryRemove(user, out trash);
}

// Broadcasts a chat message to all online users
private static void ChatMessage(string message, UserContext context) {
    var u = OnlineUsers.Keys.Where(o => o.Context.ClientAddress ==
       context.ClientAddress).Single();
    var r = new Response {Type = ResponseType.Message, Data = new {u.
       Name, Message = message}};
    Broadcast(JsonConvert.SerializeObject(r));
}

// Broadcasts a message to all users
private static void Broadcast(string message, ICollection<User> users =
     null) {
    if (users == null) {
        foreach (var u in OnlineUsers.Keys) {
            u.Context.Send(message);
        }
    }else {
        foreach (var u in OnlineUsers.Keys.Where(users.Contains)) {
            u.Context.Send(message);
        }
        }
    }
}
```

B.4 POKEIN CHAT APPLICATION

CLIENT SIDE

```
//when the send button is pressed
btnChat.onclick = function() {
  //get te message to send
    var mess = info.value;
    var message = new ChatMessage();
    message.Message = mess;
    //call Send method defined dynamically by PokeIn
    Chat.Send(message);
    info.value = "";
}

//Define start and close functions
document.OnPokeInReady = function() {
  PokeIn.Start(function(status) {

    //PokeIn Connection Closed By Server
    PokeIn.OnClose = function() {
        ChatMessageFrom({ Username: ``SERVER'', Message: ``Your
            Connection Closed!'' });
    };
};

//function created by the PokeIn framework
function ChatMessageFrom(chatMessage) {
  chatWind.innerHTML += ``<strong>'' + chatMessage.Username + ``</strong
      >:: '' + chatMessage.Message + ``<br/>'';
  chatBase.scrollTop = chatWind.clientHeight;
}
```

SERVER SIDE

```
public class ChatApp:IDisposable
{
    //collection of all connected users
    public static Dictionary<string, string> Users = new Dictionary<
        string,string>();//clientId, username

    public ChatApp(string clientId)
    {
        _clientId = clientId;
        _username = ``'';
    }
    //when a client disconnects it has to be handled by the developer.
    public void Dispose()
    {
        lock(Names)
        {
            if(Names.ContainsKey(_username))
                if(Names[_username]==_clientId)
                {
                    Names.Remove(_username);
                    Users.Remove(_clientId);
                    Send(new ChatMessage(_username, ``Disconnected''));
                }
        }
    }

    //When PokeIn sees ChatMessage custom class as a parameter, it
        automaticly defines ChatMessage JS class on client side.
    public void Send(ChatMessage message)
    {
        //Create JSON method from custom class
        string json = JSON.Method(``ChatMessageFrom'', message); //
            ChatMessageFrom( {Username:'username', Message:'message' } );
        CometWorker.SendToAll(json);
    }
}
```

Glossary

DOM Events – Languages like JavaScript can register event handlers and listeners inside a DOM document (mouse click, key pressed, etc.), and the trigger of one of those handlers/listeners is called a DOM Event.

Event Source API – Interface used to help establish a connection to a server responsible for sending events (DOM Events).

Hardware Interfaces – are usually system services that provide an interface to the device driver.

HTTP Chunked Encoding – Specification of HTTP 1.1[8] allowing the modification of data so that it can be sent in a series of messages (chunks), with the information of each message size (chunk size) but with the option of not indicating the data total size.

HTTP Get – HTTP method used to request data from the server.

HTTP Handshake – Process of exchanging information (negotiation) of the parameters to be used in a connection (communication channel), before the interaction between the peers starts.

HTTP Long Lived Connection – (A.K.A) HTTP keep-alive is a concept where a single connection is used for multiple requests/responses as opposed from using a single connection for every interaction.

HTTP Overhead – Extra header information on HTTP packets that uses extra network resources but doesn't add useful content to the message itself.

HTTP Request – Message containing the request of data from the server, composed by the method used, optional header fields and body message[44].

HTTP Request Time Out – Status message indicating when the request message had no response from the server within its established lifetime.

HTTP Upgrade Request – Specification of HTTP 1.1[36] introducing the Upgrade header field allowing two negotiating entities to upgrade the connection to a newer version of the protocol or a different protocol.

IFrame – HTML tag (meaning inline frame) is used to embed a document in the current HTML document.

IMessageBus – name of the message bus interface used by SignalR for the communications between clients and server.

Java applet – Small Java application launched from the web browser that is executed on the JVM in a separate process.
LiveConnect API – provides access to both JavaScript and Java methods using the available JRE on the system, acting like a bridge between the two technologies.
Mono – "Open source implementation of Microsoft's .NET Framework"[45] , enables .NET software to run cross-platform.
NuGet – Packet manager integrated with Microsoft's© Visual Studio©, allows installation and creation of library packages to distribute and be reused by developers on the .NET platform.
Sandbox – security mechanism to prevent potential dangerous software to interact with other system components or software, by restricting the resources available to the software running on it (in other words, encapsulating the program in a closed "box").
Scale-out – In terms of servers' hardware scale-up means upgrading the hardware to have more memory more computing power in the same machine, but scale-out means adding more servers to the system to run the same software in parallel, these servers are usually in clusters.
TCP Sockets – Point-to-point socket providing a connection oriented and sequential data connection, implementing mechanisms for error detection and data recovery.
URL Schema – indicates the name of the scheme of the connection followed by the address of the connection, example, in '*http://www.ua.pt*' the *URL Scheme* is '*http*'.
Web-farm – It is a server cluster with the purpose to run web applications usually in scenarios where load balancing is needed.

References

[1] Caesar Sengupta. *Releasing the Chromium OS open source project.* 2009. URL: http://googleblog.blogspot.pt/2009/11/releasing-chromium-os-open-source.html (visited on 2013).

[2] Micro I/O. *Micro I/O.* 2014. URL: http://microio.pt.

[3] W3C and Tim Berners-Lee. *The Original HTTP as defined in 1991.* 1991. URL: http://www.w3.org/Protocols/HTTP/AsImplemented.html.

[4] Alessandro Alinone. *Comet and Push Technology.* 2007. URL: http://cometdaily.com/2007/10/19/comet-and-push-technology/.

[5] John Zukowski. *What does Sun's lawsuit against Microsoft mean for Java developers?* Oct. 1997. URL: http://www.javaworld.com/article/2077055/soa/what-does-sun-s-lawsuit-against-microsoft-mean-for-java-developers-.html.

[6] Damian Edwards. "Jump Start - Building Web Apps with ASP.NET". In: *Real-time Communications with SignalR.* 2013. URL: http://www.microsoftvirtualacademy.com/training-courses/create-web-apps-with-asp-net.

[7] Rob Gravelle. *Comet Programming: Using Ajax to Simulate Server Push.* 2010. URL: http://www.webreference.com/programming/javascript/rg28/index.html.

[8] Internet Engineering Task Force (IETF). *HTTP 1.1 Chunked Transfer Coding.* 1999. URL: http://tools.ietf.org/html/rfc2616%5C#section-3.6.1.

[9] Praneeth Wickramasinghe. *SignalR – Real-time application development.* 2013. URL: http://developereventlog.blogspot.pt/2013/06/aspnet-signalr-real-time-application%5C_6.html.

[10] Damian Edwards. "MS Build". In: *Building Real-time Web Apps with ASP.NET SignalR.* Redmond, Washington, 2012. URL: http://channel9.msdn.com/Events/Build/2012/3-034.

[11] Eric Bidelman. *Server Sent Events - Basics.* 2010. URL: http://www.html5rocks.com/en/tutorials/eventsource/basics/.

[12] Internet Engineering Task Force (IETF). *RFC - 6455 - The WebSocket Protocol.* 2013. URL: http://tools.ietf.org/html/rfc6455.

[13] Malte Ubl and Eiji Kitamura. *Introducing WebSockets: Bringing Sockets to the Web.* 2010. URL: http://www.html5rocks.com/en/tutorials/websockets/basics/.

[14] Internet Engineering Task Force (IETF). *HTTP 1.1 - Header Field Definitions.* 1999. URL: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html%5C#sec14.42.

[15] Zondig. *PokeIn.* 2013. URL: http://pokein.com/.

[16] Zondig. *PokeIn Feature List.* 2013. URL: http://pokein.com/Products/PokeInFeatureList.aspx.

[17] Olivine Labs LLC. *Alchemy WebSockets.* 2013. URL: http://alchemywebsockets.net/.

[18] Damian Edwards. *SignalR GitHub.* 2013. URL: https://github.com/SignalR/SignalR.

[19] Patrick Fletcher. *Introduction to SignalR.* 2014. URL: http://www.asp.net/signalr/overview/getting-started/introduction-to-signalr.

[20] Patrick Fletcher and Mike Wasson. *Introduction to Scaleout in SignalR.* 2014. URL: http://www.asp.net/signalr/overview/performance/scaleout-in-signalr.

[21] StatCounter. *StatCounter Global Stats.* 2013. URL: http://gs.statcounter.com/%5C#desktop+tablet-browser%5C_version%5C_partially%5C_combined-ww-monthly-201301-201312-bar.

[22] Wade Alcorn, Christian Frichot, and Michele Orru. *The Browser Hacker's Handbook.* Indianapolis, Indiana, 2014. URL: http://books.google.pt/books?id=mR%5C_yAgAAQBAJ.

[23] CodeJava. *LiveConnect - The API for communication between Java applet and Javascript.* 2013. URL: http://www.codejava.net/java-se/applet/liveconnect-the-api-for-communication-between-java-applet-and-javascript.

[24] Nlog. *NLog.* 2013. URL: http://nlog-project.org/.

[25] Apache Software Foundation. *Log4net.* 2013. URL: https://logging.apache.org/log4net/.

[26] Binary Fortress Software. *Real-Time Log Monitoring Made Easy!* 2014. URL: http://www.logfusion.ca/.

[27] Choon-Chern Lim. *Trace Log (Real Time Log Viewer).* 2013. URL: http://sourceforge.net/projects/tracelog/.

[28] Micro I/O. *Manual SIGE 3.* Tech. rep. Aveiro: Micro I/O, 2013. URL: http://microio.pt/.

[29] Telerik. *Kendo UI jQuery and HTML5 widgets.* 2014. URL: http://www.telerik.com/kendo-ui1.

[30] Gil Fink. *Getting Started with Responsive Web Design Development Techniques.* 2013. URL: http://java.dzone.com/articles/getting-started-responsive-web.

[31] Micro I/O. *Apresentação SIGE 3.* Tech. rep. Aveiro: Micro I/O, 2013. URL: http://microio.pt/.

[32] Kim Christensen. *NLog MethodCall Target.* 2013. URL: https://github.com/nlog/nlog/wiki/MethodCall-target.

[33] Wikipedia. *Model-view-controller.* 2014. URL: https://en.wikipedia.org/wiki/Model-view-controller.

[34] W3Schools. *ASP.NET MVC Introduction.* URL: http://www.w3schools.com/aspnet/mvc%5C_intro.asp.

[35] Telerik. *Kendo UI - Documentation and API Reference.* 2014. URL: http://docs.telerik.com/kendo-ui/api/javascript/class.

[36] Internet Engineering Task Force (IETF). *HTTP 1.1 Upgrade Header.* 1999. URL: http://tools.ietf.org/html/rfc2616%5C#section-14.42.

[37] Flachmann & Heggelbacher. *RS232 Terminal / RS232 Monitor - Version 2.0.* 2014. URL: http://www.docklight.de/.

[38] Realterm. *Realterm: Serial Terminal.* 2014. URL: http://realterm.sourceforge.net/.

[39] Gerald Combs. *Wireshark Home page*. 2014. URL: https://www.wireshark.org/.

[40] N. Taylor Mullen and ASP.NET. *SignalR ShootR*. 2013. URL: http://shootr.signalr.net/.

[41] Kerry Jiang. *SuperWebSocket, a .NET WebSocket Server*. 2014. URL: https://superwebsocket.codeplex.com/SourceControl/latest.

[42] Olivine Labs LLC. *Alchemy Websockets Example*. 2012. URL: https://github.com/Olivine-Labs/Alchemy-Websockets-Example.

[43] Jose M. Aguilar. *Incredibly simple real-time features for your web apps*. 2013. ISBN: 978-84-939659-7-6. URL: www.campusmvp.net.

[44] Internet Engineering Task Force (IETF). *HTTP 1.1 Request*. 1999. URL: http://tools.ietf.org/html/rfc2616%5C#section-5.

[45] Xamarin. *Mono Home page*. 2014. URL: http://www.mono-project.com/.