RemoteUI
A Middleware for user interaction mobile applications

Miguel de Almeida Vasconcelos Raposo de Carvalho

Thesis to obtain the Master of Science Degree in
Telecommunications and Informatics Engineering

Supervisor: Prof. João Nuno de Oliveira e Silva

Examination Committee
Chairperson: Prof. Paulo Jorge Pires Ferreira
Supervisor: Prof. João Nuno de Oliveira e Silva
Member of the Committee: Prof. Renato Jorge Caleira Nunes

November 2015
Acknowledgments

I would like to express my sincere gratitude to Prof. João Nuno Silva for supervising this thesis, especially for his patience and motivation at all stages. I am also thankful to my friends for their encouragement, and for helping me through the difficult periods. Most importantly, nothing would have been possible without the love and support of my family, especially my mother, that always encourages and inspires me, to whom I dedicate this work.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013, and by the European Commission through the contract no. 610713 (PCAS).
Resumo

A computação móvel tem sido uma das mais desafiantes áreas de investigação ao longo dos anos, e recentemente tem-se assistido ao aparecimento de aplicações que proporcionam interacção entre dispositivos sem ecrã e dispositivos móveis. A proliferação de ecrãs finos de alta resolução e com grandes dimensões, e outros tipos de tecnologias, são outra das motivações para o desenvolvimento deste tipo de aplicações. Apesar da maior parte destes ecrãs serem utilizados apenas para promover diversos tipo de publicidade ou informação, os utilizadores têm-se mostrado cada vez mais interessados na possibilidade do seu uso como um recurso para as suas necessidades. Isto requer a implementação de interacções relativamente simples, como por exemplo enviar texto ou selecionar diferentes tipos de opções utilizando os seus smartphones. Baseado na literatura actual verifica-se a falta de recursos que auxiliem este processo, e a maior parte dos sistemas existentes são baseados em sistemas de comunicação através do browser. Este trabalho tem como objectivo disponibilizar um middleware que suporta o desenvolvimento de aplicações que implementem interacções com os utilizadores, proporcionando aos programadores mecanismos de abstracção. Permite também implementar aplicações desenvolvendo apenas código no cliente, em vez de toda uma arquitectura cliente-servidor. A validação deste trabalho é feita através do desenvolvimento de cenários de demonstração que permitem verificar os requisitos estabelecidos, e através de uma análise de performance ao nível da comunicação.

Palavras-chave: Middleware, Internet of Things, Ambient Intelligence, Aplicações móveis, Interacção com o utilizador
Abstract

Mobile computing has been a challenging area of study over the years and recently interaction applications between headless and mobile devices are starting to emerge. The proliferation of high-resolution flat screens with large sizes, and other technologies, have also motivated the development of these kinds of applications. Though most of the displays are used only for advertising and information, users are feeling more interested in the possibility of using them as a resource to their needs. This requires the use of simple interactions, like sending text or selecting between options through their smartphones. Based on the literature there is a lack of tools available to help in this process, and most are based on web browser communication. This work aims in providing a middleware that supports the development of applications with simple user interactions providing programmers abstraction mechanisms. It also allows implementing only a main application instead of whole client-server architecture. The validation of this work is done with the development of simple application scenarios that demonstrate that the system requirements are met, together with an analysis of the communication performance.

**Keywords:** Middleware, Mobile applications, Ambient Intelligence, Internet of Things, User Interaction
# Contents

Acknowledgments ................................................................. i
Resumo .................................................................................. ii
Abstract .............................................................................. ii
List of Tables .......................................................................... v
List of Figures ......................................................................... vii
Abbreviations and Acronyms .................................................... 1

1 Introduction .................................................................
   1.1 Motivation ................................................................. 2
       1.1.1 Ambient intelligence ............................................ 3
       1.1.2 Interacting with public displays ......................... 3
       1.1.3 IoT ................................................................. 4
       1.1.4 Case Scenarios ................................................ 5
   1.2 Problem statement ..................................................... 6
   1.3 Proposed Solution ..................................................... 7
   1.4 Document Structure ................................................ 7

2 Related Work .............................................................
   2.1 Mobile UI Systems .................................................... 9
   2.2 Remote Screen .......................................................... 13
   2.3 Discussion ............................................................... 14
       2.3.1 Mobile UI Systems ............................................ 14
       2.3.2 Remote Screen ................................................ 14

3 RemoteUI .................................................................
   3.1 Architecture ............................................................ 15
       3.1.1 Layer 1 .......................................................... 16
       3.1.2 Layer 2 .......................................................... 16
       3.1.3 Layer 3 .......................................................... 17
   3.2 Interaction Widgets .................................................. 17
   3.3 Programming API .................................................... 19
   3.4 Interaction models ..................................................... 23
List of Tables

3.1 Comparison between jQuery widgets and native OS widgets .......................... 18
3.2 RemoteUₐ proposed UI widgets ................................................................. 19
3.3 List of widgets according to tasks .................................................................. 19
3.4 RemoteUₐ API methods .................................................................................. 22

4.1 RemoteUₐ client library (C++) ...................................................................... 29
4.2 RemoteUₐ server library ................................................................................ 29

5.1 Payload on different types of messages ......................................................... 42
5.2 Network traffic comparison with VNC and X11 .......................................... 44
List of Figures

1.1 Evolution of Mobile Devices and Computer sales ........................................... 1
1.2 Using the smartphone to interact with: a) Wearable computing; b) Smart TV .......... 2
1.3 User interaction with a public display ................................................................. 4
1.4 Helping programmers in the development of interaction applications .................. 6
1.5 Behavior of the middleware interaction process ............................................... 7
2.1 Using mobile devices as UI to applications ....................................................... 9
2.2 MoCHA system architecture .............................................................................. 10
2.3 iCapture system architecture ............................................................................ 10
2.4 PuReWidgets system architecture ...................................................................... 11
2.5 Web-based toolkit system architecture .............................................................. 12
2.6 MAGIC Broker event flow ................................................................................. 12
2.7 DireWolf example of distribution of user interface components to multiple devices ... 13
3.1 System Architecture ......................................................................................... 16
3.2 C++ code example and corresponding mobile screen output ............................... 20
3.3 C++ code example and corresponding mobile screen .......................................... 21
3.4 Remote screen results processing pseudo-code: a) synchronous mode, b) asynchronous mode ................................................................. 22
3.5 Received results example ................................................................................... 22
3.6 Synchronous mode ............................................................................................. 24
3.7 Asynchronous mode ........................................................................................... 25
3.8 Update mode ....................................................................................................... 25
3.9 Code to update screen and corresponding result on screen .................................. 26
4.1 Components structure ....................................................................................... 27
4.2 Comparison between Protocol Buffers and other encoding languages ............... 33
5.1 Map application on the public display/Kiosk: a) Kiosk b) mobile phone .............. 39
5.2 Smart refrigerator UI on smartphone: a) Menu options b) barcode reader .......... 40
5.3 Code to generate UI to control lighting system .................................................. 40
5.4 Lighting system UI .............................................................................................. 41
5.5 Example used in 2 widgets tests: a) min; b) max ................................. 41
5.6 RemoteUi comparison test: a) no scroll; b) scroll required .................. 43
5.7 PureWidgets code for an Hello World Application .............................. 45
5.8 WebView attacks ............................................................................. 47
Abbreviations and Acronyms

API  Application Programming Interface
App  Application
GUI  Graphical User Interface
HTML HyperText Markup Language
IoT  Internet of Things
JSON JavaScript Object Notation
OS   Operating System
PC   Personal Computer
REST Representational State Transfer
RPC  Remote Procedure Calls
UI   User Interface
USB  Universal Serial Bus
XML  Extensible Markup Language
Chapter 1

Introduction

In the past few years there has been a popularization of personal and mobile devices with powerful computing capabilities that started to overcome the use of the traditional personal computers. Nowadays users don’t have to use these kind of computers to perform simple tasks, such as emailing or web browsing, as they can do it any time and anywhere on their smartphone or tablet [26, 27]. Figure 1.1 presents the evolution of the internet devices sales, showing a small growth in the PC sales against an exponential growth of the mobile devices.

![Global Internet Device Sales](image)

Figure 1.1: Evolution of Mobile Devices and Computer sales

Mobile devices also brought the possibility of interacting with other technologies, making it a perfect way to control or access them remotely [27, 29]. Studies have also shown that people are more and more interested in the use of mobile devices as a replacement for the traditional remote controls, mouse and keyboard input mechanisms [27, 34]. Figure 1.2.b) presents a good example, the Smart TV¹, that became popular by allowing users to access web contents and apps, and to use the smartphone or tablet in order to control it. More recently other kind of devices are emerging, such as mobile devices

¹[https://pt.wikipedia.org/wiki/Smart_TV](https://pt.wikipedia.org/wiki/Smart_TV)
accessories that expand their capabilities. Examples of these are the wristbands that can be used as fitness trackers, as shown in Figure 1.2.a).

Also Internet of Things (IoT) has motivated the research community in investigating solutions that allow to interconnect all the things that surround us. Until now the main objective of IoT was to interconnect and retrieve data from all kinds of sensors, and process it on the cloud (big data), but recently there have been raised new challenges into finding solutions that allow devices to interact directly with users.

All of these have motivated developers to design applications to improve the integration and communication between mobile devices and numerous others.

Ambient intelligence is another area where this type of applications are relevant, since one of the main goals is that the technology disappears and only the user interface is perceived by users.

Developing applications that interact with users is demanding, even when developing centralized applications (display and logic in the same device). In a distributed environment, programmers may need to develop a whole server-client architecture application, and it is from greater importance to provide them with tools that can ease this process. On consumer electronics, examples of these devices are Chromecast [3] from Google and the smartwatches, devices that interact with or are controlled by mobile phones, but require the development of both client and server applications. Other examples requiring significant programming effort include applications for collaborative purposes [32], and real-time distribution user interface systems [19, 39, 22].

![Figure 1.2: Using the smartphone to interact with: a) Wearable computing; b) Smart TV](image)

1.1 Motivation

Though the growing interest of the research community in areas like IoT or public displays, there is still a lack of tools dedicated to the development of applications that can allow clear and efficient user interaction, between users and different kinds of devices.

The existing and proposed solutions regarding the Internet of Things are based in cloud computing, that processes lots of information that is uploaded from all kinds of sensors and devices. Now, with the emergence of small hardware with good processing capabilities there is a need for decentralized solutions, that can allow local processing and communication between these devices and users.
Using public displays as a resource to users is another important factor for the development of new kinds of solutions. Several approaches exist but most of them are web or cloud based systems that don’t allow an implementation in a closed environment and require complex deployment or infrastructure.

### 1.1.1 Ambient intelligence

One of the possible areas of research in this domain is home automation/domotics, where there are already several commercial systems developed, that help people monitor and control the mechanical, electrical and electronic systems. These systems allow remotely controlling things like lighting, heating, ventilation, air conditioning, entertainment and home security in a more convenient and comfortable way, also improving energy efficiency and security.

This may be done through the use of proprietary controllers or with personal and mobile devices. Developing applications to run in mobile devices require the implementation of the necessary interaction mechanisms to help configure and control all the interconnected systems. This can be demanding in terms of programming the necessary user interactions since different devices may have different kinds of configurations. Providing efficient interaction abstractions can relief the programmer from a lot of unnecessary work, as each device could manage their necessary interactions resorting to the use of widgets, and the applications would only need to route them to the appropriate devices.

### 1.1.2 Interacting with public displays

One of the major challenges today is the integration of mobile and fixed devices. Large-area displays became so inexpensive these days that they are present in most of the public areas, and there will be a tendency for them to become interconnected. Researchers are beginning to explore the creation of new networks of pervasive displays that go beyond the traditional broadcast model and support innovative new applications and highly personalized content, tailored to nearby viewers [29, 36, 27, 38, 20]. Some studies show that in the future there will be public displays AppStores [21].

It is proved that there are many factors in the impact these displays can have in the interaction with spaces and people [24, 30, 42]. One of the main concerns is related to privacy issues. Though most of the works developed are targeted to hand gesture and touch screen controls [41], the use of personal devices such as smartphones, can be an efficient way of overcoming these issues [11]. One of the main factors that people take into account when interacting with public displays is that they don’t want their personal information to be available to observers. A solution to this problem is the use of their personal mobile devices to personalize and configure what type of information to be shown in public places.

Another advantage of the use of these devices is that they can be used as input mechanisms, since a person can write faster and more accurately than in a big screen. More important, this leaves the public display free to act as an output display only, allowing sharing the screen with multiple users, either individually or collaboratively. It also reduces the costs allowing to use screens without touch screen technology, which are about three times more expensive than a basic screen.

There are different types of interaction phases regarding public displays. Vogel et al. [29] define four
different categories: Personal interaction; Subtle interaction; Implicit interaction; Ambient display. Our work is targeted to the first two. The subtle phase presupposes an implicit cue that the user wants to interact with the screen, like standing in front of the screen, and the personal phase is defined when the user selects any kind of option and may include dealing with user personal information, if available. Figure 1.3 shows an example of a user in a personal interaction with a public display. This could be done using the mobile phone instead of having to use the display touch screen.

![User interaction with a public display](image)

Figure 1.3: User interaction with a public display

1.1.3 IoT

Internet of Things is one of the most featured areas of research today. With the availability of small devices with significant processing power, the management, communication and programming efforts face new requirements, increasing the need for efficient interaction middlewares.

Some of these devices are smartphone accessories that can connect via USB, but the most common communication technologies used are wireless, like for example Wi-Fi, Bluetooth or NFC. The Bluetooth Low Energy (BLE) or Bluetooth Smart provides reduced power consumption and cost while compared to the Bluetooth classic, and it was developed targeting the smart home, health, sport and fitness sectors [2, 18]. All the most recent smartphones are equipped with this technology.

Wearable computing is another featured topic, as devices such as smartwatches or wrist bands are gaining popularity. The advances on lightweight, small-size, and ultra-low-power monitoring sensors that can be embedded in clothes, shoes, and others, allow the creation of body area network (BANs) [6, 18]. These networks can have several applications, such as in health care systems, sports, or security. Some of those devices take advantage of the new BLE to be able to connect to the smartphone, so it may be possible to interact with the user [1].

All of the mentioned devices require user interaction, and this is usually done using the user smartphone. RemoteU is can be a useful tool in the development of these kinds of applications.
1.1.4 Case Scenarios

Scenario 1

John and Mary are planning to buy a new TV for their living room and they heard of a new shopping center that has opened only a few kilometres from home. As they arrived there they started looking for some mapping indications so they could find the stores with electronics. Soon they realized that there were displays all over the center for that purpose. John approached one of the displays and started downloading the smartphone app, so they wouldn’t need to use the display as a touch screen, since he doesn’t like to type on large screens. This way he uses the phone for querying the application about the kind of stores he needs, allowing the display to show the locations. Since there is more than one store, he downloads a map file with the locations so he doesn’t need to go to another screen if he forgets one.

Scenario 2

Peter wants to print some photos that he has on his smartphone so he went to a kiosk near his working place. As it was lunchtime there was a waiting line to the interaction screens, but he noticed a poster with a QR code to download an interactive application instead. As he was almost late to return to work he installed the app to give it a try. He soon realized that is a simple process, and from one interaction screen to another he easily chose the photos he pretended to send for printing. This way he went directly to the checkout machines, so he could pay and get the photos, and still arrive to work in time.

Scenario 3

Sam has just bought a new monitoring bracelet for his exercising activities (similar to Figure 1.2). As he arrived home he didn’t want to waste time turning up the computer to make the initial configuration, even because he was already late to meet his friends for their bike ride. Meanwhile since he was excited to test the new gear he remembered that the employee who advised him to purchase that model told him that he could use his smartphone to do the configuration. Thus he read a QR code from the bracelet box and installed the application, and went to the meeting point, hoping to do the configuration while waiting for some friends to arrive, and this way he wouldn’t be the last arrival.
1.2 Problem statement

At the present reality there are some solutions that can help programmers developing applications with distributed user interfaces, though most of them show some level of complexity in terms of deployment, requiring the existence of external infrastructures (cloud based solutions), and leading to unwanted costs of maintenance.

Let's consider the generalised illustrated scenario presented in Figure 1.4, where we have a client device (headless, public display, etc.) that needs to interact with a user mobile device (smartphone, tablet, etc.). At present there aren’t available tools that can provide abstraction mechanisms between them using point-to-point communication.

Thus we think it is important to provide programmers with a middleware for the design of these kinds of applications, that allows an abstraction from the interaction implementation, and that allow communication between devices in a closed environment. The middleware consists in a programming API that allows the development of the client application, relieving the programmer from having to code all the necessary interaction screens in a mobile application. Figure 1.5 illustrates the general concept of the proposed system.

There are some considerations to take into account in order to obtain a proper solution. Thus the remote interaction device should have some specific characteristics:

- The device is personal and mobile, such as a smartphone or tablet;
- It is equipped with communication interfaces (Bluetooth, Wi-Fi, USB, etc.);
- The device may not have internet access.
1.3 Proposed Solution

This document presents RemoteU¡, a middleware that allows the implementation of complex interaction between devices (with limited resources and no screen) and users, resorting to local connectivity and mobile devices (such as smartphones or tablets). RemoteU¡ also relieves the programmer from the communication and interaction programming burden.

RemoteU¡ is a novel middleware running on mobile devices (where user interaction happens) and on scattered devices whose logic and code need input from, or provide output to users. Since the data encoding is optimized for speed, the devices can be connected by any network link (ZigBee, Bluetooth, WiFi, or USB), independently of its bandwidth.

The proposed middleware is generic, allowing the reuse of the infrastructure in different applications, providing the most common interactions (with respect to input/output data and usage patterns) and a simple, expressive and efficient programming API.

When compared to other systems RemoteU¡ provides a simpler implementation, allowing for the easy development of applications, or the extension in already developed systems without requiring additional resources. It also provides great communication performance with low communication overhead.

RemoteU¡ middleware was published and presented as poster at the Mobicom 2015 - The 21st Annual International Conference on Mobile Computing and Networking, in Paris, France in September 2015 [17].

1.4 Document Structure

This documents is divided into six chapters. The next chapter presents systems related to the objectives of RemoteU¡, describing how today it is possible to forward and relay user interaction into remote or
mobile devices. Chapter 3 presents RemoteUI system, describing its architecture and the interaction models, as well as the set of widgets that can be implemented resorting to the programming API. The following chapter presents the system implementation, detailing the client and server libraries, as well as the communication protocol. Chapter 5 presents the system evaluation regarding the performance, the simplicity of usage of the API, as well as some costs and security analysis. This document ends presenting the conclusion and future work.
Chapter 2

Related Work

In current mobile environments the presentation of information on remote devices resort to remote screen protocols, distributed UI infrastructures or to web based architectures. These systems allow the interaction between the user and the scattered devices, but require the existence of external infrastructures, complex programming or demand high computational resources. In order to reduce the programming of explicit client/server interaction between the devices and the screens, several approaches exist.

![Figure 2.1: Using mobile devices as UI to applications](image)

**2.1 Mobile UI Systems**

Several mobile UI systems (such as MoCHA [35]) are web-based systems and allow rich interactions between pervasive displays and users with mobile devices, in which both interact through a web page available at a certain address. MoCHA is divided in two mainly components, a distributed architecture based on modern web technologies and cloud computing to provide rich interactions between mobile devices and pervasive screens, and a mechanism for practical and secure binding based on their proximity. The proposed system architecture (Figure 2.2) requires the existence of an application server for the devices to communicate. In the presented prototype they use Amazon cloud services (EC2) to deploy the server, which requires the existence of an internet connection. The binding service was implemented using a time-varying QR code mechanism, and a token-based authentication method in
order to provide security. Though they don’t provide any API for third party programmers.

The *iCapture* [33] project is a web application designed to facilitate spontaneous user-interaction with situated display environments through the use of camera enabled mobile phones. The prototype application (Figure 2.3) consists in a large display (connected to a web server) that is responsible for exhibiting news headlines with a corresponding code, which users may capture to read it on the phone. Although it is an interesting example scenario the interactions are rather simple, and the system is limited to public displays interactions, not being able to implement with headless devices. It also doesn’t provide any API.

*PuReWidgets* [15] is a programming toolkit for developing interactive public display applications, composed by a widget library and web service that handles interaction events, providing programmers with high-level interaction abstractions. They provide two separate libraries:

- the server-side library (Figure 2.4, top) allows to create applications whose main logic resides on the web server and can run independently of the public display scheduling;

- The client-side library (Figure 2.4, bottom) allows to create applications that are more targeted to the public display interaction.
It was developed using Google's App Engine platform (http://code.google.com/appengine) and Google's Web Toolkit (http://code.google.com/webtoolkit), so it also dependent of a cloud infrastructure.

A Web-based toolkit for remote direct manipulation interaction with public displays via smartphones [14] is being developed to help programmers in the creation of applications with less effort. As shown in Figure 2.5 it is composed of an interaction server and a programming library that provides interaction controls (widgets) and interaction events. The toolkit provides controls such as joystick, text input, and multi-touch cursor events. The work is still in development (dated in 2014), and it is another browser-based architecture.
MAGIC Broker [23] is a middleware toolkit that provides a set of abstractions and a simple RESTful web service protocol to easily program interactive public display applications targeted to mobile device interactions. They propose an event base system, where users can interact by voice calling a Voice XML gateway, by sending SMS messages, or using a mobile web browser. Though they mention an API they don’t provide any details on its implementation or usability, and it seems that is only used for evaluation, not being available for third party developers.

Figure 2.5: Web-based toolkit system architecture

Figure 2.6: MAGIC Broker event flow

The DireWolf [26] is a framework for distributed web applications based on widgets. As presented
on Figure 2.7 it helps managing a set of devices providing easy distribution of widgets among them, and guarantees the preservation of the state. This work is more targeted for designing collaborative and multi-user applications.

Figure 2.7: DireWolf example of distribution of user interface components to multiple devices

2.2 Remote Screen

The concept of remote screen isn’t new in computation as many systems based on this idea started being developed long time ago. Nowadays the more prevalent systems are X11 [13] (implemented in most Linux Desktop distributions) and VNC [40] like systems.

X Window System (X11) [13] is an interesting system also implementing a client-server approach. It is one of the pioneer distributed user interfaces, and was originated in the year of 1984. X11 uses a client-server model where the users terminal is the server (where UI is rendered) and the applications are the clients, and provides a framework for the development and execution of GUI applications environment. Although it doesn’t implement the system user interface, each application is responsible for the UI definition, appearance and behaviour. Although X hides the distribution of UI’s into remote displays, the programmer should take into account the events and program all the necessary handlers, requiring high programming skills and resulting in high complexity programs.

Virtual Network Computing (VNC) [40] is an application originated in the late nineties that allows sharing a screen remotely. It is particularly useful for remote technical support and accessing files on remote computers, like accessing a work computer from home. One of the most interesting features of VNC is that it is platform-independent and the code is open source, thus has generated a vast number of implementations. Although platform independent, during its operation, the complete screen (encoded
as a bitmap) is sent to the client computer, thus incurring in high communication overhead. Furthermore
the presented UI is independent of the client UI framework, offering different user experience. Currently
other remote screen systems exist, such as TeamViewer or Microsoft Remote Display.

2.3 Discussion

All existing solutions, in general, require high computation resources, such as in terms of network con-
nectivity, such as regarding the devices processing power, thus limiting its wide deployment and usability.

2.3.1 Mobile UI Systems

Most of the mobile UI systems are implemented in a browser-based architecture, and in most cases
require the existence of an external infrastructure since many are deployed on cloud services. This
has several drawbacks, one of them being the costs of maintaining these cloud systems, and also the
Internet links. Even when an internet link is already available at site, it has always problems regarding
to latency, so it isn’t as fast as a local link.

It also means that they aren’t able to operate in a closed environment, like for example with phone
accessories that aren’t equipped with Wi-Fi. Although the present reality of Internet of Things rely mostly
on cloud systems that store and manage the acquired data (Big data management), the future of IoT is
pointing to devices that have some processing capabilities and can interact directly with users.

Moreover the majority of the works presented are prototype architectures that don’t provide any API
for third-party developers, and probably will only represent case studies.

2.3.2 Remote Screen

Regarding to X11 and VNC, both of these systems present many disadvantages when implementing
in small devices, the most relevant being the fact that, as they aren’t designed for this purpose, they
consume too many resources. Also they add too much network overhead when transmitting the neces-
sary information to generate interactions. The fact that they don’t provide the same UI to the user when
dealing with different applications is another negative aspect.
Chapter 3

RemoteUI

This chapter presents the proposed system architecture, along with the details of the possible interactions, regarding the widgets and communication models. It also presents the available programming API.

The first to do, in order to have a proper solution, is define the set of requirements. Thus, one important goal of RemoteUI is to be a generic and usable middleware in multiple situations, environments and applications, in order to avoid the duplication of implementations and effort. This includes, for example, that it could be used to extend already developed applications in different systems and infrastructures. Another objective is that it should provide the user with the presentation of the usual interactions, matching the appearance of what most users are used to deal, independently of the device used. It is also required to hide the client-server architecture from the programmer, making it possible to develop the application without worrying with the underlying methods that deal with the client-server interaction. The middleware should also be interoperable in various aspects, such as with the device where the application will run and also with the mobile devices. Finally it should be efficient, especially regarding the network and processing performance.

The requirements can be summarized as follows:

i Ease of programming

ii Interoperability

iii Low overhead (network and processing power)

iv Openness

v UI consistency

3.1 Architecture

Figure 3.1 depicts the proposed solution architecture. The left side represents the main system device where the client application (developed by the programmer) runs, and the right side is the mobile device,
which acts as the user interface server. Though the main device is characterized as being headless, it may still have an accessible user interface, such as a screen or speakers. This simple characterization emphasizes the fact that the middleware is designed to a wide variety of equipments and usable in the several domains. Both the headless and mobile devices are represented and divided by layers and a description of each of them is now presented.

**Figure 3.1: System Architecture**

### 3.1.1 Layer 1

The first layer represents the network. RemoteU¡ is communication agnostic so it can operate over any kind of connection, such as Wi-Fi or Bluetooth, or even a wired connection such as USB. It only requires that a communication socket is previously established between devices, so that the interactions may take place. Thus RemoteU¡ can be integrated with multiple discovery mechanisms, though how devices recognize each other and establish the communication process is out of the scope of this work. The programmer is responsible to implement these features, and there are several solutions that can be adopted. This allows RemoteU¡ to be implemented in various network configurations. Resources and service discovery mechanisms are also independent of our solution.

### 3.1.2 Layer 2

The second layer is the core of the middleware, and it is present in both devices. The system is responsible for providing interaction mechanisms between them and this is made through the generation and management of defined messages. Protocol Buffers [9] are used to serialize the messages exchanged between the client and server. When compared with other widely available alternatives (such as XML or JSON), Protocol Buffers are more efficient when serializing data [5] (with respect to processing time and serialization results size) and is also widely available.
3.1.3 Layer 3

The programming API represents the third level of the architecture, as it is one of the distinctive features of the RemoteU$_i$ system. The application logic is programmed with the regular language structures, and the user interaction is done through the API. It provides a set of methods that can be invoked in the application that allows to automatically generate the widgets that are displayed in the user mobile device. At present it is available in Java and C++ languages.

In the next subsections all of these features are detailed, regarding the interaction widgets and models, the API methods, the implementation of the middleware as well as the communication process.

3.2 Interaction Widgets

Widgets are one of the most useful resources programmers have nowadays for implementing interaction features in their applications, as they represent interface components with clear functionality dedicated to small tasks. Widgets are reusable for multiple purposes in different applications so their use in the design of mobile applications is very common. It reduces the overhead in the development and also provides users with a common language for interacting with the applications.

There are two possible ways of generating widgets in mobile devices. One is using the operating systems native support, which will vary between different versions of the operating systems, and may be more difficult to implement. The other option is through the use of high-level libraries, such as web JavaScript.

There are a vast number of libraries available for free but the most popular for web applications is jQuery, a JavaScript framework that is supported by the largest companies such as Google or Microsoft. As our solution is targeted to mobile devices there is a more suitable, the jQuery Mobile [7], which is specially designed for the development of these kinds of applications. It is an HTML5-based user interface system designed to make responsive web sites and apps, accessible from any device such as desktop computers, tablets and smartphones. jQuery Mobile offers wide compatibility and availability and allowing the presentation using a consistent design and layout. It is particularly useful to automatically adjust the widgets to different screen sizes. The use of this high-level library provides independence of the system where the UI is presented, thus avoiding problems regarded with proprietary OS widgets, which may have different kinds of configurations.

The text field and image presentation are not provided by jQuery library, thus are implemented in basic HTML. The Bar/QR Code reader is implemented with ZXing [8], which is an open-source, multi-format 1D/2D barcode image processing library, that allows integrating with android applications. All other widgets resort to the available on the jQuery Mobile library.

Next we present a list of the most useful widgets that the library provides:

- Button
- Checkbox
• Radio button
• Swipe list items: remove list items by swiping left or right
• Flip switch: On/Off
• Icons: built-in icons in jQuery Mobile can be applied to various widgets
• List view
• Loader
• Popup: Tips and photo views
• Select menu
• Text Input

In order to make a pre-selection of the most adequate set of widgets to implement in this middleware an analysis of the native OS widgets was made and compared with the list of widgets provided by the jQuery Mobile library. In this comparison there were taken into account the main operating systems: Android, iOS and Windows Phone. In Table 3.1, the first column represents the list of widgets from jQuery Mobile, and in the next three are the corresponding native widgets, if exist.

<table>
<thead>
<tr>
<th>jQuery Mobile</th>
<th>Android</th>
<th>iOS</th>
<th>Windows Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Button / ImageButton</td>
<td>UIButton</td>
<td>Button</td>
</tr>
<tr>
<td>Check Box</td>
<td>Check Box</td>
<td>-</td>
<td>Check Box</td>
</tr>
<tr>
<td>Radio Button</td>
<td>Radio Button</td>
<td>-</td>
<td>Radio Button</td>
</tr>
<tr>
<td>Date Picker</td>
<td>DialogFragment</td>
<td>UIDatePicker</td>
<td>-</td>
</tr>
<tr>
<td>Flip Switch</td>
<td>Switch</td>
<td>UISwitch</td>
<td>-</td>
</tr>
<tr>
<td>Icons</td>
<td>Action Bar Icon Pack</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>List View</td>
<td>ListView</td>
<td>UITableView</td>
<td>ItemCollection</td>
</tr>
<tr>
<td>Loader</td>
<td>ProgressBar</td>
<td>UIProgressView</td>
<td>ProgressBar</td>
</tr>
<tr>
<td>Popup</td>
<td>PopupWindow</td>
<td>UIView</td>
<td>ToolTip</td>
</tr>
<tr>
<td>Range Slider</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Slider</td>
<td>SeekBar</td>
<td>UISlider</td>
<td>Slider</td>
</tr>
<tr>
<td>Select Menu</td>
<td>Spinner</td>
<td>-</td>
<td>ListBox</td>
</tr>
<tr>
<td>Text Input</td>
<td>EditText</td>
<td>UITextField</td>
<td>TextBox</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison between jQuery widgets and native OS widgets

From the presented list it is possible to conclude that there are some differences between the native widgets provided by each of the operating systems. Android is the most complete system while iOS and Windows Phone are more limited. This supports the use of the high-level JavaScript library, providing a vast list of widgets available at any platform.

In order to provide effective interaction RemoteU¡ should support the more relevant and useful widgets. Cardoso et al. [16] research presents the most used and important interaction tasks and controls when developing public display applications, divided in six different categories: Select; Data entry; Upload; Download; Signal presence; Dynamic manipulation. RemoteU¡ allows implementing four types of
categories identified, through the creation of screens containing instances of the more common classes of widgets. Table 3.3 presents the widgets association with task categories.

Besides the set of widgets provided by the library, it is also possible to implement other types of interaction like displaying text or images on the screen, or send files to open in the appropriate application. The list of widgets is presented in Table 3.2.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Text field</td>
</tr>
<tr>
<td>Check box</td>
<td>Image</td>
</tr>
<tr>
<td>Radio Button</td>
<td>Send File</td>
</tr>
<tr>
<td>Text box</td>
<td></td>
</tr>
<tr>
<td>Slider</td>
<td></td>
</tr>
<tr>
<td>Range Slider</td>
<td></td>
</tr>
<tr>
<td>Date Picker</td>
<td></td>
</tr>
<tr>
<td>Bar/QR Code Reader</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: RemoteUI proposed UI widgets

This final list was obtained taking into account the widely available UI programming frameworks on desktop (Windows, MAC OS X) and mobile operating systems (Android, iPhone, Windows Phone), on popular programming languages (Java SWING or Python TkInter) and web technologies (e.g. jQuery).

Since the target mobile devices are smartphones, usually used in portrait mode, the widgets layouts do not need to be highly configurable, though using jQuery Mobile allows them to automatically adjust to the screen size. In RemoteUI the widgets are displayed in a top-down layout and in the order they are created (section 3.3).

In section 4.4 are presented some considerations and the advantages of implementing this high-level library instead of each OS native widgets.

<table>
<thead>
<tr>
<th>Select</th>
<th>Data Entry</th>
<th>Download</th>
<th>Signal Presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Text Box</td>
<td>Send File</td>
<td>Button</td>
</tr>
<tr>
<td>Check Box</td>
<td>Date Picker</td>
<td>Bar/QR Code Reader</td>
<td></td>
</tr>
<tr>
<td>Radio Button</td>
<td>Bar/QR Code Reader</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slider</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Range Slider</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: List of widgets according to tasks

### 3.3 Programming API

This section presents the programming API, detailing the methods that are available and how to define them.

In order to create the remote screens and retrieve the user input, the programmer uses a set of classes (Java or C++). These classes encapsulate the remote screen and allow the definition of the widgets that will be presented. The system is responsible for encoding and decoding the messages so that the widgets are automatically generated on the screen of the mobile device, and the response sent back.
In order to better understand the API usage, we first present a simple example. So imagine that a programmer needs to present the user with a screen asking to input his name. This is as simple as generating a screen with a textbox widget. Figure 3.2.a) presents the C++ code needed to generate this interface screen, and Figure 3.2.a) shows the respective output result on the mobile device. The creation widget method (addTextBox) receive as parameter the name/label of that widget.

```cpp
//Create object RemoteWidgets
RemoteWidgets rw;
//Add text box with label "Name:"
rw.addTextBox("Name:");
//Create Submit button
rw.addButton("Submit");
//Get bytes to send
Byte[] bytes = rw.getBytes();
//Send bytes to socket
write(socket, bytes);
```

![Figure 3.2: C++ code example and corresponding mobile screen output](image)

Now, let's consider another example where the programmer needs to generate a screen requesting to input the name and age of a user, that could be used in a welcome screen of an application. A simple way to implement this feature is to generate a textbox widget for the name input and a slider to select the user age. Before both of them a text field is generated in order to present some instructions to him. Figure 3.3.a) presents the necessary code to generate a screen containing the three widgets and Figure 3.3 shows the output result. It also presents an example of an input that will be used, throughout this thesis, to detail some features of the system.

In widgets that require more complex configuration, like the slider (min, max and initial value), the creation method receives a vector with those values. In Table 3.4 are presented the list of available widgets with the corresponding methods and parameters. Some of these parameters are required, as for example the label of the widget, others are optional. When the optional parameters are not defined by the programmer, they are set with the default values. For example, in the case of the slider, if the programmer only decides to define the label, the other parameters are set by default with the following values:

- Value: 50
- Scale Min: 0
- Scale Max: 100
In section 4.3 we present the details regarding the widgets message serialization (Protocol Buffers). Besides the creation and update methods of widgets, there are also some very useful methods for some other purposes:

- **Clear Screen** - clear all the widgets in the WebView
- **Close Window** - close the WebView window
- **Asynchronous mode** - receive results whenever there is a change on users input

After the instantiation and configuration of the `rw` object the programmer should send it through the selected communication channel. As mentioned before the socket connection must be established by the developer. Then when the object is received in the mobile device the widgets are presented in the same order as defined in the code.

```cpp
//Create object RemoteWidgets
RemoteWidgets rw;
//Add text field
rw.addText("tf1", "Hello. Please enter your name and age.");
//Add text box with label "Name:"
rw.addTextBox("Name:");
//Create vector with parameters to the slider
vector<string> params;
params.push_back("Age:"); //label
params.push_back("20"); //value
params.push_back("0"); //scale min
params.push_back("100"); //scale max
//Create slider
rw.addSlider(params);
//Create Submit button
rw.addButton("Submit");
//Get bytes to send
Byte[] bytes = rw.getBytes();
//Send bytes to socket
write(socket, bytes);
```

![RemoteUI!](image)

**Figure 3.3:** C++ code example and corresponding mobile screen

In order to retrieve the user input, the program should read from the communication channel a representation of the user input. Figure 3.4.a) presents the way to process in synchronous mode, thus this code is executed after the user pressing the Submit button: only the final results are returned.

Figure 3.4.b) shows how to program the asynchronous behaviour. Every time the user changes the value of a widget it is transmitted from the mobile device. These multiple answers are read through the use of a loop, that exits when the response object states that the submit button was pressed. In the next section we detail these interaction behaviours.
Table 3.4: RemoteUI API methods

<table>
<thead>
<tr>
<th>Widget</th>
<th>Function</th>
<th>Required Parameters</th>
<th>Optional Parameters</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>addButton</td>
<td>Label</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Check box</td>
<td>addCheckBox</td>
<td>Label(s)</td>
<td></td>
<td>Vector</td>
</tr>
<tr>
<td>Radio Button</td>
<td>addRadio</td>
<td>Label(s)</td>
<td></td>
<td>Vector</td>
</tr>
<tr>
<td>Text box</td>
<td>addTextBox</td>
<td>Label</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Password</td>
<td>addPassword</td>
<td>Label</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Slider</td>
<td>addSlider</td>
<td>Label</td>
<td>value, scale min,</td>
<td>Vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>scale max, step</td>
<td></td>
</tr>
<tr>
<td>Range Slider</td>
<td>addRangeSlider</td>
<td>Label</td>
<td>value min, value</td>
<td>Vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>max, scale min,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>scale max, step</td>
<td></td>
</tr>
<tr>
<td>Date Picker</td>
<td>addDate</td>
<td>Label</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Code Reader</td>
<td>addScanButton</td>
<td>Label</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Image</td>
<td>addImage</td>
<td>Path, Label</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Image</td>
<td>updateImage</td>
<td>Path, Label</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>File</td>
<td>addFile</td>
<td>Path, Label, Type</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Text</td>
<td>addText</td>
<td>Label, Text</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td>Text</td>
<td>updateText</td>
<td>Label, Text</td>
<td></td>
<td>String/Int</td>
</tr>
<tr>
<td></td>
<td>clearScreen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>closeWindow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>asyncMode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Response resp;
read(sock, buffer, max_len);
resp.getResponse(buffer);
// for each item in the response
foreach (Result item in resp){
    cout << item.key() << item.value();
}
//verify final input
```

Figure 3.4: Remote screen results processing pseudo-code: a) synchronous mode, b) asynchronous mode

The response objects contain a key-value pair list: the keys are the widgets identifiers, while the values correspond to the user input. Figure 3.5 presents an example of a set of received results that could be generated from the screen presented in Figure 3.3, and are displayed in a command line terminal. The processing of the results is done with a simple "for loop".

---------------------
Name: Miguel A. V. R. Carvalho
Date: 06/23/2015
button, Submit

Figure 3.5: Received results example
3.4 Interaction models

There are two possible types of communication with interaction interfaces: atomic or asynchronous interactions. When a user is queried with an interface containing multiple widgets, for example, a text input and a checkbox, it is possible to gather all of the information when the user submits the final result, or it may be useful to have each interaction result asynchronously. RemoteU¡ allows the implementation of these two types of interaction. We thus make an analysis of both cases so that the programmers may be aware of the advantages and disadvantages of the two approaches.

Atomic interaction is the most useful type since it implements request-response transactions, much like an fgets call in command line interfaces. The main difference is that the interaction process is programmed on the client and presented in the mobile device and allows the input of complex data. The main advantage of having atomic interaction is that it is much simpler to implement, because it doesn't require any methods to deal with multiple responses. In order to do this the programmer can use a few lines of code like the example presented in Figure 3.3 to generate the screen and then process the response like the example presented in Figure 3.4.a).

Figure 3.6 presents the interaction between the various system components when generating an interface screen, presenting it in the mobile device, and then receiving the user input. First the programmer defines what widgets will be created on the remote device, defining them in a API class object, as already demonstrated in the previous section. The representation of the screen is sent to the remote device (function getInput) and the application can now wait for the response containing the user input. The RemoteU¡ components serialize the object and send it through a provided socket. On the mobile device the system parses the received object and shows a representation of the actual widgets. The user finalizes the interaction when presses the submit button and starts the transfer of the results that are encoded in a response object. Now the results can be processed in the client application. It is a simple and direct process that generates a clear result of the input choices, as it can be seen as an RPC procedure.
It is also possible to implement an asynchronous behaviour, in the case that the programmer wants to receive results whenever the user changes his options, or just doesn’t necessarily need a response and just wants to generate new screens, like for information purposes. Figure 3.7 represents a case where the programmer receives the result whenever there’s a change on the user input. The rest of the interaction process is similar to the atomic mode. We now mention the main difference in terms of programming:

- **Generate screen**: based on the example 3.3 the only difference is to add a line with the method (asyncMode) before sending the object (the line where it is declared isn’t important, only the widgets are taken into account for the display order)

- **Process response**: it is the same procedure that is presented in 3.4.b)

There are some aspects to consider when dealing with asynchronous interaction, but the most common and difficult problem is the fact that users often change options before submitting the final result, and this implies that we have to deal with multiple responses for the same widget. For example, in the presented example (Figure 3.7), a user receives an interaction screen with multiple selection widgets, like check boxes. As he checks each option, the device communicates the result. Though if the user chooses to uncheck a previous selected option, the system has to communicate the updated result. This will require the system to deal with multiple responses for the same widget, resulting in a more complex implementation of the client code, and also, it may represent an overhead in the communication stream. It’s important to mention that in other cases, like in a text box widget, the result is sent when the widget loses focus, not with every character change.

Thus it is important to consider if it is really an advantage to have asynchronous interactions, that it overcomes the problems that may result from this kind of approach.
Complementary to the previous modes, it is also possible to update the fields on the mobile device while the user interacts. Images or text fields can be dynamically updated in an active presented screen, as shown in figure 3.8. This allows a more dynamic interaction with the user. This is very useful when the programmer doesn’t want to change the already presented widgets but has the need to provide some feedback or instructions to the user.

To understand how these modes can work together in order to achieve a good user experience, we
now present an example based on the one of Figure 3.3.

Imagine a programmer uses a similar welcome screen in an application, but using the asynchronous mode. This means that when the user enters his name on the text box that result is sent to the client application. With this data the programmer may then update the screen, to instead of presenting the text field with "Hello, please enter your name and age", show "Hello George Frideric Handel, now select your age and click the submit button". Figure 3.9 shows the necessary code and result of this update.

This is a very simple process, since it isn’t necessary to generate a new object, only update the RemoteWidgets object with the method updateText(), using the Text field id and the new text to be presented. This has the advantage of maintaining the original fields of the screen in the object, with the exception of the updated ones. Since there could be made a significant number of updates, we chose not to store the record of the updates, only the last one.

```java
//Update text field
rw.updateText("tf1", "Hello George Frideric Handel, now select your age and click the submit button.");
//Send object to socket
write(sock, rw.serialize(), rw.len());
```

Figure 3.9: Code to update screen and corresponding result on screen

In all the presented cases the interaction flow is defined by the structure of the source code (not event driven) and is programmed using the developed API that provides methods to generate each type of widget mentioned in section 3.3. After the reception of the results, the programmer can decide what action to take and what screens to show.
Chapter 4

Implementation

This chapter presents the detailed implementation of RemoteU, first presenting an overview of the system processes flow and of the provided libraries. Then an approach of the communication serialization method is done, followed by some considerations about the OS support and security mechanisms.

4.1 Overview

Figure 4.1 presents how the RemoteU is structured and implemented. The client application is represented on the left side, where RemoteU provides some library files to include in the project so that the programmer can invoke the necessary methods to generate the messages that contain the description of the widgets. The methods are very simple and intuitive as already demonstrated in the previous sections.

The right side of the figure presents the mobile server application. In order to parse the messages in the mobile devices, we provide the communication library to include in an Android application. Once again there is a class that allows to create objects in order to decode the messages. We now detail the process flow between the two applications.
4.1.1 Client to server interaction

1) Create Objects / Method calls

First the programmer defines an instance of the class *RemoteWidgets*, which consists in a representation of an interface screen, and adds the necessary widgets resorting to the available API methods defined on that class. When all the widgets are defined a call to the function *send* is made providing a socket as parameter.

2) Sending RemoteScreen

The system serializes the object with Protocol Buffers (*RemoteScreen* object) and then send it through that socket. The details of Protocol Buffers are presented in the section 4.3.

3) Parsing / Javascript calls

After the *RemoteScreen* object is received in the mobile application, it is parsed with an instance of the *GenerateScreen* class. We detail this later in this section when referring to the libraries. The resulting string is a concatenation of all the JavaScript calls that will be injected on the page loaded in the *WebView*. So returning to the example of Figure 3.3 the result of the parsing is the following string:

```javascript
javascript:[screen(), generateText("tf1", "Hello. Please enter your name and age."),
generateTextBox("Name:", "NORMAL"), generateSlider("3", "Age:", "30", "0", "100", "1"),
generateButton("Submit"))];
```

4) Widgets Rendering

As mentioned, the rendering of the widgets is made resorting to the use of a *WebView*, that loads a web page that contains the JavaScript library. This library handles the requests that result from the parsing of the received objects, and presents the widgets.

4.1.2 Server to client interaction

4) Returning results

After the user finalizes the interaction the results are returned from the *WebView* resorting to the *JavascriptInterface* class. In asynchronous mode every time a user makes a change on an input widget the corresponding value is returned.

3) Creating Response Object

The *Response* object is generated creating an instance of the class *GenerateResult* and passing the results as parameter to the method *getResults*. This method is executed in the *JavascriptInterface* class.
2) Sending Response

After the Response object is serialized it is sent to the socket provided by the programmer. Again, the details of Protocol Buffers serialization are presented in the section 4.3.

1) Receiving Response results

The Results class allows receiving the objects regarding the responses. As already shown (Figure 3.5) they are in the form of an array of key-value pars: one single result in asynchronous mode; array of all results in the end.

4.2 Libraries

Table 4.1 presents the library classes for the client application.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoteWidgets</td>
<td>API class</td>
</tr>
<tr>
<td>Widgets</td>
<td>Protocol Buffers</td>
</tr>
<tr>
<td>Results</td>
<td>Protocol Buffers</td>
</tr>
</tbody>
</table>

Table 4.1: RemoteUI client library (C++)

The RemoteWidgets class is the API that was already presented in the previous chapter. The other two classes are generated by Protocol Buffers compiling. The Widgets class is generated by Widgets.proto file compiling and Results by the Results.proto, respectively. These files contain the description of the messages that are used in each case (details in subsection 4.3).

The rest of the files that aren’t presented on the table are the Protocol Buffers internal library.

In table 4.2 we present the library classes regarding the client applications.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenerateScreen</td>
<td>Parses the RemoteScreen Object</td>
</tr>
<tr>
<td>GenerateResult</td>
<td>Create Response Object</td>
</tr>
<tr>
<td>JavascriptInterface</td>
<td>Communication from WebView to app</td>
</tr>
<tr>
<td>WidgetsProtos</td>
<td>Protocol Buffer object</td>
</tr>
<tr>
<td>Results</td>
<td>Protocol Buffer object</td>
</tr>
<tr>
<td>IntentIntegrator</td>
<td>ZXing interface</td>
</tr>
<tr>
<td>IntentResult</td>
<td>ZXing interface</td>
</tr>
<tr>
<td>screen.html</td>
<td>JavaScript library</td>
</tr>
</tbody>
</table>

Table 4.2: RemoteUI server library
The `GenerateScreen` class is used in order to parse the received `RemoteScreen` objects, and this is done with the `ListWidgets` method. Since Protocol Buffers doesn't natively provide a way to control the order in which the widgets were defined, we had to include a field in each type of widget to achieve this. Thus the `ListWidgets` method first looks at the most simple cases, the last being the rendering of an interface screen. Next we present the steps that are coded in this method:

- Check if it contains a `clearScreen` or `closeWindow` message
- Check if it contains any update methods
- Create list of widgets with the defined order
- For each widget generate JavaScript string with the necessary parameters
- Returns a string with all the JavaScript call methods

The `GenerateResult` class is used to create the `Response` Object, and resorts to the `Results` class from Protocol Buffers to create an instance of that object.

The Javascript Interface allows the communication of the results from the `WebView` to the application. The JavaScript interface is part of the `WebView` API and allows JavaScript code inside the `WebView` to invoke application code.

The `WidgetsProtos` and `Results` classes are similar to the ones in the client library, and are used to serialize the Protocol Buffers objects.

Both classes from the `ZXing` interface are used to provide communication between the client application and the Bar/QR code reader application. This allows the application to launch the reader when needed, and after the code is successfully read the result is automatically returned.

The file `screen.html` is the web page that is loaded on the `WebView` that allows the rendering of the widgets. It contains the JavaScript library, which are all the necessary functions to create and update the widgets. Next we present the code of the functions that are invoked in the presented example (welcome screen in Figure 3.3):

```javascript
//Clear screen
function screen() {
    $( "in" ).empty();
    $( "out" ).empty();
}

//Generate TextField
function generateText(l,t) {
    var tag = '<div id="' + arguments[0] + '"><p>'+ arguments[1] +'</p></div>');</n    $( "out" ).append(tag);
}
```
//Generate TextBox
function generateTextBox() {
  var form = ('<form name="TextForm" onsubmit="stop(); return false;"/>
  if (arguments[1] == "NORMAL") form += ('<label for="' + arguments[0] + '">' + arguments[0] + '</label><input id="textinput" type="text" name=""+arguments[0]+"
  id=""+arguments[0]+"" />
  else form += ('<label for="' + arguments[0] + '">' + arguments[0] + '</label>
  <input id="textinput" type="password" name=""+arguments[0]+"
  id=""+arguments[0]+"" />
  form += '</form>
  $( "out" ).append(form).trigger( "create" );
}

//Generate Slider: Arg1 -> order; Arg2 -> label; Arg3 -> Value; Arg4 -> Scale_min;
Arg5 -> Scale_max; Arg6 -> Step
function generateSlider() {
  var order = arguments[0];
  var form = ('<form name="Slider">
  </form>
  $( "out" ).append(form).trigger( "create" );
}

//Generate Button Widget
function generateButton(label) {
  if (label == "Submit") var form = ('<form name="ButtonForm"><input type="button"
  name="submit" data-inline="true" value=""+label+"" onclick="values(this.value)" />
  </form>
  else var form = ('<form name="ButtonForm"><input type="button" name="button"
  data-inline="true" value=""+label+"" onclick="values(this.value)" />
  $( "out" ).append(form).trigger( "create" );
}

All of these functions are called injecting the JavaScript from the application, based on the information
that is received from the network, this is, whenever a remote screen is received. Some other functions
like the presented here exist, in order to generate the other widgets.

### 4.3 Communication

Regarding the communication we need to identify the best data representation protocol to use to serialize
the messages exchanged by the client and server applications.

#### 4.3.1 Language Format

XML is the most common language used to describe various types of data that can be read both by
humans or machines, though JSON turned to be very popular as an alternative. More recently Google
developed a better solution, Protocol Buffers. We now present an analysis of the main features regarding
the XML and JSON technologies, and then present the advantages of Protocol Buffers.

**XML VS JSON**

In terms of simplicity, interoperability and openness it's fair to say that they are almost the same, maybe
JSON being a little more simple than XML due to having a smaller grammar. Though this brings a
disadvantage in terms of extensibility, as JSON only allows storing data like text and numbers, and
XML gives the possibility of storing documents. Due to this JSON is more human readable, but both
benefit from this feature. As mentioned only XML gives the possibility to share documents, and this
includes images, charts, graphs, audio and video. On the other hand JSON is the best choice for sharing
traditional data as it stores in arrays and records while XML stores data in trees. This is important when
importing data from a file to JavaScript. This may be the most relevant advantage of JSON that can
apply to our work, though it doesn’t overcome the XML advantage of a much larger variety of data, as
some applications may benefit from this feature. From this analysis we conclude that XML represents a
better solution when compared to JSON.

**Protocol Buffers**

Protocol buffers are Google’s language-neutral, platform-neutral, extensible mechanism for serializing
structured data. [9] There are several tests that prove that Protocol Buffers have better serialization
performance when comparing to other alternatives. For example, Figure 4.2 shows the comparison
regarding the serialization of objects to files and vice-versa (reads and writes) using Protocol Buffers
and other languages like XML or JSON [4]. The tests were conducted with variable number of child
objects, and repeated for different file sizes. The results were consistent for files with sizes up to 100MB.

The results show that Protocol Buffers have better performance, both in reading and writing, when
compared to all other solutions. Also they show that the file sizes are smaller, leading to better efficiency
when transmitting over a network. For these reasons this mechanism was chosen over the other two
presented alternatives.
Next we detail how the widgets were implemented with Protocol Buffers. This is done defining message formats in a .proto file, in this case Widgets.proto. Each data structure is serialized adding a message to the file that contains a name and a type for each field. To easily understand this we now present an example of the text box widget.

```protobuf
message TextBox {
  required int32 order = 1;
  optional string label = 2;
  optional int32 label_i = 3;
  enum BoxType {
    NORMAL = 1;
    PASSWORD = 2;
  }
  optional BoxType type = 4 [default = NORMAL];
}
```

The first field is the order, and this is used to define how the widgets are displayed, so all the widgets have this property. It is a required field but is automatically defined by the API each time the programmer defines a widget. Next there are two possible ways to define the label of the text box: one is using a string, and the other is with an integer. This last option allows having a list of strings defined in the mobile server application that is then referenced throughout the API methods. Both these fields are optional so that any of them can be set. Enum types are used to set a predefined list of values: here we specify that a text box can be "NORMAL" or "PASSWORD". This is to differentiate a regular text box from a password text box, that in this case must show "***" instead of the actual text. With this property we can
avoid having to define two types of messages for each kind of widget. The last field is the setting of the "BoxType", which by default is set to normal, to simplify the implementation of the API. Only when it is required to be a password type we need to set the value to "PASSWORD", and that's why it is defined as an optional field.

The next structure is a check box widget.

```java
message CheckBox {
    required int32 order = 1;
    repeated string label = 2;
    repeated int32 label_i = 3;
}
```

The check box has the order field, as all the widgets must have, and then there are again two ways to define the options, with a string or an integer. Here these fields are defined as repeated since they can be defined repeatedly, accordingly to the number of options needed.

```java
message Slider {
    required int32 order = 1;
    optional string label = 2;
    optional int32 label_i = 3;
    optional double value = 4 [default = 50];
    optional double scale_min = 5 [default = 0];
    optional double scale_max = 6 [default = 100];
    optional double step = 7 [default = 1];
}
```

The slider uses the same schema as the previous structures to the label, and double type to the value fields. Once again there are defined default values so they don't have to be set in the API every time they are created.

```java
message Date {
    required int32 order = 1;
    optional string label = 2 [default = "Date:"];
    optional int32 label_i = 3;
}
```
The API date widget is one of the examples that may be defined with no argument at all. This is because it was defined the default label to "Date:“, which is the most common used. If the programmer wants another label he can set it with a string or integer argument.

```protobuf
define SendFile {
    required int32 order = 1;
    required bytes file = 2;
    required string type = 3;
    optional string label = 4;
    optional int32 label_i = 5;
}
```

The "SendFile" structure allows to send files from the client to server application. It requires to set the bytes of the file to transfer and also the mime type of the file.

At last we present the main message type which is the `RemoteScreen`, that is a representation of the whole screen. All the fields are the structures that represent each widgets. They are all repeated fields so that they may be defined for as many widgets needed.

```protobuf
define RemoteScreen {
    repeated TextBox text_box = 1;
    repeated CheckBox check_box = 2;
    repeated RadioButton radio_button = 3;
    repeated Slider slider = 4;
    repeated RangeSlider range_slider = 5;
    repeated ShowImage show_image = 6;
    repeated UpdateImage update_image = 7;
    repeated Date date = 8;
    repeated Button button = 9;
    repeated SendFile send_file = 10;
    repeated Text text = 11;
    repeated UpdateText update_text = 12;
    repeated ClearScreen clear_screen = 13;
    repeated ScanButton scan_button = 14;
    repeated CloseWindow close_window = 15;
    repeated Async async = 16;
}
```

One of the main features of Protocol Buffers is that they provide backwards compatibility, which is useful when one has to deal with an older version or evolve the system. A more important aspect is that
they also provide great extensibility. If a programmer, for example, wants to implement another property to a widget, it’s as simple as adding a line with that definition to the `Widgets.proto` file and recompile it.

### 4.4 OS Support

In this section we present some considerations that lead us to choose an high level library for widgets rendering (presented on section 3.2), instead of using the proprietary OS as support of user interactions. Since RemoteUI system has to provide user interface interaction in real-time and in asynchronous mode, using directly the UI widgets provided by the operating system may be difficult to implement: both with respect to devices compatibility (Android, Windows, IOS) and to the dynamic generation of the interface elements. Also as illustrated before (table 3.2) it is possible to see that the type of widgets differ from one system to another, this being a major problem in generalization. Developing an application using available standard dynamic UI technologies is the best way to guarantee that the application meets the requirements defined in chapter 3. This implies the implementation of a display engine (using a web browser rendering engine), and the use of JavaScript together with a widget library to dynamically create the interface elements.

### 4.5 Security and Resource Discovery

One important aspect that we need to consider is security mechanisms, though we don’t implement these, as it should be dependent of the programmers choice, regarding the kind of connection and discovery mechanisms that he decides to use. As proposed in the architecture presented in section 3.1, the middleware operates in a network layer that may be represented by a various number of technologies. This implies, in some cases, like wireless networks, that there are implemented authentication mechanisms. There are several studied solutions that can be adopted but this has to be implemented by the programmer [43, 31].

Developing interaction mobile applications usually also require having resource discovery mechanisms though we won’t address this feature, as it is out of the scope of this work.
Chapter 5

Evaluation

Evaluating a middleware like RemoteUI is challenging, mainly because it is related to recent areas of research, not existing many similar solutions, given the fact that most of them are web and cloud based architectures. Our best approach to evaluate this work was to consider some applications that could be deployed in some different areas, and try to verify whether the main requirements were met. The most relevant application is related to an European project that uses RemoteUI as the UI framework, which will be addressed on the next section.

Another important feature that was taken into account was the communication performance, and RemoteUI was compared to two other systems, in order to evaluate the results.

An analysis about the programming model was done, comparing it to some of the related works. We also evaluated the costs that can be saved with our solution when compared to others.

This chapter ends with some considerations about security issues related to the technology of the UI, namely, the WebView.

5.1 Demonstration scenarios

The first thing we present in this section are a few demonstration scenarios to better understand where and how the middleware can be implemented, showing its potential. There are several case scenarios where RemoteUI can be useful, although we only present the most relevant cases, but first we address a prototype of an application that is using the middleware.

5.1.1 PCAS

PCAS project [37] is an European Funded project aiming at reducing data exposure when using mobile devices. PCAS will provide a secure mobile data storage and transfer environment, using biometric technologies to ensure the identification and authentication of users. The secure storage and biometric authentication is done inside the SPD (Secure Personal Device), which is a smartphone accessory that mimics a protective sleeve but will store information, authenticate the user and mediate secure communication with service providers.
The smartphone will allow the SPD to communicate with the remote services, but will also be used as the user interface. In order to ease the development of the SPD code, RemoteUᵢ is being used as the UI framework.

Developers of the SPD code resort to the presented APIs to program interactions with user (for instance when authentication is required) and USB is used to connect the accessory to the mobile device. The PCAS software infrastructure running on the smartphone includes RemoteUᵢ middleware to show the widgets to the user. This is done with a simple Android application that runs a background service that handles the requests, whenever is required to present some interface screen to the user.

The first prototype is already running allowing the interaction with the user using RemoteUᵢ [12].

5.1.2 Public Displays

We now present the first prototypical scenario regarding public displays, since they are starting to emerge on most of public spaces in our daily lives.

One of the most common applications that are provided in this context are mapping applications that allow users to search points of interest and also get directions to a certain destination. This is very useful when the user is travelling and doesn’t want to download the maps to the mobile phone, or use roaming data. Another case is when one wants to get directions inside buildings.

In any case, this usually requires that the public display has a touch screen so that users can interact and input data. This makes these displays more expensive, requires study on the display location and makes it difficult to be shared with multiple users.

Using the mobile phone as an input mechanism can solve these drawbacks, and RemoteUᵢ allows the easy implementation of the necessary interactions, even in already existing systems. RemoteUᵢ does not require significant changes in a developed application in order to extend the UI into mobile devices.

Figure 5.1 illustrates a possible usage scenario, where there is a map application running on the public display. When a user approaches, the system may present him with a welcome screen and then ask to input his target point using the mobile device. The user then may type the location using the device input methods and select the required action (show the route on map or download it to the smartphone). Besides being easier to the user as he is well familiar with his device, it has the advantage that it can automatically transfer the file allowing to release the display for other users.
5.1.3 Internet of Things

Another good example of use is in the context of Internet of Things, for instance a smart refrigerator that can keep track of the products you put in, and help you manage the stock alerting when a product needs to be replenished.

To manage the inventory, either all products have RFID tags (and the refrigerator may automatically detect it) or it requires the scanning (bar code or QR code) of each product. In either cases the refrigerator should have these readers, though QR codes are better than bar codes since they can carry much more information in a smaller place and are capable of being read in 360 degrees, from any direction. Some products already have QR codes on their packages with links to the producer web page, or some kind of informations, thus it is expected to have some standards defined in a near future. There are different approaches to manually insert products on the refrigerator and an efficient way may be using the mobile phone as scanner.

In both cases the smartphone can be used to replace the reader on the refrigerator. Whenever the user operates the refrigerator, the various options can be presented on the smartphone screen (Figure 5.2.a). Then when an option to add one or more products is selected the user may use the smartphone camera to scan the products code (Figure 5.2.b).

RemoteUI can be useful to implement this feature in the application, allowing the presentation of the screen in the phone with just few lines of coding. As mentioned before, one of the main advantages of its use, is that it may be used as an extension to an already developed system.

5.1.4 Ambient Intelligence

The last example, in the context of ambient intelligence, is in home automation domain. Imagine you want to use the mobile phone to control the lighting system of a living room. There are already several proprietary systems available at the market, but they all represent a major effort in development.
Figure 5.2: Smart refrigerator UI on smartphone: a) Menu options b) barcode reader

So instead of the programmer having to implement the interfaces according to each light device, that in many cases may be different from one another, the devices could generate their own interactions, and the programmer just needed to aggregate and route the requests and responses.

Even in the present reality RemoteUį can help the developer providing interaction abstractions and allowing generating an interface screen with just a few lines of code. Figure 5.3 shows the necessary code (in C++) to generate the interface screen example presented in Figure 5.4. It allows changing the present configuration of two sets of light of a room, the ambient and task lighting.

```cpp
//Create RemoteWidgets object:
RemoteWidgets rw;
//Add Text field:
rw.addText("Id-1", "The system automatically adjusted the ambient "
  "and task lights by default. You may change it now.");
//Add button
rw.addButton("Turn off ambient lights");
//Define slider options in vector
vector<string> optionslider;
  optionslider.push_back("Choose intensity of lamp lights:"); //title
  optionslider.push_back("3"); //default value
  optionslider.push_back("1"); //min. value
  optionslider.push_back("5"); //max. value
  optionslider.push_back("1"); //step value
//Add Slider to adjust lamp lights:
rw.addSlider(optionslider);
//Add button to close screen
rw.addButton("Close");
//Get bytes to send
Byte[] bytes = rw.getBytes();
//Send bytes to socket
write(socket, bytes);
```

Figure 5.3: Code to generate UI to control lighting system

In this case the asynchronous interaction model can be used so that the system may automatically
reflect the changes with the user input interaction, and make it easier to decide his definitions.

5.2 Communication Performance

In order to evaluate the performance of the system some interaction tests were conducted and the messages payloads were measured with different screen configurations. In these tests, besides the examples of Figures 3.3 and 5.1, four more tests were performed with variable number of widgets and with variable complexity (for example, Figure 5.5).

Table 5.1 presents the payload of the requests (creation of the screens) and response messages (after user clicks submit). We do not consider the “Submit” button as part of the number of widgets used, but it is included in the total payload.
The results show that there is a fixed size message payload (corresponding to the Single Widget test), and that it is minimal, only about 30 bytes in average.

The Request size varies with the number and type of the widgets created. To simple widgets (that only require the transmission of the identifier the request size does not increase much, ranging from 27 bytes to 196 bytes. If more complex widgets are used (those that require more labels, such as a checkbox), the size of the request increases with the number of widgets, number of options and length of the option labels. For example, a checkbox with three options (Red, Green, Blue) requires a larger message payload than a date: 22 bytes against 4 bytes.

The same happens with the response object, in some widgets it is fixed, like for example the date, since the number of response characters is always the same (payload of 22 bytes). In other examples it varies with the user input. The response payload for a text box varies with the length of the input text, for example with the input *Frideric Handel* it has the cost of 27 bytes.

The worst result (2 kByte) corresponds to a screen with 10 widgets each one containing about 200 characters.

The results show that, in all cases, the payloads are minimal, leading to efficient communication independently of the bandwidth of the network link. This supports the fact that the system can be used in any kind of network configuration and with many kind of devices.

RemoteU\textsubscript{1} can also be compared with two existent remote screen technologies and techniques: VNC (or remote displays) and X11 (higher level protocol with remote widget creations). In this comparison the following screens (Figure 5.6) were implemented using RemoteU\textsubscript{1}, as android native application, and as X11 application.

<table>
<thead>
<tr>
<th></th>
<th>Request size (byte)</th>
<th>Response size (byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.3 example</td>
<td>44</td>
<td>87</td>
</tr>
<tr>
<td>Figure 5.1 example</td>
<td>127</td>
<td>73</td>
</tr>
<tr>
<td>Single widget</td>
<td>27 (average)</td>
<td>58 (average)</td>
</tr>
<tr>
<td>2 widgets (Figure 5.5)</td>
<td>29 (min) — 138 (max)</td>
<td>77 (min) — 437 (max)</td>
</tr>
<tr>
<td>3 widgets</td>
<td>48 (min) — 201 (max)</td>
<td>91 (min) — 646 (max)</td>
</tr>
<tr>
<td>10 widgets</td>
<td>196 (min) — 642 (max)</td>
<td>443 (min) — 2105 (max)</td>
</tr>
</tbody>
</table>

Table 5.1: Payload on different types of messages
5.2.1 VNC

To perform the first test scenario a VNC like application (TeamViewer\(^1\)) was installed on both devices, a smartphone and a desktop PC. A simple android application that displays the same set of widgets as presented in the RemoteU\(i\) application (Figure 5.6) was developed and presented on the remote device using TeamViewer. The application was developed using the same rendering technology as provided by the middleware. One of the examples required the user to scroll the screen, having a significantly impact in the traffic flow between the VNC client and server.

5.2.2 X11

The setup for the X11 case is very similar to the previous one, but a X11 application was developed using the TkInter\(^2\), which is a Python standard GUI (Graphical User Interface) toolkit. The widgets used in the designed screen replicate those in figure 5.6, with the exception of the date widget, that in this case was alternatively represented by a textbox. The interaction with the screen was very similar: entering the first and last name on the textbox, selecting the age on the slider, selecting two options on the checkbox and typing the data on the last textbox.

5.2.3 Comparison

The results of the remote interactions using RemoteU\(i\), Remote Desktop and X11 are presented in table 5.2. Each test only took a few seconds (the minimum required to fill in the form) and the traffic was

---

\(^1\)www.teamviewer.com

\(^2\)wiki.python.org/moin/TkInter
monitored and measured using Wireshark\textsuperscript{3}.

<table>
<thead>
<tr>
<th></th>
<th>RemoteUI (sync)</th>
<th>RemoteUI (asynch)</th>
<th>VNC</th>
<th>X11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5.6. a)</td>
<td>0.5 KByte</td>
<td>5.5 KByte</td>
<td>88 KByte</td>
<td>1000 KByte</td>
</tr>
<tr>
<td>Figure 5.6. b)</td>
<td>0.6 KByte</td>
<td>7 KByte</td>
<td>1738 KByte</td>
<td>1800 KByte</td>
</tr>
</tbody>
</table>

Table 5.2: Network traffic comparison with VNC and X11

As shown, there is a significant difference in the payloads between RemoteUI\textsubscript{i} and the other options. One of the reasons for this difference is the efficient encoding of the screen. In RemoteUI\textsubscript{i} a representation of the screen takes a few bytes, in X11 the objects creation is less efficient and in VNC a bitmap representation of the screen is transferred.

In RemoteUI\textsubscript{i}, using synchronous mode during the filling of the forms there is no data transfer: only when the application creates the remote screen and the user submits the form there is network traffic. In asynchronous mode, whenever the user changes the value of a widget the data is transferred, though it only depends on the actual user input (and changes on the widgets).

On all other options (VNC and X11), whenever a relevant event happens, data is transferred: VNC transfers data when the mouse cursor position changes and when screens are updated (user writes on a text box or scrolls the screen). X11 also transmits the mouse events, along with all keystrokes.

All tests demonstrate that RemoteUI\textsubscript{i} has much better performance in terms of traffic generated on the network, when dealing with this kind of interaction mechanisms. Even in asynchronous mode (updates sent with every widget change) RemoteUI\textsubscript{i} offers better performance than all other options. None of the generated traffic depends on the interaction duration, but only of the data produced by the user.

5.3 Programming model

We now compare the programming model of some similar systems with RemoteUI\textsubscript{i}, in order to demonstrate its easy implementation, and the advantages of imperative programming versus other common approaches, like event programming.

The most similar system compared to RemoteUI\textsubscript{i} is the PuReWidgets toolkit. In their work they provide an example of an "Hello World" application, and the code is presented in Figure 5.7. The necessary lines of code in RemoteUI\textsubscript{i} to present the same kind of interaction interface are shown below:

```c
RemoteWidgets rw;
rw.addText("Id-1", "Hello World!");
rw.send(socket);
```

It is trivial to see that RemoteUI\textsubscript{i} code is simpler, since it only requires to create an instance of a

\textsuperscript{3}www.wireshark.org/
RemoteWidgets object, add the text field widget, and then send it to the socket. The use of the API provides an easy implementation of interactions, even on an already deployed application.

On the other hand, PuReWidgets is based on a web service. The developer uses the available functions of the API, instantiating widgets and registering interaction event callback functions.

Event programming represents a more complex implementation, requiring more programming effort, when compared to imperative programming. Object-oriented methods represent several advantages in embedded systems programming [25], but still require the programmer to explicitly control and verify the state of the interaction.

```java
1 public class HelloWorld implements EntryPoint {
2     @Override
3     public void onModuleLoad() {
4         PublicDisplayApplication.load(this, "HelloWorld", true);
5         GuiButton guiButton = new GuiButton("helloButton",
6             "Hello World");
7         guiButton.setShortDescription("Say hello!");
8         guiButton.setLongDescription("Say hello to be greeted
9             by the HelloWorld application");
10        guiButton.addActionListener(new ActionListener() {
11            @Override
12            public void onAction(ActionEvent<?> e) {
13                PopupPanel popup = new PopupPanel();
14                popup.add(new Label("Hello " +
15                  e.getValue());
16                popup.show();
17            }
18        });
19        RootPanel.get("main").add(guiButton);
20     }
21 }
```

Figure 5.7: PureWidgets code for an Hello World Application

Similar to PuReWidgets, the web-based toolkit for remote direct manipulation interaction with public displays via smartphones is another example that resorts to interaction events programming. When users interact with the widgets, interaction events are sent to the application through a web socket connection. When these events are received they trigger the appropriate callbacks on the application code.

MoCHA system doesn't provide any API, and it is dependent on a service that can be deployed either on the cloud or on local infrastructure, so it is expected to rely on event based programming too. It also has another major drawback which is that both applications are browser-based dependent, thus limiting the implementation to web based languages.

5.4 Costs Analysis

Overall, RemoteUj implementation and programming API allows the reduction of the deployment, development and operation costs of a mobile interaction system.
5.4.1 Deployment costs

Regarding deployment costs, RemoteU¡ allows the use of low cost headless devices to implement complex applications and services, since processing and network requirements are minimal and don’t require specific computational resources (processor or memory). This makes RemoteU¡ being easily implemented in IoT context applications.

With respect to the implementation of applications in public displays, the replacement of a touch screen with a Wi-Fi adapter can also reduce implementation costs, specially in the case of a distributed infrastructure, like university campus or airports.

Furthermore, since all interactions are restricted to the pair headlessdevice ↔ mobiledevice, none of the devices need internet connection, since RemoteU¡ is not web-based nor web-service dependent.

5.4.2 Development costs

The simple programming model allows to reduce the development costs, requiring lower programming effort than comparable event-base or client-server programming. Even comparing with descriptive GUI languages (such as XUL [10]), all the UI state machine is programmed using the regular language structures: it is not necessary to program event handlers, nor explicitly manage the UI state.

If compared with a system where interactions are implemented using web server and services, RemoteU¡ will reduce the UI definition stage, the coding of the application logic and reduce infrastructure costs.

RemoteU¡ also guarantees that the look and feel of the application is consistent with that of the mobile device, with minimal programming effort. This is due to the use of the jQuery Mobile library that allows to create widgets similar to the native OS ones.

Another important feature is that it can be implemented with already developed applications not requiring much programming effort.

5.5 WebView Security

The WebView allows mobile applications to embed a browser with basic functionalities such as page rendering, navigation, and JavaScript execution. Its implementation is rather simple, just requiring the creation of an instance of the WebView class, and then use it to display contents.

WebView not only allows to display web contents, but it also enables applications to interact with the contents through its APIs. The interaction can happen from applications to web pages, or vice-versa.

RemoteU¡ implements the two cases:

- RemoteU¡ to web page: invoke JavaScript functions within the web page to display the widgets (it is also possible to insert the code). This interaction model also allows to monitor and intercept events within the web pages.
Web page to RemoteUI: the registered JavaScript interface allows that the JavaScript code embedded on the page can invoke the interface and return results from it.

The use of the WebView on mobile devices has raised some security concerns. Familiar browsers, such as IE, Chrome or Safari, were developed by trusted companies. Now applications can provide their own browsers, and most of them are developed by programmers not supported by recognized companies, so they can’t be trustworthy.

Thus we analyse some attacks that can be made to WebViews. In Luo et al. study [28] they define two types of attacks: attacks from malicious web pages (Figure 5.8 left) and attacks from malicious apps (Figure 5.8 right).

Since in RemoteUI the WebView is used to load a local web page, not accessing external content, we only address the second case. Thus the programmers may resort to the middleware to develop malicious apps, implementing this type of attacks. They are classified in two different categories:

1. **JavaScript Injection**: Android app can directly inject its own JavaScript code into any web page loaded within the WebView component.

2. **Event Sniffing and Hijacking**: Attackers can intercept the WebView APIs, and launch sniffing and hijacking attacks from the outside of WebView.

### 1. JavaScript Injection

In RemoteUI we use the loadUrl() API method to execute the JavaScript calls to the functions that render the widgets. The loadUrl() receives an argument of string type, that when starts with "javascript:" it will treat the entire string as JavaScript code, and execute it in the context of the web page that is currently displayed.

There are many attacks that can be made injecting JavaScript code into a web page using loadUrl().

We now mention two possible scenarios:

- **JavaScript code injection**: string containing a JavaScript program that is injected into the web page. When it is executed it fetches malicious code from an external server, and executes it.

- **Extracting Information from WebView**: malicious application can ask its injected JavaScript code to send out information from the page. This way the attacker may retrieve all the data entered by the user.
2. Event Sniffing and Hijacking  

*WebView* also offers a number of hooks to Android applications, allowing them to intercept events, and potentially change the result of them. The *WebViewClient* class defines 14 interfaces where applications can register event handlers, and makes it possible to observe or change the event.

- **Event Sniffing:** Host applications can know almost everything that a user does within a *WebView*, as long as they register an event handler. For example, the *onLoadResource* hook is triggered whenever the page inside *WebView* tries to load a resource, such as image, video, or flash contents. If the application registers an event handler to this hook, it can observe what resources the page is trying to fetch. Another example is to use the *onFormResubmission* hook. What it does is ask the Android application if the browser should resend the form, so it can get a copy of the data users have typed in the form.

- **Event Hijacking:** Applications can also hijack events by modifying their content. For example, whenever the page within the *WebView* attempts to navigate to another URL, the page navigation event occurs. *WebView* provides a *shouldOverrideUrlLoading* hook, which allows to intercept the navigation event by registering an event handler. Once the event handler gets executed, it can also modify the target URL associated with the event, directing the navigation to a different (malicious) URL.

Since we can’t deny the programmer from having access to the source code, it’s impossible to prevent these kinds of attacks. The only way that users can have some level of security, is using applications that are developed by trusted or certified entities. This is a common problem with any kind of application (mobile or desktop) that users are aware when they choose to install any software.

---

Chapter 6

Conclusions

In this work we presented RemoteU, a middleware that provides interaction abstractions, in order to help programmers developing applications or extending existing ones.

There are many factors that have motivated this thesis. One is that public displays are gaining popularity, and that users are feeling more interested in the possibility of using them as a resource to their needs. In the area of IoT, RemoteU will allow the interaction with a series of objects on the everyday life, for configuration, management and normal operation.

Though there are several existing solutions available, most of them are web or cloud based systems, and require the existence of external infrastructures, complex programming or demand high computational resources.

The RemoteU in conjunction with the programming languages control structures allow the efficient definition of complex interactions with the users. Besides the simple declaration of screens to present, the program logic allows the definition of the interactions without resorting to event programming, complex resource files or complex communication patterns.

The communication performance is high compared with other existing approaches, due to the reduced message payload (on the order of bytes). The efficient encoding of screens allow its implementation in any kinds of devices, even with power or connection constraints, since the communication requirements are simple with respect to infrastructure, data processing and data payload.

The work here described is being used in a prototype in the context of the European funded PCAS project and it was published and presented as poster at the Mobicom 2015 - The 21st Annual International Conference on Mobile Computing and Networking, in Paris, France in September 2015 [17].

6.1 Achievements

This work aimed to the development of a middleware to fulfil some initially defined requirements, and the presented evaluation validates each of them as follows:

- **Ease of programming** - The developed API allows the simple definition of screens and the interactions are defined by the program logic, resorting to regular language structures, without requiring
event programming;

- **Interoperability** - Its generality and communication channel agnostic allows the use of the middleware in the interactions with several resources (smartphone accessories, embedded device, IoT, public displays) by a common infrastructure;

- **Low overhead** - The efficient encoding of screen provides low message payloads and allows the use on ad-hoc, encounter networks, and the deployment on low resources devices;

- **Openness** - It is possible to develop a whole server-client application resorting to several languages, but also to integrate RemoteUI with several existing systems, extending its UI. Its efficient implementation, provides great extensibility, not only regarding the interaction widgets, but also allowing new API language implementations;

- **UI consistency** - The use of a high-level JavaScript library provides a consistent layout, independently of the device where it is presented.

### 6.2 Future Work

One of the important features that was out of the scope of this work, are low-level communication mechanisms, so this is one of our goals in future work. Thus we plan to extend this work to the network domain, investigating discovery and secure binding mechanisms.

The objective is to include on the API, a series of methods to allow that the devices can recognize each other and establish the communication. Implementing these kind of features will allow to have an integrated solution that can interact with multiple applications, presenting several advantages for programmers and users.

From the programmer perspective it makes the development simpler, as one can resort to the API to interconnect the devices, without having to implement these mechanisms in the UI server application. Another advantage is that, once the server app is installed on the user device, all the systems that integrate the middleware will have the opportunity to easily announce their services.

It also brings advantages to users since they won’t have to install multiple server applications on their devices according to each client application they wish to interact. This also solves some drawbacks related to security, as the application becomes more trustworthy, preventing the mentioned WebView attacks.

Finally we pretend to validate the work using small devices like an arduino or some other similar hardware. The objective is to develop a prototype using these devices in some case scenario related to the context of IoT.
Bibliography


[38] Pears, N., Jackson, D. G., and Olivier, P. Smart phone interaction with registered displays. IEEE Pervasive Computing 8, 2 (Apr. 2009), 14–21.


