MASTER THESIS

TITLE: Design and Implementation of a bidirectional, secure and real time communication between Windows Phone 8 App and Windows Store App.

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Guillem Carles Mayol Ramis

DIRECTOR: José Manuel Yufera Gomez

DATE: February 24th 2014

Title: Design and Implementation of a bidirectional, secure and real time communication between Windows Phone 8 App and Windows Store App.

Author: Guillem Carles Mayol Ramis

Director: José Manuel Yufera Gomez

Date: February 24th 2014

Overview

Emerging multimedia applications require real-time information delivery over computer networks. Traditionally, real-time communications have been using specific transport protocols resting multi-platform support and hindering service maintenance. But the latest transport protocols for real-time web applications enable to create low-latency services without requiring native applications and easily going through firewalls and proxies.

On the last years, several frameworks offering real-time web communication have appeared. These frameworks implements different transport protocols as fall-back measure to ensure that the connection will be established independently of the technology available on client-side. As a first part of this project, some of these frameworks have been reviewed and compared given the devices targeted in this project: Windows Phone 8 and Windows 8.

The second part of this project consist in developing a real-time service using a given real-time framework and applying the most recent tendencies on Software architecture and code recycling.

Finally the solution has been tested and evaluated in two environments: on local networks and on Internet using on cloud services.

ÍNDEX

Table of Figures

CHAPTER 1. INTRODUCTION

1.1.   Context

Whether online gaming, online collaboration, streaming, sharing or just a chat: The need for communication in the Internet grows as much as the number of connected devices do; and with them the demand for safety, volume and speed. Interactivity has become as important as information itself, and instead of traditional transport protocols, a more convenient and standardized way of data exchange is now needed; not only between client and server, but also between the diversity of clients that nowadays are connected to Internet. Hence, the use of real-time web technologies is almost inevitable, as them provide cross-platform/browser connectivity and can go through proxies, firewalls and NATs easily.

Despite in this project the clients are native applications, providing the service as web service increases the flexibility and portability of it. Additionally, it is a chance to test and evaluate if real-time applications are feasible on web services.

In 2012, Microsoft presented Windows Phone 8, Windows 8 and Windows RT (OS for tablets) making Windows available in all platforms (mobile and desktop). Moreover, a new kind of Windows application with simplified user interface and oriented to be controlled using touch gestures was released under the name of "Windows Store Applications" (also known as Metro applications). The attractive of these applications is that they run on any Windows 8, RT and further versions (i.e. Windows 8.1) independently if it is a portable device or desktop one.

In this project, the Windows 8 client has been developed as "Windows Store Application" making possible to run the same client on any Windows platform from Windows 8 onwards.

Finally, a real-time web service has been deployed to study and conclude if in this scenario (devices, web service, frameworks and use-cases) a real-time communication is feasible and the limitations of this.

1.2.   Scope of the project

The aim of the project is the study and deployment of one service with real-time features between Windows 8 and Windows Phone 8 using a web service. The project includes the next phases:

1) Study and comparison of technologies available for real-time web communication given the client platforms to use.

2) Select a real–time web technology and design a service to try such technology.

3) Develop pieces of software required for the real-time service.

4) Study and apply measures to secure the service.

5) Test and evaluate the service to determine if it is feasible for production.

CHAPTER 2. REAL-TIME COMMUNICATIONS

Real-time communications are any mode of telecommunications in which the user can exchange information instantly or with negligible latency. This can use either Full or Half duplex transmission modes depending on whether the communication is simultaneously bidirectional or not, respectively (Rouse, 2008).

2.1.   Real-time web services

The first services with Real-time features were Chat platforms such as mIRC (1995) and MSN Messenger (1999). But all these platforms were developed using specific protocols for each service forcing the users to install one application for each service as well. Additionally, some of these protocols face network hazards such as proxies and firewalls that can preclude the access to the service or degrade the quality of it.

Then, technologies such AJAX and HTML5 (Hypertext Markup Language v.5) appeared and web applications started to be able to provide bidirectional communication with low or reasonable latency between client and server, making feasible a real-time web service.

The main advantages and drawbacks of real-time web applications are:

- ✓ Update and maintain web app without distributing and installing software on all the clients.
- ✓ Cross-platform support (browser dependence).
- ✓ Faster development.
- ✓ None impact for proxies, NATs and firewalls.
- ✗ Adding additional payload.
- ✗ Using none specific transports for real-time (i.e. TCP or UDP instead RTP).

2.2.   Transport technologies for real-time web services

The web was built around the idea that a client's job is to request data from a server, and a server's job is to fulfil those requests. It was conceived to be a collection of Hypertext Markup Language (HTML) pages linked one to each other to form a conceptual placeholder of information. Over time the static resources increased in number and richer items, such images, which began to be part of the web. Server technologies advanced allowing to create and update content based on queries.

Trying to bring interactivity to Web and offer a richer experience, browser scripting was introduced renaming HTML to DHTML (Dynamic HTML). But the pages still needed to be refreshed frequently to get new information from the server.

As the browser scripting evolved, new techniques appeared to improve the user interactivity. Cross Frame Communication is a technique used to load a frame from the page with new information from the server without refreshing the entire page. But what if the server has some new or additional information for the user?

2.2.1.   HTTP Polling

The first solution to this problem came from the client to poll the server at regular intervals. This solution was, and still is, inefficient and leads to stale data being displayed in web pages and applications.

2.2.2.   Forever Frame

The forever-frame technique uses HTTP 1.1 chunked encoding feature to establish a single, long-lived HTTP connection in a hidden *Iframe*. Data is increasingly pushed from the server to the client over this connection, and rendered incrementally by the web browser. This provides one-way real-time connection from server to client. But, on the other way, an additional connection is required and like standard HTTP request, this connection is established for each piece of data that needs to be sent (see Figure 1).

Figure 1 Forever frame technique.

2.2.3.   AJAX

Then, a new object *XMLHttpRequest* appeared introducing asynchronous JavaScript and XML (AJAX). This is a group of interrelated web techniques used on client-side allowing to send and receive data from the server asynchronously (in the background) without interfering with the existing page. JavaScript and the *XMLHttpRequest* object provide a method for exchanging data asynchronously between browser and server to avoid full page reloads. But it doesn't change the paradigm of one response for each request.

2.2.4.   Long-Polling

Long-polling techniques are variations of the traditional polling technique. These work by establishing a connection to the server which is held opened. When the server has more data for the client it sends that data through and it closes the connection. The client then re-establishes the connection and waits for any new data and so on. The main problem with this technique is that during the reconnection process the data on the page could be out of date. Moreover, this technique doesn't change the paradigm of one response for each request (see Figure 2).

Figure 2 Long-polling technique.

2.2.5.   Server Events

Server-sent events (SSE) is a technology that works in a similar way than long-polling mechanism, except that it does not send only one message per connection. The client sends a request and server holds a connection until a new message is ready, then it sends the message back to the client while still keeping the connection open so that it can be used for another message once it becomes available. Once a new message is ready, it is sent back to the client on the same initial connection. Client processes the messages sent back from the server individually without closing the connection after processing each message. So, SSE typically reuses one connection for more messages (called events in the context of this technology, see Figure 3).

The main drawback of this technology is only unidirectional communication from server to client. So, on the other way, an additional connection is required and like standard HTTP request, this connection is established for each piece of data that needs to be sent.
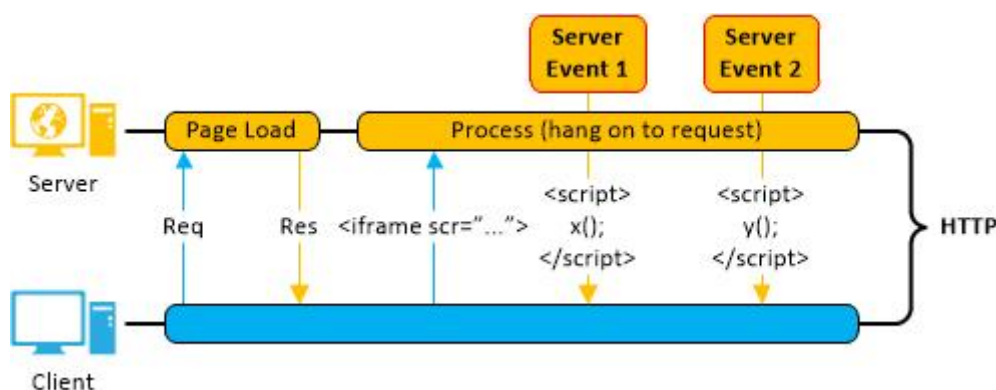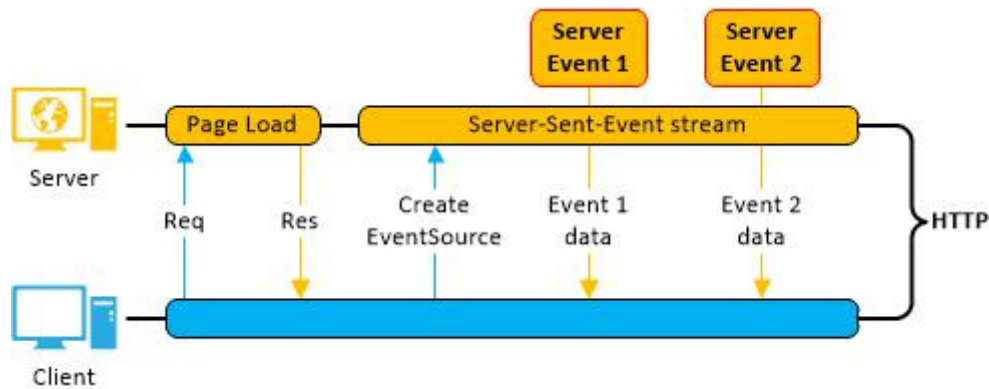
Figure 3 Server Events technology.

2.2.6.   WebSockets

The problem with all push technologies (previously announced) is that they carry the overhead of HTTP. Every time you make an HTTP request a bunch of headers and cookie data are transferred to the server. This can add up to a reasonably large amount of data that needs to be transferred, which in turn increases latency.

WebSocket technology is different from previous technologies as it provides a real full duplex persistent connection that client and server can use to start sending data at any time.

The client establishes a WebSocket connection through a process known as the WebSocket handshake. This process starts with the client, who sends a regular HTTP request to the server. An "upgrade" header included in this request informs the server that the client wishes to establish a WebSocket connection. If the server supports the WebSocket protocol, it agrees with the upgrade and communicates this through an "upgrade" header in the response (see Figure 4).

```
GET ws://websocket.example.com/
HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

```
HTTP/1.1 101 WebSocket
Protocol Handshake
Date: Wed, 16 Oct 2013
10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

Figure 4 WebSocket handshake headers in HTTP.

Now that the handshake is completed the initial HTTP connection is replaced by a WebSocket connection that uses the same underlying TCP/IP connection and *ws* (unsecure) or *wss* (secure) protocol on top. At this point either party can start sending data (see Figure 6).
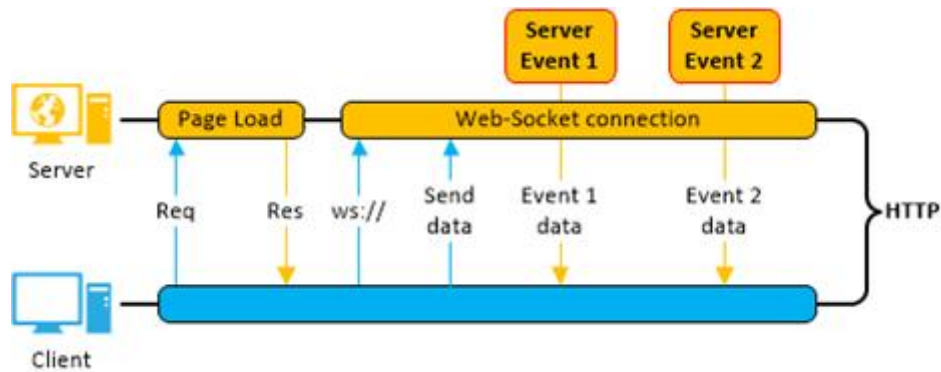
Figure 5 WebSockets technology.

2.3.   Frameworks for real-time communications

Frameworks are bridges that help to develop applications faster and easier; adding abstract layers that handle lower or complementary functionalities. The aim of these is alleviate the overhead associated with common activities performed in development, promoting code reuse. Like a technology, a framework has advantages and downsides to consider before deciding to start using one:

- ✓ Efficiency: Using pre-built functions or classes save hundreds of lines of code and very often are more than tested and optimized.

- ✓ Cost: Almost all frameworks are open-source but some of them are subject to licensing. For example Pusher (Pusher) and PubNub (PubNub) charge for the number of messages sent per day and for enhanced features like SSL protection (PubNub additionally charges for the number of active devices).

- ✓ Support: It is important to use a framework with strong and wide acceptance from developer community as the community forums are usually the best documentation about it (especially open-source ones).

- ✗ Limitation: The framework's code behaviour cannot be modified, meaning that you are forced to respect its limits and work the way it is required.

- ✗ Public Code: As it is available to everyone, it is also available to people who pretend to find and exploit the vulnerabilities of it.

Several frameworks have been considered for this project and below are listed the ones which had the requirements to deploy a service using Windows Phone 8 and Windows 8 native clients using (C# .NET). The commercial candidates have been discarded as almost all of them require using their hosting with a cost associated.

2.3.1.   Sockets.IO

Sockets.IO is a JavaScript library for real-time web applications using WebSockets as preferred transport protocol and with multiple others fall-back transport technologies.

On server-side uses node.js that is a software platform written in JavaScript that enables to run a web server without using external software such as Apache, which gives more control of how the web server works.

This library is one of the most used for real-time web applications with an extensive documentation and support from the developer community. The main constrain evaluating this library was that it doesn't have C# API itself, and it relies on an independent project called SocketIO4NET client for such API. Furthermore, the client doesn't have a stable release yet (SocketIO4NET, 2013).

2.3.2.   Xsockets

Xsockets is a real-time communication library built on Microsoft .NET Technology and provides APIs for both server and client ends. The main advantage of this platform is that it doesn't have dependencies as the transport protocols are implemented on library itself. Additionally this platform offers WebRTC support enabling direct connection between clients.

As downsides of this platform, it is not open source project and it is not as consolidated platform like SignalR.

2.3.3.   SignalR

SignalR is a library written in Microsoft .NET technology that simplifies the real-time communications for web services providing two APIs for the communication layer: *Persistent connection* and *Hub*. The first one provides access to the lower layer, which it is an abstract bidirectional and persistent communication between client and server (enabling send raw data). The second API provides access to a higher and more abstract layer, which provides serialization and remote procedure calls in both directions built-in.

Remote procedure calls (or remote invocation or remote method) are inter-process communications across a shared network allowing from a computer program to cause a subroutine to execute in another computer or device without the programmer explicitly coding the details for this remote interaction (Wikipedia).

Moreover, SignalR provides APIs for .Net client and it is officially supported for Microsoft and consequently easily portable to Microsoft cloud services such Microsoft Azure.

2.4.   Taking decisions

The table below summarizes the main characteristics involved in the decision.

Features	Sockets.IO	XSockets	SignalR
Platforms APIs	JavaScript	C# / VB.NET/ JavaScript	C# / VB.NET/ JavaScript
Transports	1. WebSockets 2. Adobe Flash Socket 3. AJAX long Polling 4. AJAX multipart Streaming 5. Forever Iframe 6. JSONP Polling	1. WebSockets (own implementation) 2. Long-Polling	1. WebSockets 2. Server Sent Events 3. Forever Frames 4. Long-Polling
WebRTC	No	Yes	No
Remote Procedures	Yes	Yes (publish / subscribe)	Yes
Grouping	Yes using Rooms	Yes (adding subjects to controllers)	Yes using groups or different *Hubs*
Hosting	Node.js	WebServer or Self-Hosted (Windows and Linux)	WebServer or Self-hosted (Only Windows*)
JavaScript Proxy	No	No	Auto generated using *Hub*
Scaling		Only Clustering	Azure, Redis, sqlServer
SSL/TLS	Yes	Yes	Yes
PCL support	N/A	No	Yes
Documentation & Support	Good and free from community (socket.io wiki) Official support not available.	Good and free from community. Official support is chargeable. (xsockets.net support)	Very good and free support from official developers. ((ASP.NET SignalR forum)

Table 1 Real-time frameworks comparison.

*Only SignalR 1.X server versions can use Mono (Mono Project) to use it on Unix.

Sockets.IO is the framework which provides more fall-back transports but the fact that it does not provide a C# API makes very difficult to develop any native application using it.

XSockets strength is the WebSockets support on all the clients that they announce but it does not provide so many fall-back transports (only Long-polling). Moreover, it does not have a PCL library so would be much more difficult to reference it from PCL (see 4.4.2).

Finally, the main advantage of SignalR is that is fully supported by Microsoft with an extended documentation and it is easily portable to Windows Azure (on cloud services of Microsoft).

CHAPTER 3. SIGNALR IN DEEP

3.1. Transport Technology

SignalR fully handles the connection management letting the developer to focus on the service to develop. Clients can broadcast messages to certain groups or just send a message to one receiver. But in both cases, only one packet between client and server is sent, and the server forwards it accordingly.

SignalR exposes a bidirectional and persistent connection independently if the real transport in use can really provide such kind of connection. The connection starts with a negotiation process using simple HTTP requests (i.e. GET) and then it is promoted to a WebSockets connection if it is available.

WebSockets is the ideal transport for real-time connections, since it makes a more efficient use of server memory, has the lowest latency and provides full duplex communications. But it also has the most stringent requirements, especially in Windows (requires Windows Server 2012 or Windows 8 with .NET 4.5). If the requirements are not met, SignalR will fall-back to other transport technologies to make its connections, such as Server Sent Events or Forever Frame. See in Figure 6 fall-back strategy of SignalR.
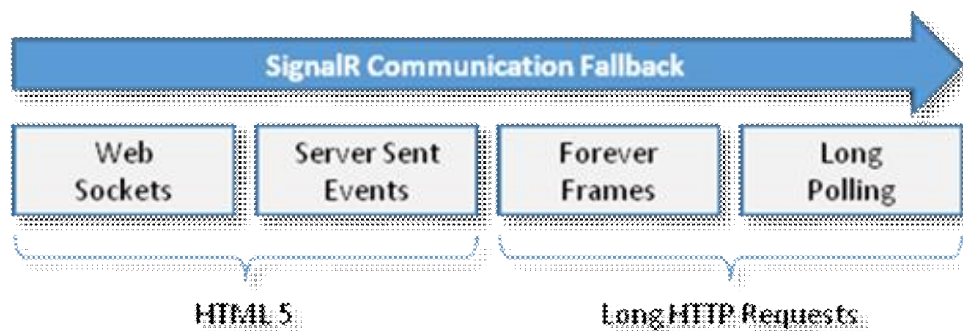

Figure 6 SignalR fall-back transport strategy.

In a consecutive way, these are the transport protocols that will be used to establish the connection:

1. WebSockets. It is the only transport providing a true persistent two-way connection between client and server. However, it is still not widely supported (i.e. .NET framework for Windows Phone 8 does not support it).

2. Server Sent Events (SSE).

3. Forever Frame (for Internet Explorer only).

4. AJAX Long-Polling.

The negotiation process takes a certain amount of time and resources in both ends (client and server). If the client capabilities are known, then a transport can be specified when the client connection is started. If not, a connection for each transport mode will be tried following the list order above.
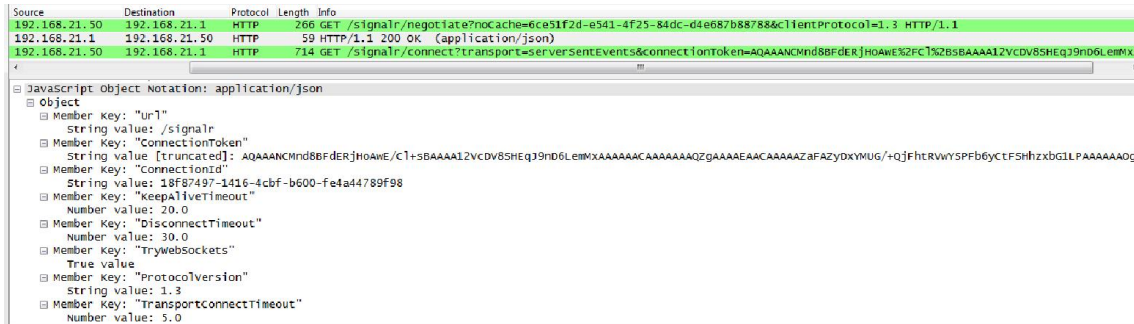

Figure 7 Transport negotiation process in SignalR.

See in Figure 7 an example of transport negotiation process, in which the client (192.168.21.50) is initiating the connection indicating the version protocol is using (1.3). Then, server (192.168.21.1) replies indicating the connection token assigned to the client (see SignalR connection token in 3.4) and some additional information about the connection. Notice that the server is announcing WebSockets support but as the client doesn't support it, client initiates the connection using SSE. This example was made using Windows Phone 8 as a client and self-hosted application running on a local machine as a server.

3.2.   Architecture

SignalR introduces two APIs for communication technologies: *Persistent connection* and *Hubs*. The first one provides access to the lower level communication protocol that SignalR exposes, allowing to send data between client and server and vice versa (message-oriented communication). The second one is a more high-level pipeline built upon the persistent connection and enables client and server to invoke remote methods (instead of exchanging mere data).

Architecture provided by SignalR is composed by a group of stacks from transport technology layer to the application layer (.NET application / JavaScript). See in Figure 8 SignalR server stack where there are four transports technologies available for the transport layer. On top, there is the *persistent connection* API handling all connection management and exposing a bi-directional communication. One layer higher, there is *Hub* API providing remote procedures, serialization, grouping and other enhanced features.
Finally, on top of all stack there is the application code where the programmer just have to worry about the service and handling few events for connection status.
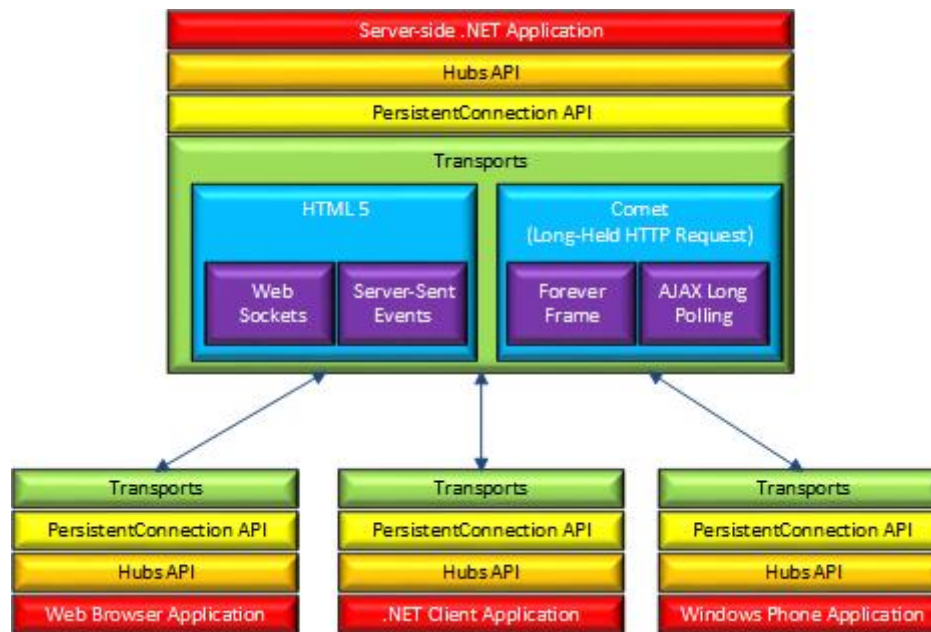
Figure 8 SignalR Architecture.

3.3.   Hubs API

Hub is the highest API exposed by SignalR enabling call remote functions in both ways. The communication from server-to-client is based on remotes procedure calls that call JavaScript or C# functions in client-side (depending on client application) from server-side. If WebSockets are not available, normal HTTP request (GET, POST) are used from client to server.

The methods and parameters are serialized using JSON but others serialization technologies can be used. If some remote procedure is not matched on the other end-side (not defined in the code) the method is not called and the procedure is discarded.

3.4.   Security

SignalR does not provide authentication or cyphering methods for user data, but it provides *[Authorize]* attribute to specify which users have access to a *hub* or method. Actually, SignalR is relaying the authentication to ASP.NET *Identity* module (known formally as Windows Identity Foundation). This is the security module provided by .NET Framework 4.5 to unify all tasks related with authentication and authorization.

SignalR needs to identify the connections to avoid commands be executed in behalf of others by sending identification information of other users. For this reason SignalR uses a connection token technique.

• **SignalR connection token**

SignalR mitigates the risk of executing malicious commands by validating the identity of the sender. For each request, both client and server pass a connection token which contains the connection id (and username for authenticated users). The connection id uniquely identifies each connection (none authenticated users) or connected client (authenticated users). This id is randomly generated by the server for each connection request and it persists for the duration of the connection (see Figure 9). The user name is provided by the authentication mechanism (if applies). The connection token is protected using encryption and digital signature.
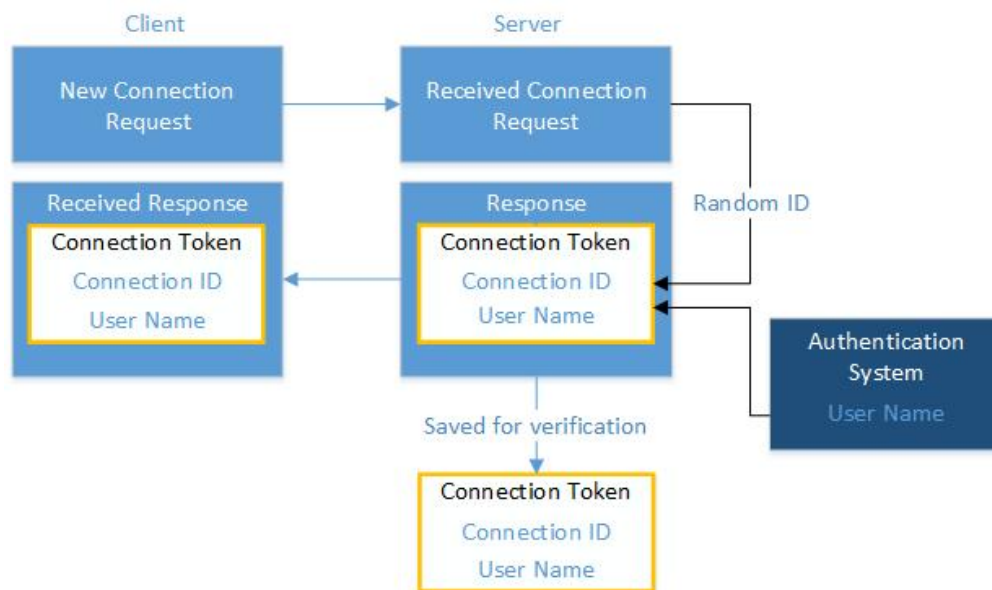

Figure 9 SignalR procedure for connection token.

For each request, the server validates the contents of the token to ensure that the request is coming from the specified user. If the user is authenticated, username must correspond to the connection id. By validating both the connection id and the username, SignalR prevents a malicious user from easily impersonating another user. If the server cannot validate the connection token, the request fails.

CHAPTER 4. SERVICE DESIGN AND IMPLEMENTATION

4.1.   Functional Service Specifications

The purpose of developing a real-time web service is to evaluate the capabilities and limitations of real-time communications on web servers using SignalR. The service provides several functionalities with different requirements to test different user cases (see Figure 10). An important fact is that the source of information to transmit (and speed rate of it) is different for each functionality; resulting in better conclusions as each one has its own requirements. The following functionalities have been implemented on the service:

1) A collaborative painting tool. The data of this service is characterized for small packets of information which are sent as the user touches the screen (burst of packets in a short periods of time but in average the bandwidth requirement is low).

2) A Chat room. The data sent on this case are small packets which are sent at sporadic basis (punctual packets with minimum bandwidth requirement).

3) Share motion sensors information (i.e. Accelerometer and Gyroscope). In this case, the data is still using small packets but they are sent constantly at the same rate (moderate and constant use of bandwidth as the packets are sent at the same interval of time constantly). Note that the interval between packets can be modified at compilation time as it is a parameter in the code.
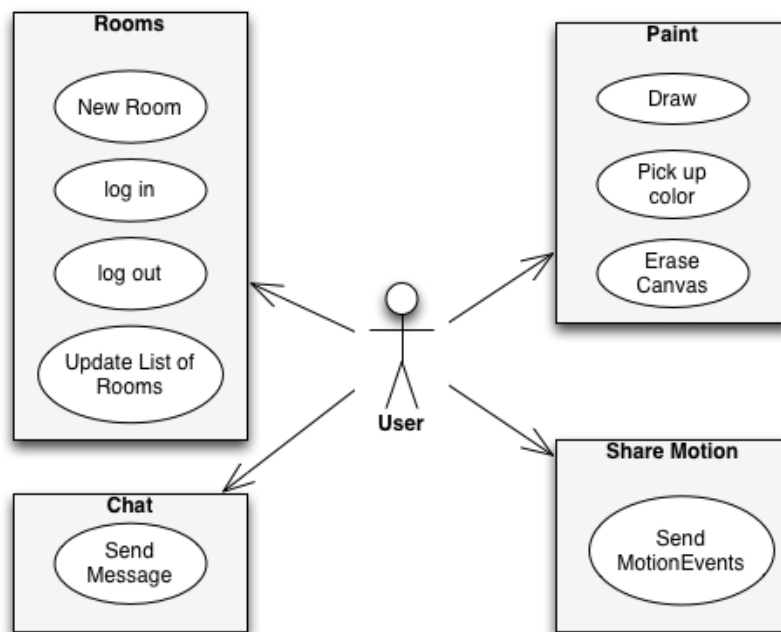

Figure 10 Use cases for the service.

To make more flexible the service and allow several users to be connected at the same time, the concept of room has been implemented. Then, the first page after opening the application is a landing page (referred as a Home page as well) with several items where each one represents a room. Rooms can be created per user request and the access to it can be restraint using a password, which is set it up in the room creation procedure. Rooms are self-deleted on timeout basis starting from the last user logged out of the room.

Once the user is in the room, another page (referred as Paint page, see Figure 11) is used to display the canvas, Chat messages and Motion sensor if it is enabled. In function of the client, some features such as seeing Motion sensors values are not available (i.e. in Windows Phone 8 only).

The canvas is the area where a user can draw and on real-time the others users can see it. It is a broadcasted sharing as all the users in the room can draw and erase whole canvas.

Another part of the service is the Chat that works broadcasting all messages to all room participants.
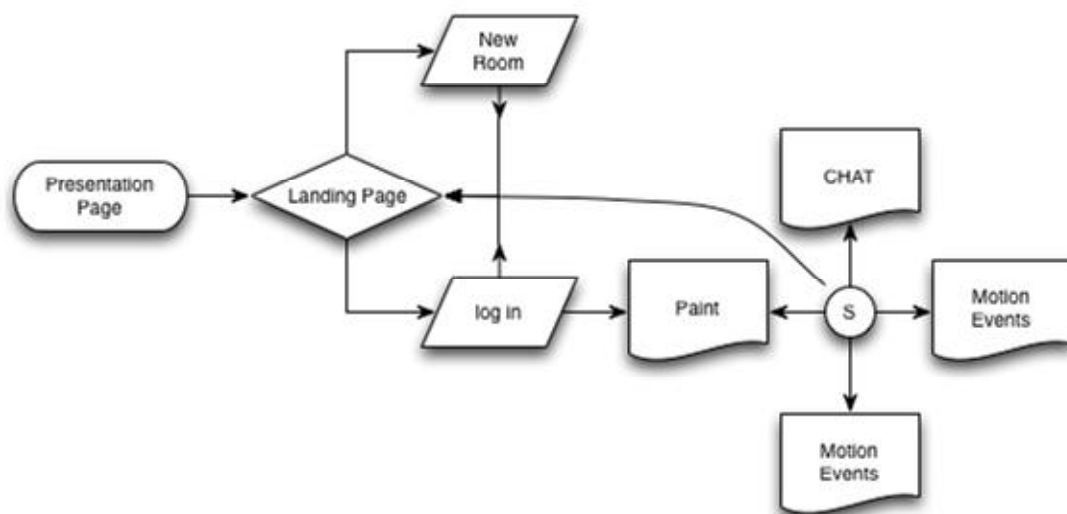

Figure 11 Flow chart for client applications.

Additionally, it is possible to receive on real-time the motion sensors values of the users that use a device with such sensors (i.e. Window Phone 8). To use this feature, the user with a capable device needs to enable Motion Seed function (client indicates to the server that wants to share motion sensors information to the rest of the participants in a room) on his application. Then, the rest of the participants in the room will start to receive this information and an indication of which user is sending it will be displayed.

4.2.   Development Service Scenario

The service is based on client-server paradigm: server provides a single point of communication between clients which are directly connected to it. This scenario has specific constrains as all the service relies on one entity and any performance or bandwidth limitation of this directly impacts on the service. But at the same time simplifies service deployment and maintenance.

Development scenario is composed of a laptop running a self-hosted instance of the server and two clients: a Windows 8 on the same laptop and a Windows Phone 8 client connected to the server using a wireless connection.
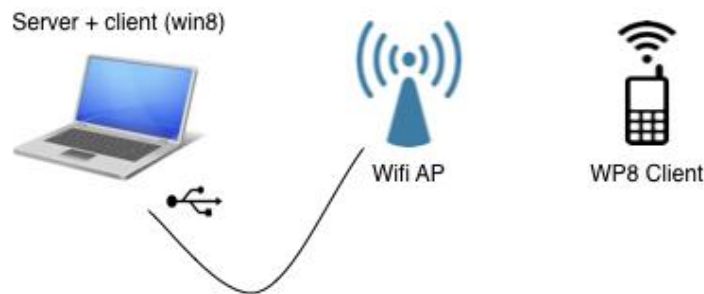

Figure 12 Development service scenario.

As seen in Figure 12, Windows Phone 8 device is accessing to the server using wireless connection (WLAN 802.3n). Wireless connectivity is characterized by a high number of collisions in the physical medium (which implies retransmissions) deteriorating connectivity performance especially for TCP connections. As the service is using HTTP, it highly depends on wireless connection performance. To minimize medium collisions and avoid the service being affected by the load on wireless access point (AP), a USB access point directly connected to the laptop has been used. Additionally, a wireless scanning was performed before setting up the development scenario to check the best WLAN channel to avoid overlapping any present wireless signal.

4.3.   Walking through client native applications

Two client applications have been developed: Windows Store App and Windows Phone 8. In this example, Windows Store App is running on Windows 8 (as seen in *Context* Windows Store App can run on multiple devices).

Once client application is open and loaded, several icons representing rooms are displayed on Home page (see Figure 13). Each room indicates how many members are logged in and if there is some password to log in (lock icon).
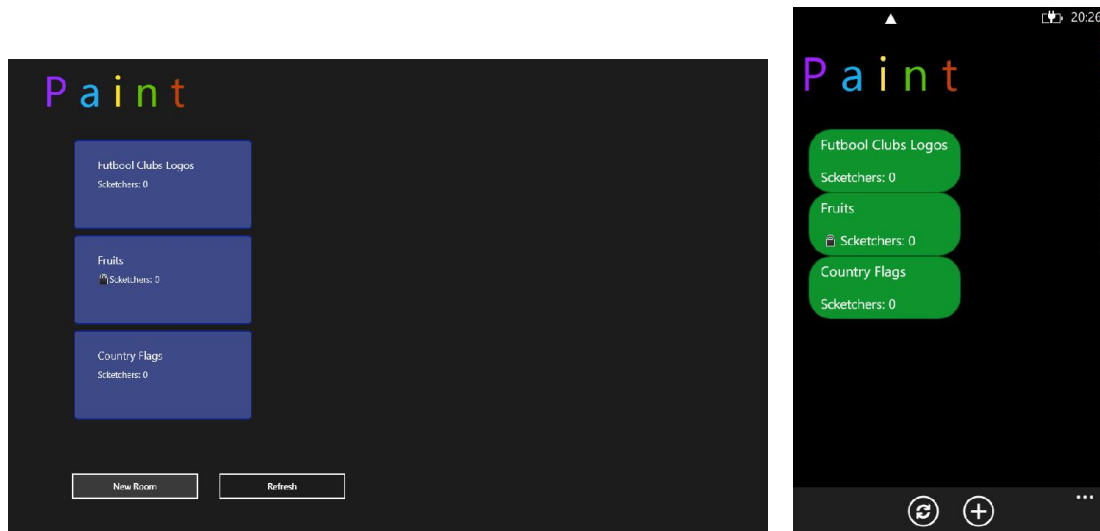
Figure 13 Home (landing) pages on client applications.

Then, the user has two options: create a new room ('+' icon on Windows Phone 8) or join an already created one (double tap on the room on both applications). Additionally, there is a button to refresh the list of rooms (rooms list is also updated by the server when there is some new information).

When a new room is requested, a popup will prompt asking to assign a room name and optionally a password to secure it (see Figure 14).
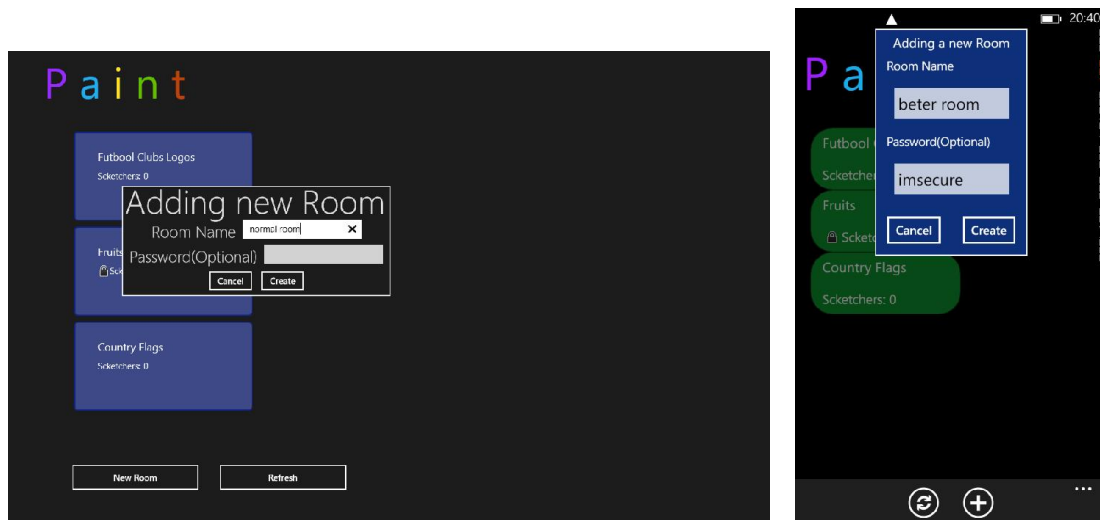

Figure 14 Adding a new room on client applications.

After filling both fields with valid values it is possible to see that the list has instantly been updated (see rooms on the background in Figure 15). To join one room is just necessary to double tap on top of the room icon and enter user name and password if it is required for that room.
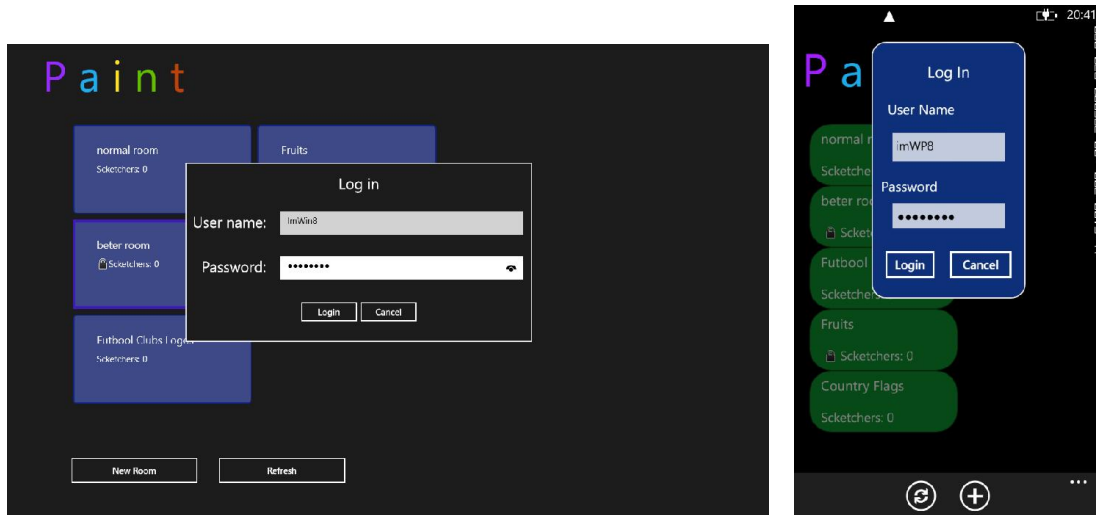
Figure 15 Login in a secured room.

If the log in is successful, the user will be redirected to the paint page which represents such room.
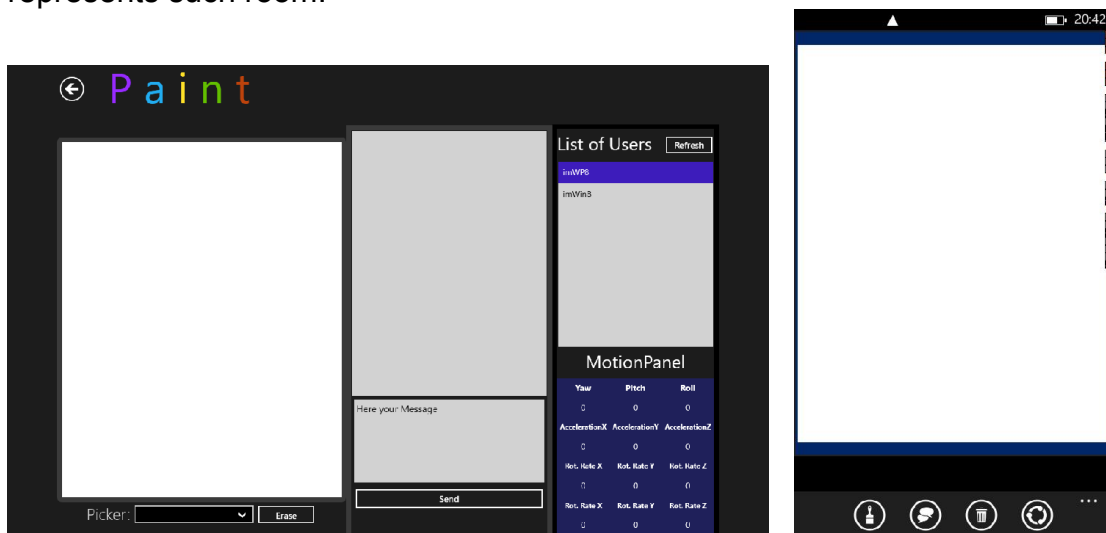

Figure 16 Paint page on client applications.

Now, the user is ready to draw using any colour available on a colour picker floating window (see Figure 16). This becomes visible using the brush icon on Windows Phone 8 or the picker button on Windows 8 (see Figure 17). To erase the whole canvas, the user can use trash icon button (any user in the room can erase it in any moment).
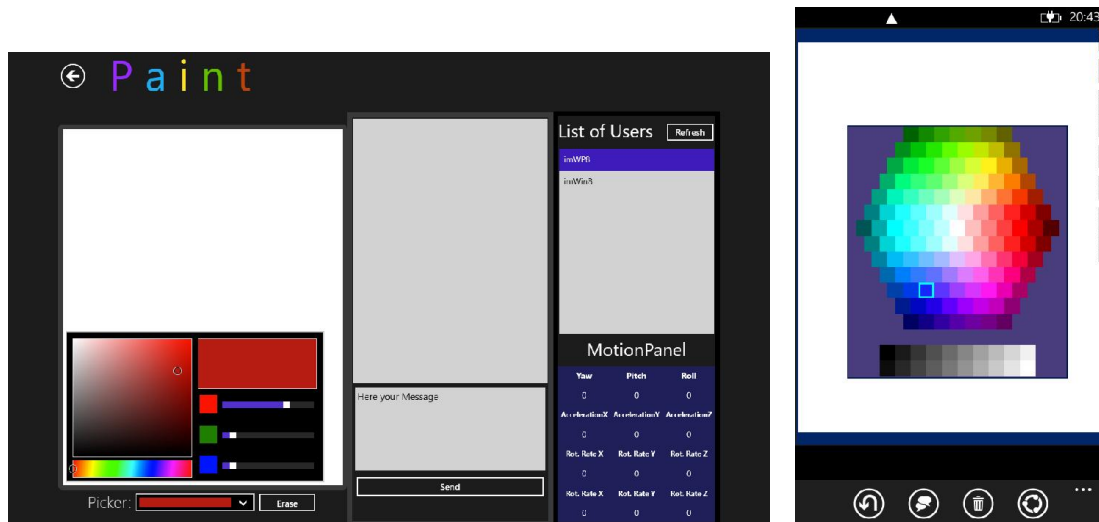
Figure 17 Colour pickers on client applications.

On Windows Store App (in this case running on Windows 8), due to enough size of screen (both tablets and laptops), the chat and motion panel are already in the Paint page (see Figure 18). Otherwise, on Windows Phone 8 the user needs to use one tap to access each functionality (i.e. dialog icon button displays the chat screen).
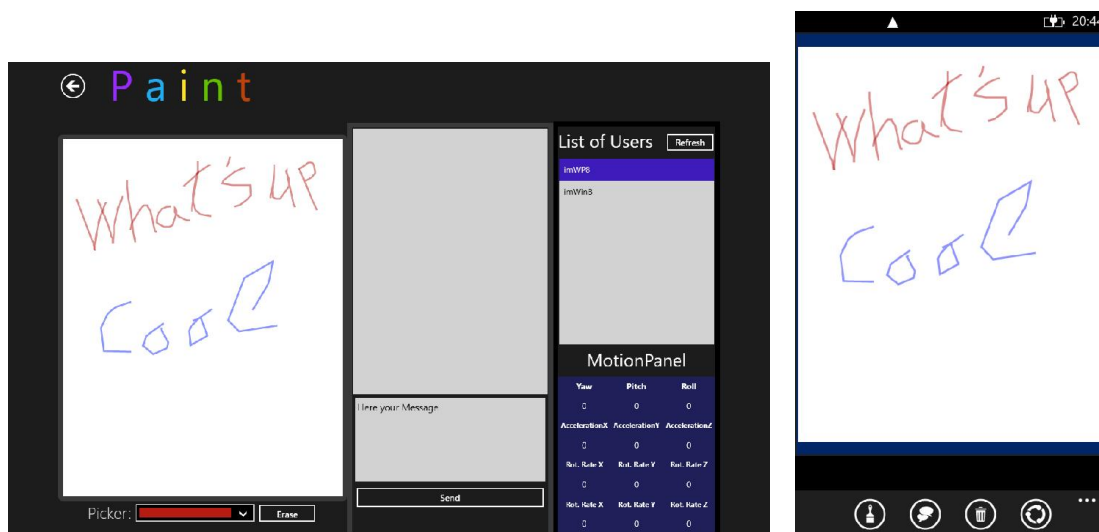

Figure 18 Sketching on client applications.

When the messenger is displayed, another icon with a canvas appears allowing the user come back to the canvas (see the canvas icon on Windows Phone 8 app in Figure 19).
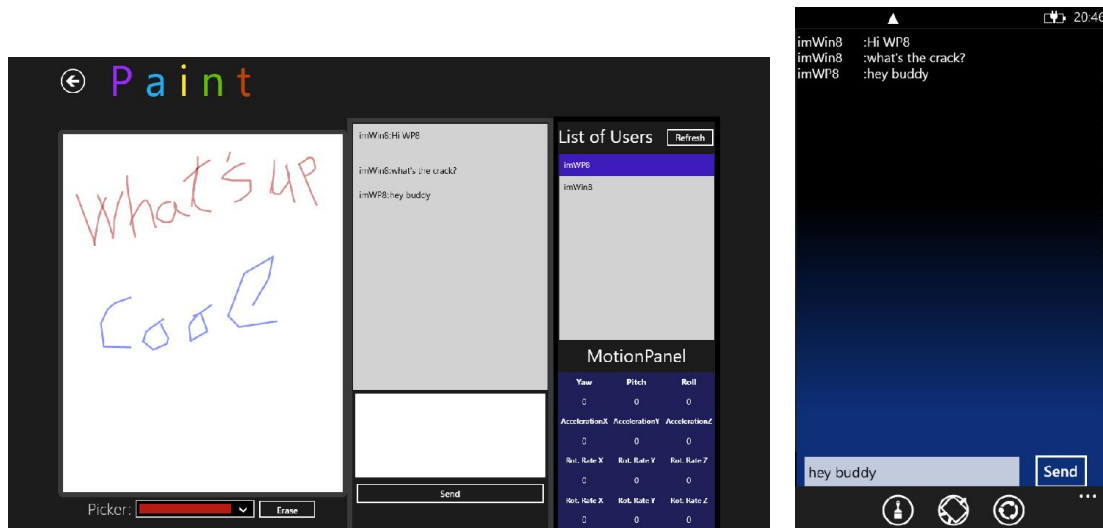
Figure 19 Chatting on client applications.

When Windows Phone 8 client is equipped with Motion sensors (i.e. gyroscope or accelerometer) an additional icon button will be available at the bottom of the user interface (see the cycle icon on Windows Phone 8 in Figure 19). This button enables Motion Seed function described on *Functional Service Specifications*. To disable this function, the same button but now with a cross icon must be tap.
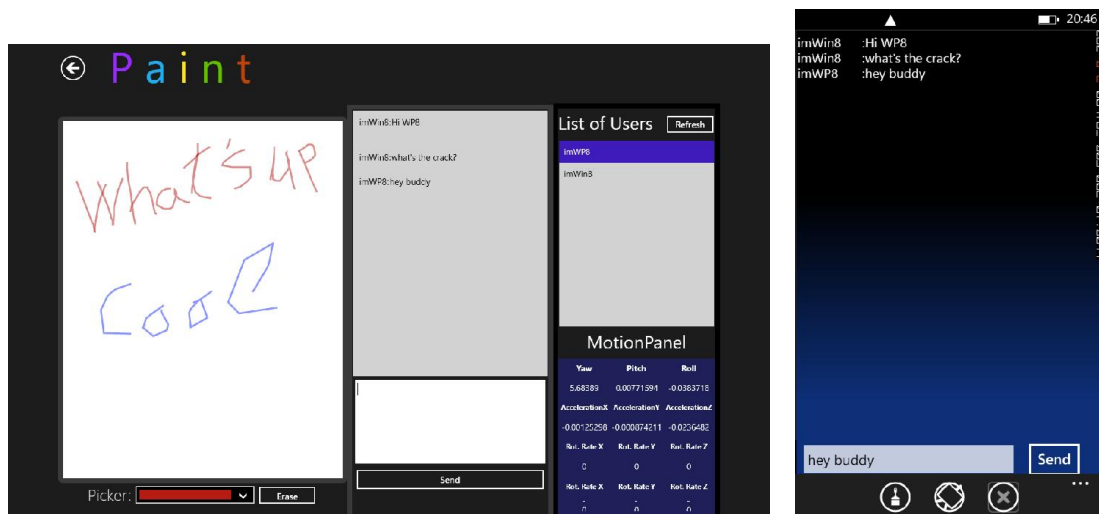

Figure 20 Sending motion information on client applications.

Finally the user can go back to the Home page using the back arrow button physically present on Windows Phone 8 (under the screen at left side) or the black arrow on the top left side of the screen on Windows Store App. This action implies to log out of the room.

4.4.   Software Architecture

All the code developed on this project is written in C# using .NET framework and some external references. See below some concepts that are required to fully understand the next section:

- Windows Presentation Foundation (WPF) is the graphical system for rendering user interfaces in Windows-based applications. WPF employs XAML-based language to define and link various UI (User Interface) elements (Wikipedia).

- XAML is a declarative makup language applied to .NET framework programming user interfaces. UI classes are composed by two files, one containing all UI elements (.XAML extension) and other one containing all the run-time logic (.cs extension and known as code-behind) (Microsoft Dev Network).

- Data binding is the process to connect (or bind) an element of the UI to data object allowing data to be synchronized between two entities. Then, the changes on the data object are automatically reflected on the UI element. Data bindings can be configured in *two way* mode and changes on the UI elements would modify the data object (Windows Dev centre).

- An assembly in .NET is the minimum unit of software deployment. Usually corresponds to a single file but it doesn't have to. Single-file assemblies are usually DLLs or .exe files.

4.4.1. Model-View-ViewModel (MVVM)

MVVM architectural pattern was developed by Microsoft as a specialization of the Model-View-Presenter (MVP) pattern. MVP was itself derived from Model-View-Controller (MVC) pattern. MVVM was created to leverage the advanced data binding features in WPF and Silverlight (and now Windows Store apps) and facilitate a strict separation-of-concerns between the (XAML-defined) View and the Model.

In Windows Store and Windows Phone MVVM-based apps:

- The Model encapsulates business/data logic.
- The View is defined using XAML.
- The ViewModel makes Model data available to the View and responds to changes in the View.
- The strict separation-of-concerns in MVVM means that a View can call the ViewModel, but not the Model. Similarly, the Model can't call the View directly (see Figure 21).
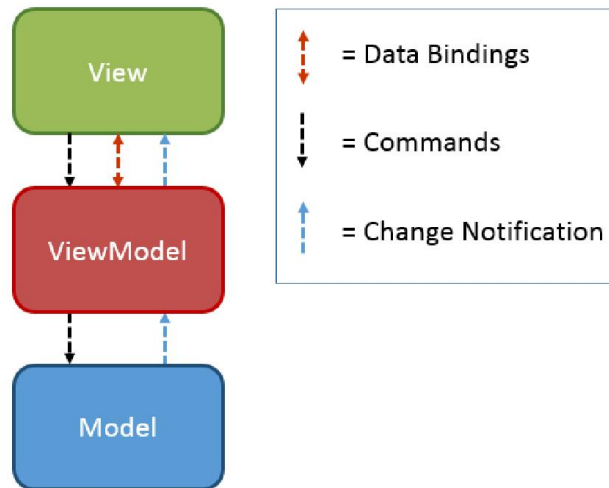
Figure 21 MVVM Software architecture.

The goal of using MVVM pattern is to minimize the amount of "glue" code needed to manage the flow of data between the View and ViewModel. Interactions between the View and ViewModel are achieved through data bindings specified in the View's XAML, and change notifications raised by the ViewModel.

```
// this is in Home.xaml
<GridView x:Name="RoomsGridView"
 ItemsSource="{Binding Rooms}"
>
```

```
// this is in HomeViewModel.cs
public class Rooms :
ObservableCollection<BaseRoom>;
```

Figure 22 Data binding example.

In Figure 22 it is possible to see the data binding for the list of rooms in Home page. On the left side, there is an user interface object *GridView* which is bonded to *Rooms* data object. This code is in Home page XAML file of both client implementations.

On the right side, there is the definition of *Rooms* class as inhering *ObservableCollection* of *BaseRoom. ObservableCollection* implements *InotifyPropertyChanged* that it is the requirement for a class to be biddable. This code is placed in the Home ViewModel file.

Then, any change on the class *Rooms* like adding an additional room, it is automatically propagated to the *GridView* and an additional item (representing a room) is added on it.

❖ *MVVM Light Toolkit*

For an easier application of MVVM pattern, MVVM Light Toolkit has been used. This provides several helper classes that accelerate the deployment and promote the reuse of code. Among these, the most used in this project are:

- ViewModelBase class is the common base for all ViewModels as it implements basic and common methods to apply MVVM pattern. For example implements *InotifyPropertyChanged.*

- Messenger class, used to communicate within the application using sender/subscribers pattern (receiver classes must register a listener method for each kind of message).

- EventToCommand behavior, allows to bind any event of any UI element to a Command (UI events are not biddable, only commands and some properties of the UI elements are biddable).

4.4.2. Portable Class Library (PCL)

The Portable Class Library (PCL) provides cross-platform development for applications using .NET Framework. PCL projects supports a subset of assemblies from the .NET Framework, Silverlight, .NET for Windows Store apps, Windows Phone, and Xbox 360, and generates a portable assembly that can be shared across apps for all these platforms.

The aim of PCL is to reuse all the application logic code embedding it in a portable assembly and at the same time, to reduce specific platform application to the View. As this project uses MVVM pattern, Models and ViewModels can be shared and located in PCL. Views classes and platform specific classes must be coded in their respective platform applications (see Figure 23).
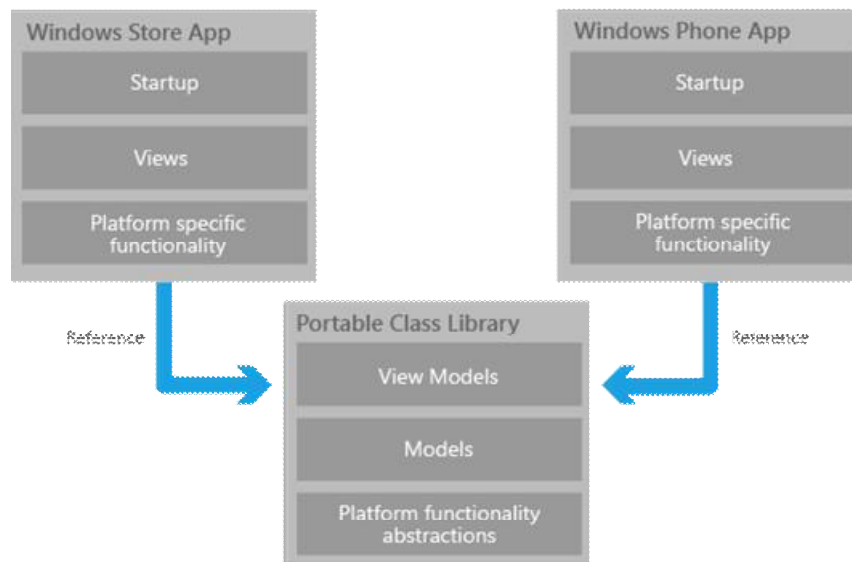

Figure 23 Classes location using PCL.

However, there are some limitations using portable class libraries:

- Only none XAML classes can be shared and in some cases part of these could be shared as well.

- PCL references must be other PCL assemblies and in some cases these packages are not available.

- The namespaces available are those ones that are common in all targeted frameworks (see Figure 24). That means, as more frameworks are targeted, smaller is the namespace available in PCL.
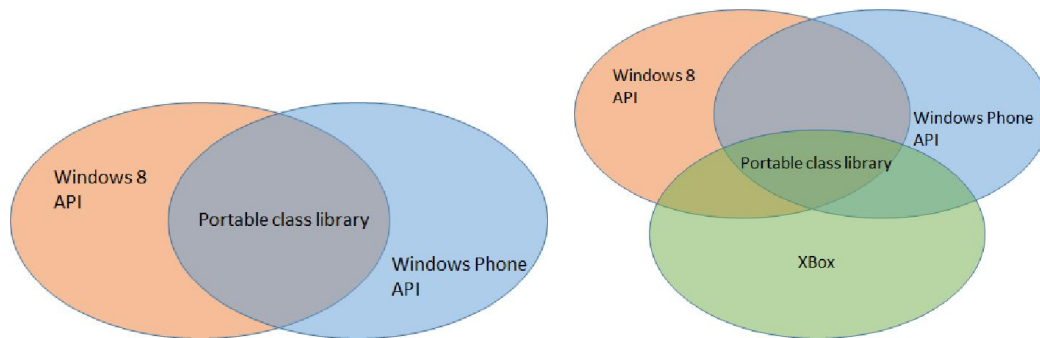

Figure 24 Two combinations of namespaces using PCL.

4.4.3. Service Developed

The service scenario is based on the client-server paradigm in which, both entities are fundamental as both provide the functionalities that combined enable the service.

❖ *Server*

For the service proposed, the server requires the set of features listed below:

1. *Hosting for the web service.*

   OWIN is the definition of a standard interface between .NET web server and web applications (this layer is known as middleware) and Katana is the implementation of OWIN for Microsoft servers and frameworks. The main advantage of using OWIN is that the web server and application are completely decoupled. This means that the application can run on any web server exposing OWIN interface.

   The service has been developed using OWIN-Katana, allowing to deploy the service on any web server or host supporting ASP.NET (main dependency of SignalR) and implementing OWIN interface. For example, Windows Azure, self-host as Windows application and so forth.

2. *Connection management.*

As the service is using the *Hub* API provided by SignalR, the connection management is performed for lower layers and only three methods triggered on connections events are exposed to the service code: *OnDisconnected(), OnConnected() and OnReconnected()* (See SignalR Hub class on Figure 25). Names of the methods are self-explanatory and indicate when each method is called. Using these methods, the server can be aware of which users are connected, in reconnecting state or disconnected preforming the correspondent task in each case.

3. *Rooms and users management.*

As users can create rooms, log in and log out. Server needs to track which users are logged in each room, room names (to avoid duplications) and number of room participants (delete rooms). All this management functionalities are provided by two classes: *RoomManager.cs* and *UserManager.cs.* Below listed some features of these classes:

- Add and delete users. Only user names not already present in the server can be added (as the user name is the key to track them).

- Add and delete room. Only room names not already present in the server can be added (as user name, the server name is the key).

- Connection ID and user name mapping. As each connection is associated to a connection ID (see SignalR connection token), it is not required to pass the user name as a parameter to identify the sender. Then, this mapping allows the server to track the user using the connection ID.

- Room name and user name mapping. Necessary to know which users are logged in for each room. Additionally, it is used to remove the rooms when during a certain amount of time, none user has been logged in (self-destructed by timeout).

4. *Providing authentication, authorization and encryption to the service.*

The authentication process implemented uses cookies to know if the user is log in or not and which account they are logged in with. A cookie is a small piece of data sent from a website and stored in an user's web browser or application while the user is accessing that website (Wikipedia). Detailed explanation is in 4.5.1 Authentication and Authorization using cookies and ASP.NET Identity module.

As seen before the authorization is built-in in SignalR (see 3.4) and the encryption is provided using SSL (see 4.5.2 Encryption using Secure Socket Layer (SSL)).

The diagram below shows the main classes on the server with their public properties (top box) and methods (bottom box).
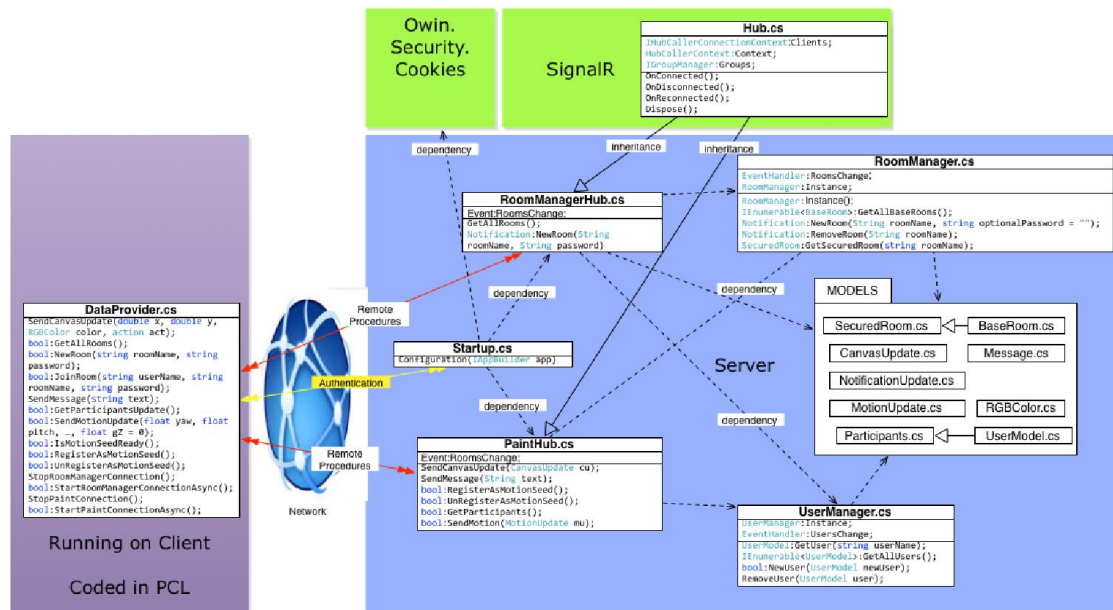

Figure 25 Server Code diagram.

In the Figure 25 is possible to see the main server classes already highlighted and the dependencies and inherence between them.

Notice that all the remote procedures take place between *DataProvider* class and both *Hubs* classes. On one hand, *RoomManagerHub* handles all the logic related to rooms at high level working with both manager classes. On the other hand, *PaintHub* handles all the logic related to painting, chatting and motion messages.

Models are required to de-serialize information received from the clients and send responses to them.

❖ *Clients*

On the other part of the connection, clients require the next list of features:

1. *Connection management.*

   As in the server, SignalR exposes exactly the same three methods for connection management. In this case, these methods have been used to notify to the user in case of disconnection or reconnection and control life-cycle of the client application.

2. *Navigation between pages.*

   A navigation class (*NativationService*) has been implemented for each targeted client to handle platform specific navigation functionalities.

3. *Capturing information from three different sources to send it to the server and receiving information from server and present it to the user interfaces.*

   This involves capturing from different sources: touch events, reading from buffer (for chat service) and motion events. Then all this information has to be modelled and serialized to be sent to the server. In the other way back, information is de-serialized and notified to the user interface which is refreshed with the new information. The classes involved in this process are:

   - *Home.xml* or *Paint.xml*. As explained before (see 4.1), client applications have two pages; *home* to select and log in to one room, and *paint* where main functionalities are available to the user (painting, chat, motion events) and for each page there is one ViewModel. View classes only handle the presentation of information and exceptionally capturing motion events or touching events as ViewModels are declared in PCL and none platform specific code can be coded within (see PCL limitations 4.4.2).

   - *HomeViewModel.cs* or *PaintViewModel.cs*. Both are the classes containing all the logic of the page handling user interface commands (click on a button or triggering some event such as close a popup).

   - *DataProvider.cs*. It encapsulates all the client-side procedures and functions calling to the server-side procedures (see Figure 25). This exposes several functions to send information to the server. In the opposite way (server to client), *DataProvider* uses an internal messaging tool (only within classes of the application) provided by MVVM Light toolkit (see 4.4.1) to notify the correspondent ViewModels classes about the new information received from the server.

   Serialization and deserialization tasks are carried out by SignalR client C# API.

Figure 26 contains the most important classes of the client's applications and indicates where such classes are coded. So it is possible to see that almost all the classes are coded in the portable class library (PCL) recycling a lot of lines of code. Moreover, it shows how the MVVM pattern has been applied across the client applications. Notice that all classes' names are really self-explanatory.
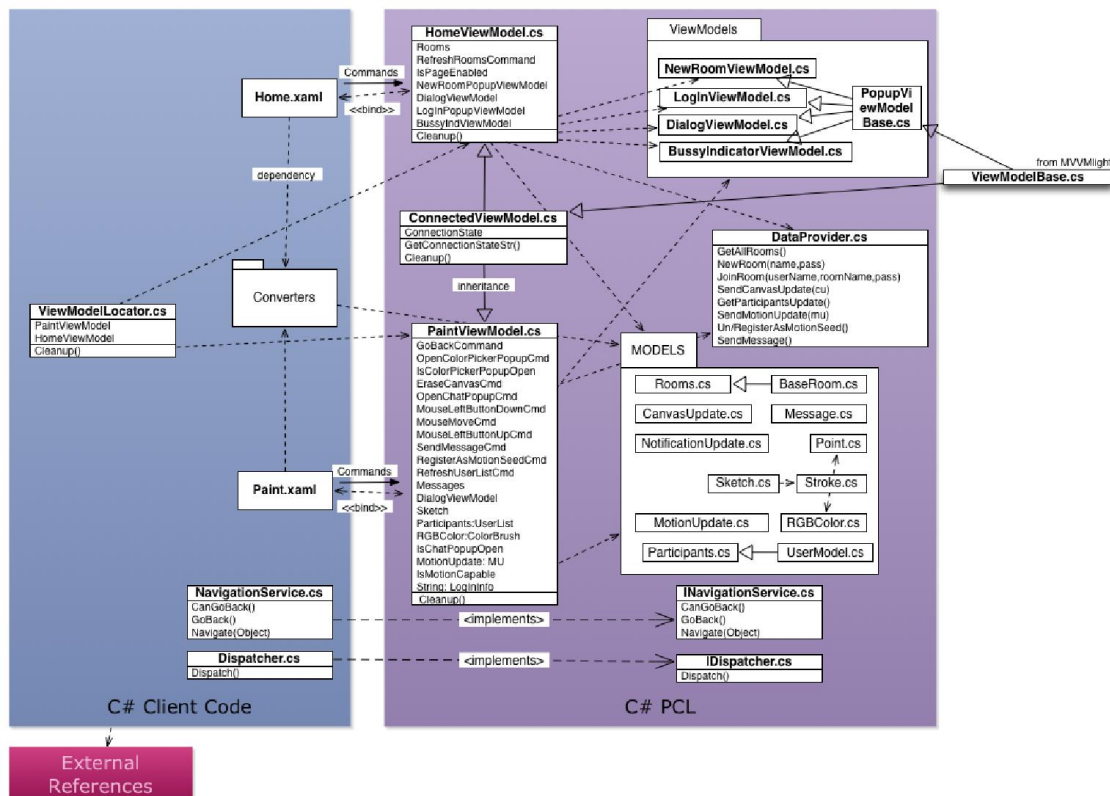
Figure 26 Diagram for the main classes in .Net client applications.

To resolve the conversion between Models classes and available platform classes a set of converters classes are required. These classes are coded on the platform specific assembly.

ViewModelLocator class is used to create and expose the ViewModels like singletons (unique instance of one class) as these usually live during all one application life-cycle. Moreover, *ViewModelLocator* uses *SimpleIoc* (enables dependency resolution at runtime instead of doing it at compilation time) for register and then retrieve the ViewModels (and other classes as well) and resolve any dependencies (parameters in its constructor) by looking at the interfaces that have been registered with it. The achievements using this are:

- Code Interface-drive. This means that in PCL it is possible to reference interfaces rather than concrete classes. Later on, at compilation time the platform specific class will be used.

- Code loosely coupled. The implementation of one interface can be changed but classes depending on that interface are not affected as are referencing to the interface.

- Resolve classes dependencies in an automatic way.

NativationService and *Dispatcher* are two examples of classes which need to be resolved using *ViewModelLocator* as both classes are platform dependent but they must be used on PCL. For that reason, both implements they own

interface on PCL assembly. The first class provides the navigation between pages and the second one exposes a method to dispatch new threads.

Notice that some Models classes could be shared between PCL and server code as they are the same classes (compare Figure 25 and Figure 26). But, actually the classes are slightly different (different dependencies too) as the server has some additional properties and methods that clients do not require and vice versa. Then, for the small amount of code of these classes was decided not to share them among server and PCL assemblies.

4.5.   Securing the service

As seen in the previous section, the service implements Authentication using cookies and Authorization provided by ASP.NET Identity. This section describes how the authentication procedure works and how the service has been ciphered.

4.5.1. Authentication and Authorization using cookies and ASP.NET Identity module

As seen before, the authentication process has been implemented using cookies which allow tracking the users during the connection session.

A cookie must be provided to the client application when the user has been successfully authenticated. To generate and map the cookies with the users, *owin.security.cookies* module has been used. This module is listening for a log in request (http://site.com/Account/Login) and when such request arrives, the module calls to an authentication method which checks if the credentials are correct. If these are correct, the module returns a cookie to the client application which will use it for all the next HTTP requests (to the server). Otherwise, a HTTP 401 (unauthorized) error is returned to the client application (see Figure 27).

Consecutively, if the user has logged in, server adds a *claim* (class which contains a piece of identity information such as a name or e-mail) in *Identity* module only for such user. This enables the user to use *Hub* methods protected with *[Authorize]* tag as SignalR relies authorization to ASP.NET *Identity* module (see 3.4).
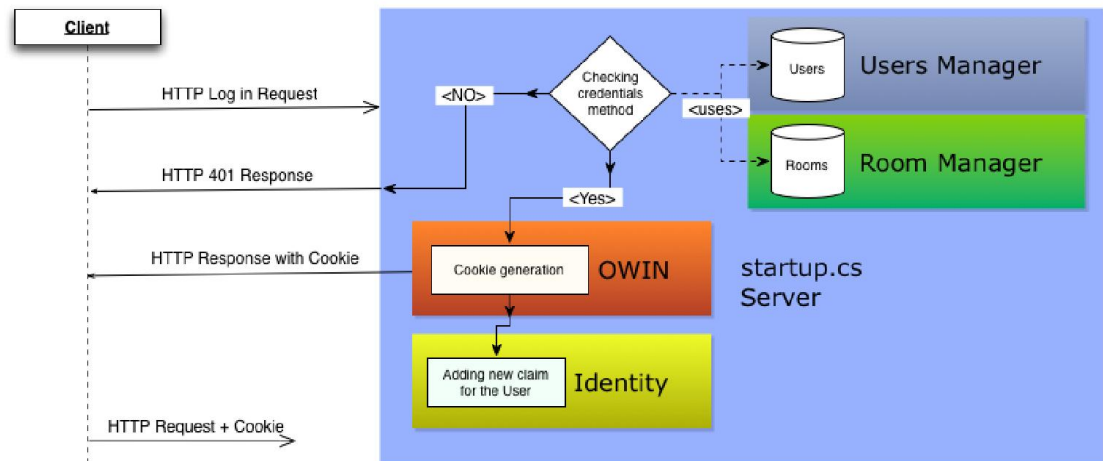
Figure 27 Authentication procedure.

Checking user credentials method only requires two steps. In first place, it ensures that there is no other user with the same username. Secondly, matches the password passed in the credentials with the one stored in memory for such room. If both steps are successful, the method grants the access to the user. Access is declined in the rest of the cases.

4.5.2. Encryption using Secure Socket Layer (SSL)

SignalR only encrypts the connection tokens and information on messages is sent as plain text. Therefore, a SSL layer has been introduced to encrypt the messages between client and server using self-signed certificates.

This has a double impact on the service performance as introduces a new layer of the service: additional handshake, additional task process at both ends (cyphering, un-cyphering) and more payload to the packets. Figure 28 shows the SSL handshake from one client connecting to the server deployed on Windows Azure services.

Figure 28 SSL handshake captured accessing to the service deployed on Windows Azure.

But in practice, the user experience impact is almost negligible. Among the reasons it is possible to highlight the small number of users concurrently connected to the server (two or three), the small size of the SSL headers, or the CPU resources available nowadays (multithreading and multi-core).

❖ *Limitations using SSL with self-signed certificates*

In typical Public Key Infrastructure (PKI), a particular public key certificate is signed by a certificate authority (CA) attesting that certificate is valid. But this has a considerable cost associated and the use of a self-signed certificate is a good alternative for development stages of one service.

In this project a self-signed certificated has been used and some problems have been faced in consequence.

The first issue is related to the domain name server (DNS) used to issue the certificate. A first solution is to use the server's hostname and then add it in *hosts* (local domain name resolution file) with the correspondent IP. This can be a solution for Windows 8 or any other device where hosts can be edited. But on Windows Phone 8 that file cannot be edited and a own public domain name is required (cost associated) if the service needs to be provided on line.

The second one is if the CA is not a trusted authority. Then, the application does not trust within the self-signed certificate. Usually if the service is accessed using a web browser, is possible to add an exception and trust with the CA used to sign the certificate (see Figure 29). But for .NET clients targeting Windows Phone 8 or Windows Store App, there is no such way to do it in the code. Then, the only solution is to install manually the public certificate on the device which rest a lot of the flexibility in the client application.
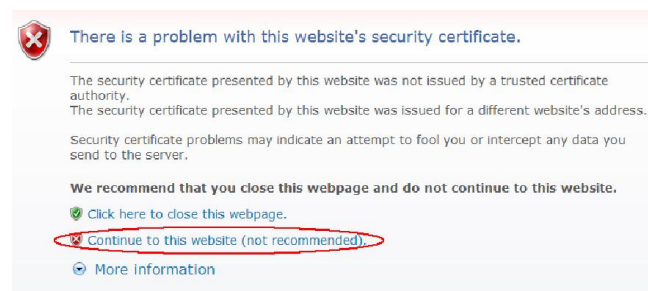
Figure 29 Browser advices for certificates issued by none-trusted CAs.

Then, in the development scenario (LAN) only Windows 8 client could get an SSL connection with the server using a self-signed certificate issued with the hostname of the local machine.

❖ *Using Windows Azure certificates*

As the service has been deployed on Windows Azure and this cloud service offers its own SSL certificate to all the subdomains in azurewebsites.net; It is possible to connect to the service using SSL (paint.azurewebservices.net) without any of the previously described problems (see Figure 28).

CHAPTER 5. MANAGING & TESTING THE SERVICE

5.1.   Devices used to test the service

The hardware used for developing and trying the service is the following:

a) A MacBook Pro 2.26 Dual Core 6 GB of ram running Windows 8 Professional using Parallels Desktop 8 (virtualization software).
b) A Nokia handset model 620 with Windows Phone 8 O.S.
c) An ALFA Network WLAN access point 802.11b/g/n USB adapter model AWUS036NH.

To see the developing scenario and see how this hardware has been used refer to Figure 12 Development service scenario.

5.2.   Source Code Management

To properly develop software, some code management is required to track the changes, updates and customs versions for special purposes (i.e. for testing). For this specific purpose Git has been used. Git is one distributed version control software (known as source or revision control as well) which provides a code repository with control on the code changes.

For more convenient development, a code hosting (BitBucket) compatible with Git has been used and additionally it is a backup of the code in case the development laptop could be affected for some issue.

5.3.   Deploying the service on Windows Azure

LAN environments are ideal for development and testing features but are too ideal scenarios where there are a few hazards to take into account. To evaluate the service in real conditions where the connection can go through different networks, firewalls, proxies or NATs requires deploying it in some server with Internet connectivity.

So, the service has been deployed on Windows Azure making the service accessible on Internet. Windows Azure is a cloud computing platform for deploying and managing applications and services through a global network managed by Microsoft.

Figure 30 Service scenario using Windows Azure.

This new scenario introduces additional factors that might impact on the service performance:

- Physical location of the server and consequently the RTT (Round-Trip Time) on the communication.

- Characteristics of the networks in which the communication goes through, like MTU (Maximum Transmission Unit), routing or load balancing policies and so on.

- And the status of these networks: congestion, losses and so forth.

- Hardware and OS resources allocated are unknown (because a free Windows Azure account doesn't specify the resources allocated for the service).

5.4.   Connectivity is the key for real-time experience

Both clients have been tested using different kinds of connections such as WLAN or 3G and any problem has been detected accessing to the service (getting connected) even using public WLAN behind firewall and NATs. But, it has been observed that the real-time experience is affected by latency and bandwidth of the connection.

Taking into account that it has been developed a real-time web application and all the communications take place using HTTP/TCP, the service is highly affected by (in order from more to less):

1. Losses in the connection forcing to a slow-start process by TCP congestion control. This is a mechanism to avoid send more data than the network is capable of transmitting and reduces the bandwidth

available drastically to increase in slowly basis to find the optimum value for it (Wikipedia). This mainly affects to Paint and Motion Events functionalities as they require a reasonable amount of bandwidth.

2. Latency. In order for the delay between user action and program output to be perceived as non-existent the latency must be low (ideally less than 30ms). Nowadays only cable and fibre connections can provide latencies close to this value, however most widely used xDSL connections provide 50+ milliseconds.

3. Handshakes are negotiation processes presents in some transport protocols at the beginning of the communication. HTTP, TCP and SSL protocols has their own handshake process and in case of reconnection due timeout all handshakes need to be redone.

4. IP fragmentation and load balancing techniques. If IP packets are fragmented and/or load balancing techniques are applied, it is highly possible receive the packet not in order. This implies a waiting time at receiver-side to complete the IP payload introducing extra delay to the latency and server processing time.

CHAPTER 6. CONCLUSIONS & FURTHER WORK

In this project, several frameworks real-time for web services have been evaluated. Among these, one has been used to develop a real-time service with several functionalities covering a range of user cases with real-time requirements.

During the development, recent techniques of code recycling and code architecture have been applied reducing the size of the application significantly and making it very flexible to build additional functionalities in. In percentage, the code reused between both clients (PCL code) hits 60%. It has been a challenging work to properly define the *Models* and *Converters* classes allowing allocate mostly all the code in the PCL. The reason is the low number of classes in common between both targeted clients because even a simple class like *Point* is not part of this common namespace (see 4.4.2).

Additionally one version control software has been used creating a backup repository and allowing to share the code with others developers or users.

Moreover, the service has been secured using three main requirements for information security: Authentication, Authorization and Encryption.

Then, the service has been deployed on cloud services making it available on Internet and testable everywhere.

Finally, some practical tests have been performed showing that the service functionalities provide a good interactivity to the user when the server is hosted in the same LAN. However, the response time increases notably affecting seriously to the user experience when the server is deployed in Azure. Latency and the other effects (described in 5.4) increase the response time until certain point in which the user has to wait some seconds to get an output on the user interface. These situations have been observed especially using Windows Phone 8 client as only has WLAN connectivity (more delay and likely to have losses).

❖ Improving the service keeping the high availability

The client-server paradigm is the best scenario for any kind of control communication like signalling, but usually is the worst scenario for user information if this is addressed to others users and there is no need to be processed on server-side (very usual in real-time applications).

Additionally, some users can be physically located close by and it has no sense establish a communication between them using a server physically located on another part of the world. Consequently, quality of the service can be degraded due to higher RTT or potential network hazards, as there are more networks involved in the communication.

Then, a better scenario where control and user communications follow different paths (client-server for control plane and user-user for user plane) would provide better performance and flexibility to the service. As a backwards, more resources at the ends (more sockets and CPU) would be required and the service development would get more complexity. Recent technologies are providing this kind of scenario for web services offering an attractive chance to be combined with SignalR and get the strengths of both technologies.

WebRTC is an open-source technology for Real Time Communications on web services. It is mainly focused on media applications but can be used with any kind of data. The main attractive is that is fully based on JavaScript enabling to be used in any browser without installing plugins (can be used in native application using C# API too).
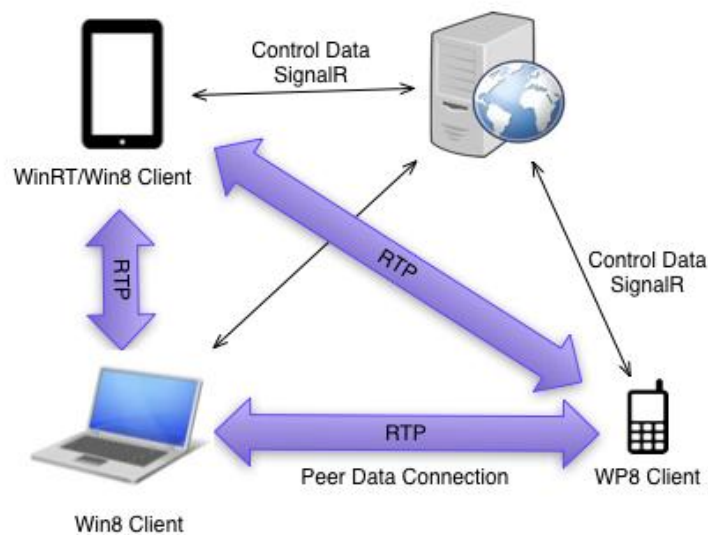

Figure 31 Improved service using WebRTC.

The scenario would be similar that the one represented on Figure 31 in which each client has a peer to peer data connection between them and an additional control Data connection for control messages of the application. But this scenario introduces a new challenge as peer to peer connections cannot go through firewalls or NATs. So, one possible option to mitigate this risk would be use SignalR technology as a fall-back in case the peer to peer connection could not be established.

Bibliography

[1] Dylan Schiemann, "The forever-frame technique", Comet Daily, http://cometdaily.com/2007/11/05/the-forever-frame-technique/ , (2007)

[2] "Ajax (programming)", Wikipedia, http://en.wikipedia.org/wiki/Ajax_%28programming%29

[3] "What are Server-Sent Events", Jersey 2.6 User Guide, https://jersey.java.net/documentation/latest/sse.html

[4] "About HTML5 WebSockets", WebSocket.org http://www.websocket.org/aboutwebsocket.html

[5] Matt West ,"An Introduction to WebSockets", treehouse blog http://blog.teamtreehouse.com/an-introduction-to-websockets

[6] Carsten Siemens,"SignalR", TechNet Microsoft US https://social.technet.microsoft.com/wiki/contents/articles/16984.signalr.aspx

[7] Brock Allen,"A primer on OWIN cookie authentication middleware for ASP.NET developer", brockallen http://brockallen.com/2013/10/24/a-primer-on-owin-cookie-authentication-middleware-for-the-asp-net-developer/

[8] Laurent Bugnion,"MVVM Light Toolkit", GalaSoft http://www.galasoft.ch/mvvm/

[9] "Share functionality using Portable Class Libraries", Windows Phone - Dev Center http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj714086%28v=vs.105%29.aspx (2013)

[10]   "Introduction to ASP.NET Identity", Microsoft ASP.NET http://www.asp.net/identity/overview/getting-started/introduction-to-aspnet-identity, (2013)

[11]   Laurent Bugnion, "IOC Containers and MVVM", MSDN Magazine, http://msdn.microsoft.com/en-us/magazine/jj991965.aspx

Resources Cited

- *ASP.NET SignalR forum*. (sense data). Recollit de http://forums.asp.net/1254.aspx
- *Bitbucket*. (sense data). Recollit de https://bitbucket.org
- *Microsoft Dev Network*. (sense data). Recollit de XAML Overview: http://msdn.microsoft.com/en-us/library/ms752059%28v=vs.110%29.aspx
- *Mono Project*. (sense data). Recollit de http://www.mono-project.com/Compatibility
- *PubNub*. (sense data). Recollit de http://www.pubnub.com/
- *Pusher*. (sense data). Recollit de http://pusher.com/
- Rouse, M. (March / 2008). *Search Unified Communications*. Recollit de http://searchunifiedcommunications.techtarget.com/definition/real-time-communications
- *Socket.io*. (sense data). Recollit de Wiki: https://github.com/learnboost/socket.io/wiki
- *SocketIO4NET*. (2 / Nov / 2013). Recollit de http://socketio4net.codeplex.com/
- *Wikipedia*. (sense data). Recollit de Windows_Presentation_Foundation: http://en.wikipedia.org/wiki/Windows_Presentation_Foundation
- *Wikipedia*. (n.d.). Retrieved from HTTP_cookie: http://en.wikipedia.org/wiki/HTTP_cookie
- *Wikipedia*. (sense data). Recollit de Remote_procedure_call: http://en.wikipedia.org/wiki/Remote_procedure_call
- *Windows Dev centre*. (sense data). Recollit de Data binding overview: http://msdn.microsoft.com/en-us/library/windows/apps/hh758320.aspx
- *Xsockets.net* . (sense data). Recollit de Support: http://xsockets.net/services/support

Acronym List

AJAX (**A**synchronous **J**avaScript **A**nd **X**ML), 9
API (**A**pplication **P**rogramming **I**nterface), 14
CA (**C**ertificate **A**uthority), 37
DHTML (Dynamic Hypertext Markup Language), 9
DNS (**D**omain **N**ame **S**erver), 37
HTML5 (HyperText Markup Language, version 5), 9
JSON (**J**ava**S**cript **O**bject **N**otation), 18
LAN (**L**ocal **A**rea **N**etwork), 39
mIRC (Internet Relay Chat), 9
MTU (**M**aximum **T**ransmission **U**nit), 40
MVP (**M**odel-**V**iew-**P**resente), 27
MVVM (**M**odel-**V**iew-**V**iew**M**odel), 27
NAT (*Network Address Translation*), 7
OS (**O**perating **S**ystem), 40
OWIN (**O**pen **W**eb **I**nterface for .**N**ET), 30
PCL (**P**ortable **C**lass **L**ibrary), 29
PKI (**P**ublic **K**ey **I**nfractructure), 37
RTP (Real Time Protocol), 9
RTT (**R**ound-**T**rip **T**ime), 40
SimpleIoc (**Simple I**nversion **o**f **C**ontrol), 34
SSE (**S**erver **s**ent **E**vents), 11
WebRTC (**Web R**eal-**T**ime **C**ommunications), 14
WLAN (**W**ireless **L**ocal **A**rea **N**etwork), 22
WPF (**W**indows **P**resentation **F**undation), 27
XAML (E**x**tensible **A**pplication **M**arkup **L**anguage), 27