UPPSALA
UNIVERSITET

IT 13 076
Examensarbete 45 hp
November 2013

Live Programming for Mobile Application Development

Paolo Boschini

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Live Programming for Mobile Application Development

Paolo Boschini

Live Programming is a style of programming where the result of editing code can be immediately seen. This approach differs from the traditional edit-compile-run development model, as it reduces the gap between the meaning of the program representation and its behaviour. By giving immediate feedback, live coding can speed-up the development cycle, enable exploratory programming, and give a better understanding of a program. The gap is especially notable in mobile development, where the development cycle also includes the deploy to device step. Live Programming has the potential to make mobile development both easier and faster. The goal of this project is to investigate if a Live Programming environment can speed-up and make mobile development easier, by building and testing a working prototype of a Live programming tool. The results of building and testing the working system with users show that live editing speed is achievable for basic mobile applications and that the interviewed developers found live feedback valuable.

Handledare: Mikael Kindborg
Ämnesgranskare: Arnold Pears
Examinator: Ivan Christoff
IT 13 076
Sponsor: MoSync

Tryckt av: Reprocentralen ITC

Acknowledgements

First and foremost, I would like to thank my supervisor Mikael Kindborg for his extensive guidance and immeasurable patience. I enjoyed every moment when working with him and I had a lot of fun discussing programming languages techniques and how to experiment with them. I also want to express my appreciation to the whole MoSync team for welcoming me and making me feel at home at their headquarters in Stockholm.

Moreover, I want to thank all the people who supported me during these years and during the cold winters up here in the north. Massimo and Mirella for their everlasting friendship, Micaela and Monica for their warmth and affection. You always gave me much more than I could ever have wished for.

Finally, I want to dedicate this work to my family, my brother and sister, and my parents, who have always believed in me and had the sensitivity to accept my choices without ever criticising them.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Since the early days of computing, the main interface to create software has been a text editor. Developers write source code in the form of text, save the program code to a file, and finally run it through tools such as compilers or interpreters. These intermediate tools translate the source code written by the developer, producing the same exact program but in a format that a computer can understand and can execute.

If anything goes wrong during the translation process, that is the program contains errors causing the compiler to fail compiling a piece of source code, the developer has to inspect the error logs, understand what went wrong, and correct the spots in the source code where the errors reside. These are usually syntactic errors generated from incorrect grammatical statements. In addition to syntactic errors, semantic ones are more subtle to notice since they can only be observed during the execution of a program. semantic errors are logical errors, and have the consequence of producing a result that differs

from what the programmer had expected to see.

During the process of writing code, it is highly likely that the programmer will introduce both syntactic and semantic errors in the source code. This happens not only to novice programmers, but also to experienced ones. As a consequence, most of the time is spent between reading lines of code trying to spot errors, and executing the edited program to see if the output is correct. During this process, the programmer is forced to understand what is going on blindly, since she can not see the result of what she is doing directly. The root cause of this problem resides in the fundamental difference of how humans think, compared to how a computer thinks. There is a gap between the representation which the brain uses when reasoning about a problem and the representation of the exact same problem that a computer will accept [52]. The mapping between the code and the desired result is not direct and it is not transparent. When programming, humans have to think like a computer, and keep all the instructions they write and the data flow of the program in the brain, which is a hard task to accomplish. Studies have tried to demonstrate that the problem may reside in the programming language that is been used, but it has been shown that the language itself does not have any impact on either the learning nor the act of programming itself [52].

1.2 Problem

The traditional development cycle – edit-compile-run-debug – prevents the developer from really seeing what she is doing, making it difficult to fully express ideas quickly and see the result produced by the program directly. In essence, we are using tools that separate us from our ideas and the realisation of them. Tools are one of the most important part of realising ideas and we should strive to build tools that can help growing ideas [57].

As discussed above in the background section, programming is hard and the traditional model of developing software could be improved by giving direct feedback to the developer when writing code. Recently, new platforms and solutions have been created to address those issues (as presented in chapter 2), except for mobile application development. Programming in itself is hard, and even harder is mobile development, where the developer has to face the complexity of a heterogeneous world of devices and different platforms with different programming languages. Developers have to face the challenge of device fragmentation between different systems and technical specifications.

To reach the largest number of potential users of an application, developers need to create the same application for different systems and devices, a task that requires a considerable amount of resources. This becomes even more difficult for novice programmers who want to approach mobile software development, since a broad set of skills (diverse tools and programming languages) has to be gained to develop applications for multiple platforms. This implies writing several code bases, increasing both the process of development and maintenance of the code.

In this project, an effort has been made to try to solve parts of these problems, and more specifically, four questions have been explored to see if a different approach to software development can help both novice and experienced programmers to develop mobile applications more easily.

1.3 Questions

- Is it possible to achieve "Live Performance" (as the ability to see the result of code changes in less than about one second) when developing on mobile devices?

- Which (network) architectures are suitable for Live Programming of mobile devices?
- Does Live Programming help developers to program mobile devices?
- What are the main differences that developers experience compared to conventional mobile development?

1.4 Goal

The aim of this project is to investigate if Live Programming can speed-up and make mobile development easier for both novice and experienced developers. This is studied by building and testing a working prototype of a Live Programming tool.

While the core of this work will be mainly centred on technical aspects on how to build a working system, the project will also consider usability aspects so that the approach for inexperienced programmers will result as as simple as possible.

1.5 Method

To research the questions stated in this study several different methods have been applied. A theoretical study has been made with the purpose to identify important aspects of Live Programming environments. A survey of previous research has been made with this purpose. An empirical survey of existing Live Programming tools was performed to further explore the field and gain experience in the area.

To test the feasibility of Live Programming for mobile devices a prototype was designed and implemented. Finally, and empirical study with users was made. The purpose of this study was to gather feedback and investigate real-world use of the prototype. This study was conducted using an internet based questionnaire. The questionnaire contained both closed and opened questions related to the prototype and to Live Programming in general (see Appendix A).

1.6 Delimitations

This project is about testing if Live Programming is suitable for mobile development. For this purpose, a prototype of a web programming environment and a mobile container for showing the result of the running code have been developed. The system makes use of the HTML5 and JavaScript programming languages and existing tools such as the MoSync SDK for programming hybrid mobile applications. In this project new languages or extensions to enable Live Programming have not been built. Although this project aims to give support for Live Programming through text based programming, custom graphical tools have been created to enable and enhance the Live Programming experience. Focus has been put on the communication of the system components to make it possible for the remote execution of program code on different devices. The tool developed in this project is for experimentation and users cannot, at this stage, pack and deploy their applications.

Concerning the Live Programming experience, the environment gives the users some tools for controlling the execution of a program. State can be preserved with some limitations depending on which action users take, so the produced mobile applications do not always reflect the program code. To achieve this, more elaborate parsing of the program code would have

been required.

This environment is a starting point for future work in this area, giving the developers the possibility to, for example, share their applications online. The web application has been implemented to run on a local server, without taking into account multiple users connecting to it simultaneously. No real-time collaboration as in e.g. Google Docs has been implemented, but that could also be included in future work.

Although Live Programming can encourage novice developers to learn to program and to possibly master new concepts quicker, the main focus of this project was not to create an educational environment. The focus of this project was instead to provide an easy way to test Live Programming. The scope of this study is not to craft a new tool for creating full featured mobile applications, but to provide a proof-of-concept system to see if Live Programming can coexist with mobile development.

Chapter 2

Live Programming

2.1 Introduction

Live Programming goes back to the earliest days of computing, and it has been used in different contexts, from interactive interpreters to live art performances. Live Programming was adopted by early development environments that had the common property to be "alive". This means that while coding, the user has instantaneous feedback and evaluation of the written code. Consequently, this programming style has the potential to reduce the programmer's load on the memory when reasoning about a problem, empowering both experienced and novice programmers.

The term Live Programming was first coined by Christopher Hancock in 2003 [26]. In his dissertation, Hancock presents a live programming environment for teaching children to program robots. Flogo, the programming language used in the environment, represents the programs in terms of data flow, enabling liveness in the program. The code displays its own state as it runs. Two versions of Flogo are presented: Flogo I uses graphic interfaces to display

the data flow of the program, while in Flogo II the liveness is embedded in the program code itself. In Flogo II programs can be edited as they run, and the programmer can at any time decide what parts of the programs should be live.

An important aspect related to Live Programming and software development is tinkering. Turkle introduces the term tinkering to describe the process of exploratory process of programming [56]. Via Live Programming, a tinkerer can have a tool that gives support for exploratory programming using basic building blocks. Both experienced developers and novice programmers can take advantage of programming by tinkering. The former can reach a goal more rapidly via experimenting, while the latter can be able to develop working software from a total lack or incomplete knowledge of a programming environment.

Before Hancock, Tanimoto also talks about liveness of a program, specifying four degrees of liveness that are measured according to how much live feedback they present to the programmer [55]. The first level in the scale is informative, defined as the visual representation of the program, used as an aid to understanding the document. In this level no semantic feedback is provided to the user. The second level, however, provides semantic feedback upon a request from the user. Typing an expression in an interpreter exemplifies level two of liveness. At level three, a representation of a program must provide a direct feedback of the program as new changes are made when editing it. At this level the system will execute or re-execute the programs whenever the user makes any changes. Once the user stops typing, the system becomes idle. Finally, the fourth level of liveness means that the system is continually active, updating the display according to the processed streams of data specified by the program.

What kind of properties are required for a programming language to sup-

port Live Programming? A classification of programming languages can be defined as compiled programming languages and interpreted programming languages. The main difference between the two groups is that compiled programming languages need to be translated to a machine level to be executed, while interpreted ones can be executed without compilation. The interpreter executes the program statement by statement, and the absence of the extra compilation step allows interpreted languages to evolve during runtime and new code can be injected in the program in execution.

In an environment that supports it, interpreted languages enable Live Programming, where the program can be evaluated as the programmer changes parts of it. For example, in languages like Lisp [49], Smalltalk [22] and Erlang [5], a program can be modified without being restarted by injecting new code. In any case, the programmer has to re-execute the program to see the changes. Similarly, a developer could use a REPL [59] (read-eval-print-loop) to experiment with program statements.

In Apple's Quartz Composer [4], a visual programming language for building animations, components can be connected showing a reactive data-flow depending on the connection scheme. Changing a connection, the result of the the new animation is immediately observed.

Compared to textual languages, visual languages have many benefits [25], among others being user friendly, more abstract and thus easier to understand. Studies have also showed that the human mind is optimised for processing shapes instead of words. However, when reasoning about a program, its representation seems to be way less important compared to the structure of the information [25], making textual languages hardly replaceable. Moreover, text can be written more quickly than placing graphical components and connecting them manually [41]. Also, when building large programs, textual languages do not have to deal with the scaling-up problem [11].

The following list summarises the benefits of Live Programming to the process of developing software:

- Developers have to store less information in their brain, since the output can be seen directly making small increments in the software. In this way there is less probability that a programmer updates a big portion of the code without testing it early, avoiding the risk of having to rewrite large chunks of code.
- Live Programming speeds up the development cycle. Quick feedback allows for a flow of creativity, since new ideas can be tested and changes are visible instantaneously;
- Live Programming, and as mentioned before, programming by tinkering, encourages exploratory programming, making it easier for novice programmers to experiment. Experienced developers can also take advantage of it for quickly experimenting with a new platform or a new programming language;
- Instant error detection, since the code is executed at each change. This prevents errors to accumulate as the programmer edits the code many times before testing it;
- A positive effect of Live Programming can also be that the developer understands better what a certain piece of code does, increasing the programmer's knowledge of the code;
- As noted in Flogo II, Live Programming can enhance not just the readability of a program text, but also the readability of its execution [26](p.103).

2.2 Early live environments

Live Programming is an idea included in early programming languages and environments. In Smalltalk [22] everything is alive, i.e. the programmer can change the running program code without interrupting its execution. Code becomes mutable data and can therefore be inspected in real time. In Smalltalk the results are tangible, giving immediate feedback and reward [21]. In *Refactoring steroids* [51], a live refactoring of the game Asteroids is performed while the game is running, increasing the programmer's productivity and maintaining the flow. This behaviour is present in other environments similar to Smalltalk, such as Self. One of Self's strengths is its user interface Morphic [37], built to support directness and liveness. Self's UI is direct in the sense that the programmer can manipulate and alter the components by pointing at their graphical representation. It is also live in the sense that every modification to an object is immediately evaluated, making components reactive and responsive which allows information to be continuously updated. Another highly interactive language that enables live coding is LISP [49]. LISP supports functionality for inspecting and updating the code without restarting a running program. This is achieved by either using a REPL [59], or a popular interactive programming environment called SLIME [23]. Because of this properties, LISP is often associated with the term incremental programming, where the programmer does not just program but grows the code. In HyperCard [3], the state of any object is editable directly for instant results.

2.3 Recent live environment

2.3.1 Art

Recently, Live Programming has gained different meanings and has been used in different contexts. The term can refer to a tutor presenting a program to an audience and writing code from scratch while explaining the meaning of the program. Another context is within music composition and arts, where a programmer manipulates program code by improvisation to compose and generate music and animations from various algorithms. The latter is an area where Live Programming has been developed and used extensively in the recent years. By using interpreted programming languages, artists and musicians can practice Live Coding on stage. Impromptu [54], a Scheme based environment, and Overtone [48], a Clojure based environment, are examples of interactive live environments for music composition. Sorensen [53] and Aaron [1] both show and give examples of the use of the two environments respectively. A Live Programming environment that targets more specifically graphics and animations is GLSL Sandbox [12], a project for programmers who want to become familiar with OpenGL. More ideas and environments can be explored on TopLap [38], an organisation emerged in 2004 that has the aim to gather projects about Live Programming and to promote live coding. TOPLAP focuses mainly on electronic music and video, encouraging people to experiment with code projection. Research in the field of Live Programming has also been done by Blackwell [6], proposing that general programming research can benefit from studying new and unusual programming contexts such as Live Coding in the music domain. His claim is based on the fact that many advances within programming languages often began by considering new groups of users. Addressing untraditional needs within programming, new creative solutions can be found. This is the case of, for

example, Smalltalk [34], ToonTalk [33], or the invention of spreadsheets [43]. Blackwell also compares the practice of programming to music composition by emphasising the importance of the feedback loop when realising new ideas. When composing a new piece, musicians have a tight feedback loop from the instrument they are using by hearing the results of their work. Programmers can in the same way make use of dynamic programming languages to achieve the same kind of feedback described as progressive evaluation.

2.3.2 Educational

A lot of research has been done in the field of programming, and more specifically on how to educate and introduce programming to students and people without expertise. Live Programming can play an important role when introducing programming to novices, and different platforms with Live Programming support have been developed with the goal to teach programming principles from the ground up. One of the actors that has put a lot of effort into the educational aspect of programming is the not-for-profit organisation Khan Academy [2]. Khan Academy's computer science platform consists of a Live Programming environment for exploring and experimenting with graphics to learn programming principles. This tool is based on an implementation of the Processing language [20] for the graphics in JavaScript. JavaScript, being both interpreted and highly dynamic, allows for experimenting and for modifying a program during its execution.

2.3.3 Experimental IDE

More advanced prototypes and tools rich of features have also been developed to support and demonstrate Live Programming. Among others, LightTable

[24] is an experimental project based on the idea of offering the developers a real work surface to code on and placing the information where it is needed most. The principles of LightTable are a simple way of showing documentation and a quick responsiveness of the environment for direct result when changing the code. Two other interactive environments based on Smalltalk are Lively Kernel [31] and Amber [42]. The former is a complete platform for creating web applications with support for graphics, networking and development tools. In Lively everything is alive, letting the user extend and alter anything in the environment with immediate result, enhancing creativity. Amber is an environment built for the web that supports JavaScript through Smalltalk syntax. The web browser becomes in this case an incremental development environment with Smalltalk integration.

2.3.4 Experimental Web Design

Another area where Live Programming has flourished is web design and web development. Many of the environments that support Live Programming have been developed as web applications, thanks to the fact that the standard programming language of web browsers happens to be JavaScript. Web developers, but also programmers that need to experiment with JavaScript, can program directly in the web browser, lowering the barrier to start programming. JS Bin [50] is an open source collaborative web development debugging tool, where users can write code and get web pages rendered in real time. Other similar environments based on JavaScript that support multiple JavaScript libraries and Live Programming are livecoding [19], Tributary [32] and Plunker [17]. A language built and specifically designed for Live Coding is Circa [18], used in the graphical code editor Improv. The peculiarities of this language are state preservation, introspection and the ability to code both via text and via visual programming.

2.4 MoSync and Mobile Development

MoSync is a company based in Stockholm. MoSync's vision is to create development tools to reduce the burden of building mobile applications. These tools unify the process of creating mobile applications by offering the developers a platform with a unique code base, which is then translated to native applications for different mobile operative systems and devices present on today's market. The goal of MoSync is to enable developers to spend the majority of their time focusing on development, instead of adapting to a world of heterogeneous platforms.

The target users of MoSync are both developers that already have a basic understanding of programming principles and novice programmers. MoSync provides a range of tools to give support for different skill levels when programmers build mobile applications.

2.4.1 Reload

One of MoSync's flagship products is Reload. Reload is a platform that combines the dynamic nature of JavaScript and the MoSync SDK letting the user develop native mobile applications in a container application with a unique code base. A developer can edit code in JavaScript, HTML and CSS and see the result on multiple devices that run the client version of the Reload platform. The process of edit-build-deploy-test is shortened dramatically, as the program can be uploaded to the container app and visualised immediately. Reload supports both HTML elements but also native UI elements of specific devices by using a JavaScript API that can access the different services present on a device. Reload was an inspiration for parts of this thesis project, since some of its functionality was integrated in the live environment.

Chapter 3

System Design

In the following chapter the system functionality and the system architecture of the prototype build during this project are presented. They are the result of experiments and requirements to enable Live Programming for mobile development.

3.1 Functionality

The built system should provide functionality for Live Programming, in a way that is accessible and easy to learn. This list describes the functionality the system should support:

- **System support:** The system aims to be as easy to use as possible, so the programming environment should be a web application accessible from any web browser on as many machines as possible. This approach eliminates incompatibilities between different platforms and operative systems, since the system runs in a normal web browser. The choice

for implementing a web solution comes also from the need to make the environment as flexible as possible so that it is easy to add features for improving the Live Programming experience.

- **Code editing:** Since the programming environment runs in a web browser, there should be support for writing code in a code editor with some of the essential functionality that a modern code editor should support (auto-completion, syntax highlighting, etc.);
- **File system:** There should be support for a file system that lets the users store the work done and open previous projects;
- **Error detection:** The environment should provide support for immediate syntax error detection giving the user some kind of visual feedback, incrementing the development speed by eliminating obvious mistakes in the code;
- **Graphical aids:** The environment should provide some form of graphical components to help developers implementing new ideas and achieve the results quickly while writing code;
- **Auto-completion:** The environment should provide auto-completion facilities to help finding language constructors quickly without the need to remember all the keywords that are present in the language or in the system;
- **Code execution control:** It should be possible to selectively evaluate portions of the code as it is done in Smalltalk, giving the users the ability to experiment with the code execution and program behaviour in a flexible way;
- **Live code editing:** The system should provide a way to execute code on a mobile device in real-time while typing in the code editors;

- **MoSync support:** The environment should provide support for the MoSync JavaScript SDK for creating native mobile applications;
- **Documentation:** The environment should provide easy to access documentation and possibly a tight integration between the code and its documentation;
- **Code results:** The system should include a mobile application that can connect to the system to display the program representation of the code developed in the coding environment;
- **Native applications:** There should be support for accessing the mobile device services as when developing a native application, and also the option to create and use native components on different types of mobile devices;
- **Mobile OS support:** The system must target and work with the three main operative systems iOS, Android and Windows Phone.

In Chapter 4 the implementation of these functionality will be described and discussed.

3.2 Users

The intended target audience for the system are both novice users and advanced users. Familiarity with the MoSync JavaScript SDK is an advantage, but not necessary. However, basic knowledge of web technologies such as HTML5 and JavaScript is required. Anyone with basic understanding of programming should then be able to experiment and quickly create simple prototypes of mobile applications.

3.3 Architecture

The main three parts of the prototype for achieving Live Programming on mobile devices are a web server, a web application and a mobile application. The idea is very basic: at every code update in the web editor, the changes are pushed to the mobile device connected to the server, and the client mobile application executes the received code and displays the result. Figure 3.1 shows a picture of the system architecture. The three main components of the system are described below:

- **Server:** the web server is responsible for serving the web application to one or more clients that want to use the web interface, and for coordinating the communication between the web user interface and the mobile application on the mobile device. This means that when the web application wants to send and receive data to and from the mobile device, the communication has to go via the server. This is due to the nature of the client-server architecture, where one server usually serves many clients.

- **Web application:** the web application is the main user interface where a developer can write HTML5 or JavaScript code to create mobile applications. Depending on the scenario, different actions can be performed by the user to send the code to the mobile target devices. Code can be sent by manually pressing a button, causing the web application to send the whole program and hence creating a new runtime on the mobile device. In this way the application acts as if it was closed and restarted. Another option is to selectively evaluate pieces of arbitrary code, while the third option is to enable Live Programming for seeing the results of the program code as new code is typed in the code editors. The web application should also provide a way to access some

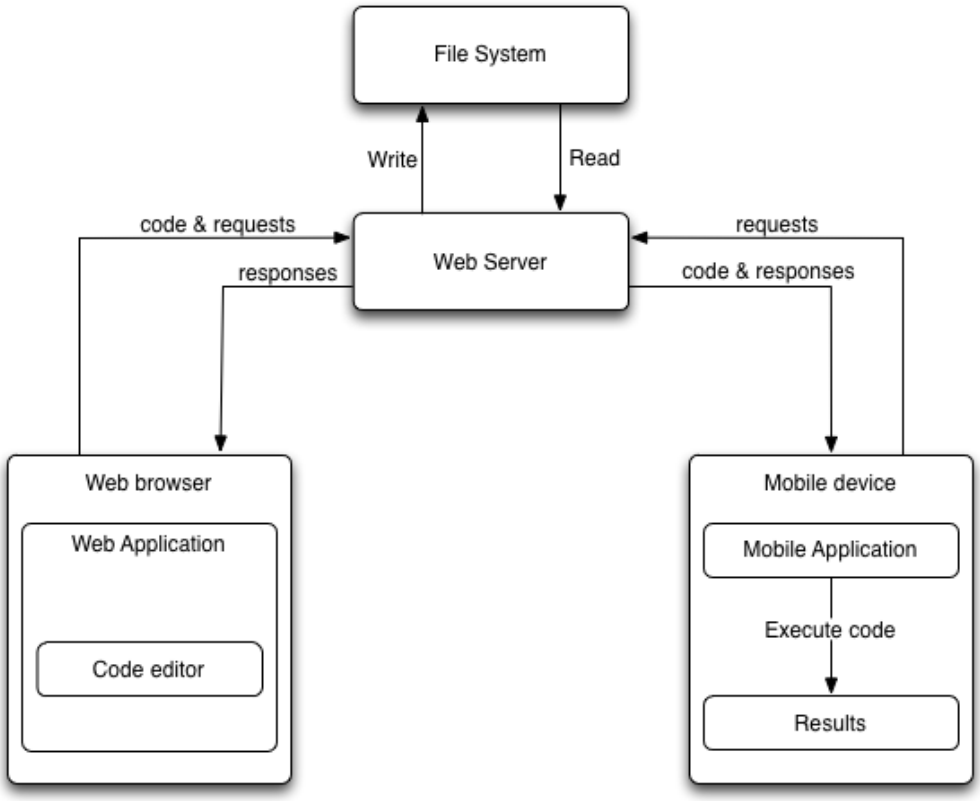

Figure 3.1: System architecture

kind of file system, so that users can load and save files and projects.

- **Mobile application:** the mobile application is responsible for executing the code it receives from the web application via the server and shows the results of the program that is being developed. This application is able to connect to the server and to listen to incoming messages containing new code to execute. The only component that this application displays is a web view, which is responsible for executing the new received code and displaying the results of the execution. Users can either create HTML5 components, Native UI components with the support of the MoSync infrastructure, or a combination of both.

3.4 Editor/server - client communication

The web application and the mobile application communicate with each other through the server, and exchange data with the server using both asynchronous communication and push technology [35]. Asynchronous communication allows for a more responsive web application, while push technology allows the server to send data to a client without first waiting for a request. In this way, the web application can send code to the server with an asynchronous request, and the server can push the message to the mobile device that is listening to new incoming messages.

Chapter 4

Implementation

In this chapter the implementation of the prototype for enabling Live Programming on a mobile device is described. First, the mobile application is described, then the web application and server functionality are presented. The source code of the mobile application and the web application/server can be found on GitHub [9] [7]. A brief tutorial of how to use the prototype can be watched in a demonstration video [8].

4.1 Introduction

To enable Live Programming on mobile devices, HTML5 and JavaScript were chosen as main languages. In most implementations JavaScript is an interpreted language, and thanks to this property, code can be reloaded and injected in the current runtime directly without the need of compilation. A way to exploit this behaviour is to use the language constructor *eval*, a function that can execute an arbitrary string as if it were a piece of JavaScript code. Another strong factor behind the choice of HTML5 and JavaScript

as language development of mobile applications is that those languages are already used and integrated in Wormhole, a JavaScript library developed by MoSync [39]. Hence, the use of HTML5 and JavaScript enables a tight integration with the MoSync infrastructure, that allows for accessing native services and native graphical components on a mobile device exclusively via JavaScript.

4.2 MoLive - Basic live app

One of the central components that are part of this system is a mobile application developed with the MoSync SDK environment. The MoSync SDK is an IDE that lets users build mobile applications for different mobile devices and different operative systems with only one code base. Applications can be written in pure C/C++, in HTML5/JavaScript and/or a combination of both. The IDE will then compile the source code for the relative target platforms. The MoSync SDK exposes a set of libraries and APIs for accessing native services and components on a broad range of mobile devices. Hence, an application built with this tool can be developed in the aforementioned languages and gain access to the device services and native graphic components independently of what language is used. The MoLive mobile application is built with this tool.

To enable HTML5 and JavaScript to access native behaviours of a mobile device, a straightforward solution for executing these languages is to create an instance of a web view into MoLive. Web views are a type of component that are available on most mobile operative systems. They allow displaying web content and most importantly they are capable of rendering HTML5 and executing JavaScript. In this way, HTML5 and JavaScript code can be sent to MoLive so that the web view can render web pages and at the same access

native services by executing JavaScript code. A developer is able to create a mobile application so called hybrid, by using both native components of the device such as the camera, and by executing HTML5 and JavaScript code in a web view, a tool that is typically used for presenting web content. MoLive can connect to a server by inserting the server address in a text field, and will then listen to incoming messages transporting code to execute. When executed, HTML5 and JavaScript code is able to access native services on the mobile device by using the JavaScript Wormhole library developed by MoSync. Wormhole can communicate with the C/C++ APIs runtime of a device by making bridge calls that will be able to access the native APIs of a specific platform and listening for responses. In this way, a programmer can exclusively code in JavaScript and create rich native user interfaces, having at the same time all the features of this language at her disposal. In summary, a user developing native applications with web technologies does not have to worry about the details of how a specific API works. Figures 4.1 and 4.2 show the architecture and a screenshot of the MoLive mobile application.

4.3 Server

The MoLive server is a web server written in NodeJS [30] and uses socket.IO [46] to coordinate the communication between the web application and the mobile application. Socket.IO is a well known application framework that integrates well with NodeJS, enabling real time web applications in all browsers and mobile devices by hiding the transport mechanism used in the communication for the developer. It primarily uses web sockets if supported by the platform, or other different technologies to support push technology. In MoLive sockets are for example used to alert a mobile device that some code have been updated and needs to be run in the web view. An example of how

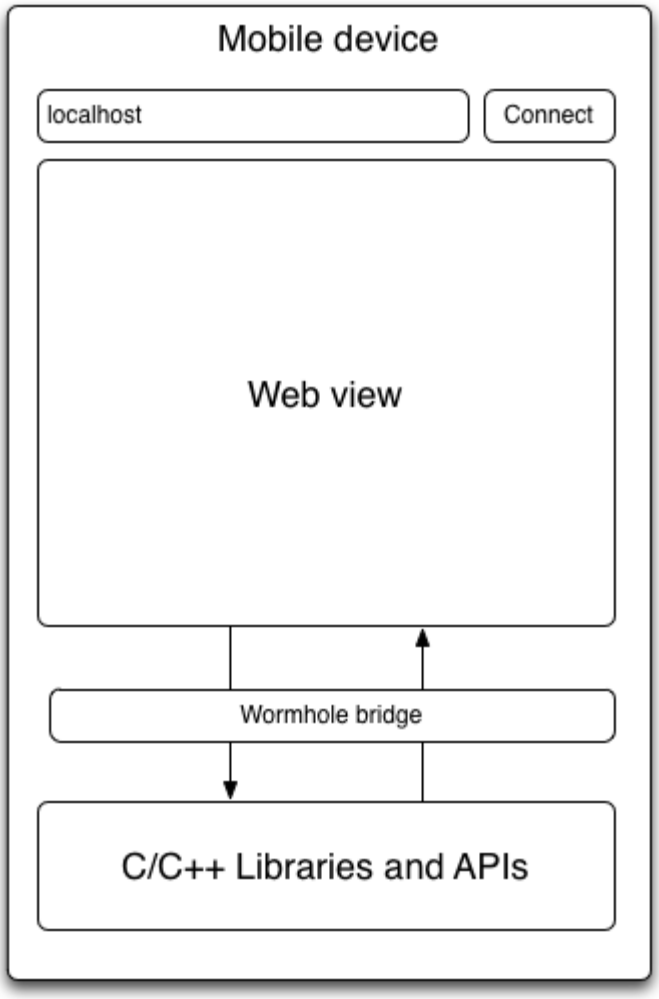

Figure 4.1: Mobile Application Architecture.

Figure 4.2: Mobile Application.

Socket.IO works is shown in listing 4.1.

```
1  io.sockets.on('connection', function(socket) {
2
3    // Add a client to a room
4    socket.on('room', function(room) {
5        socket.join(room);
6        console.log('A client has joined: ' + room);
7    });
8
9    // Send the html code to the mobile room
10   socket.on('html', function(code) {
11     io.sockets.in('mobile').emit('html', code);
12   });
13
14   // Send the JavaScript code to the mobile room
15   socket.on('javascript', function(code) {
16     io.sockets.in('mobile').emit('javascript', code);
17   });
18 }
```

Listing 4.1: Socket.IO example from the server code.

First, the server listens and instantiates a connection with the incoming clients that want to establish a connection. Then, every time a message is received on a socket, the server tries to find a listener for the matching message, executing the body declared for the relative message. The code above shows also a functionality of Socket.IO, namely the possibility to select which connected clients will receive a particular message. This is achieved by grouping clients in rooms, making it possible to send a message to a set of clients in one function call.

The server also provides support for communicating with Gist [29], a service provided by GitHub that enables users to save and retrieve files. In this

way, users can have a file system in the cloud, making the projects available anywhere and opening possibilities like code sharing and social coding [15].

4.4 Communication

Most of the communication between the web application, the server and the connected devices is done by a combination of asynchronous pull requests and web socket messages. Which technology is used depends on the scenario and on the type of functionality and actions to be performed. For example, asynchronous pull requests (AJAX requests) are made when a new file is created in a Gist project through the web interface. In this case, the sender of the request is also the receiver, so the web client will send a request to create a a new file to the server, the server will contact the GitHub API to create the file, and on successful response or a timeout, the server will reply to the web client with a positive or negative response. A different scenario is when a client has to be informed that something has happened, without having made a request for it. This is the case of a mobile client that keeps an open channel that listens to incoming messages, such as new code to execute.

4.5 Web application

The web application is the main interface for the users, and it is illustrated in Figure 4.3. The web application is the place where one writes code to be executed on the connected mobile devices. To do this, the web interface provides two code editors, one for coding HTML5 and one for coding JavaScript. These editors implement a framework called CodeMirror [28], a JavaScript plugin that extends the HTML5 text area and adds functionality

to it as in a modern code editor. Such features include for example syntax highlighting and smart indentation, to help developers in the coding process. A lot of work has been put into the code editors, by customising existing functionality of CodeMirror and by adding new ones that could improve the Live Programming user experience. The JavaScript editor provides feedback for early error detection by linting the source code and showing widgets that warn the user and indicate where the error was found. Linting is the process of static checking the source code of a program to find potential errors. This is shown in Figure 4.4 and Figure 4.5.

Another programming aid that the code editors provide is autocompletion. Autocompletion lets users find what they are looking for quickly, by showing a list of suggestions as code instructions are typed. In this system, autocompletion is available on both editors. In the JavaScript editor this functionality picks up new tokens typed by the user dynamically, and shows different suggestions depending on the type of the token. In the HTML editor, autocompletion is context-aware, meaning that different suggestions are showed to the user depending on what element or widget the user is creating. A list of all the attributes a widget can have is also displayed. This is exemplified in Figure 4.6.

A custom number slider and colour picker were added to the code editors, making it easy to select colours and values. These visual components are used for updating code by dragging an indicator with the mouse for changing tokens in the code. This is achieved by listening to the caret activity in each code editor and pattern matching each token that the caret passes to find relevant matches. The number slider component is showed when a token contains or is equal to a number, while the colour picker is showed when the caret is on a token containing an hexadecimal colour representation. These widgets can help the developer to re-position a component on a user

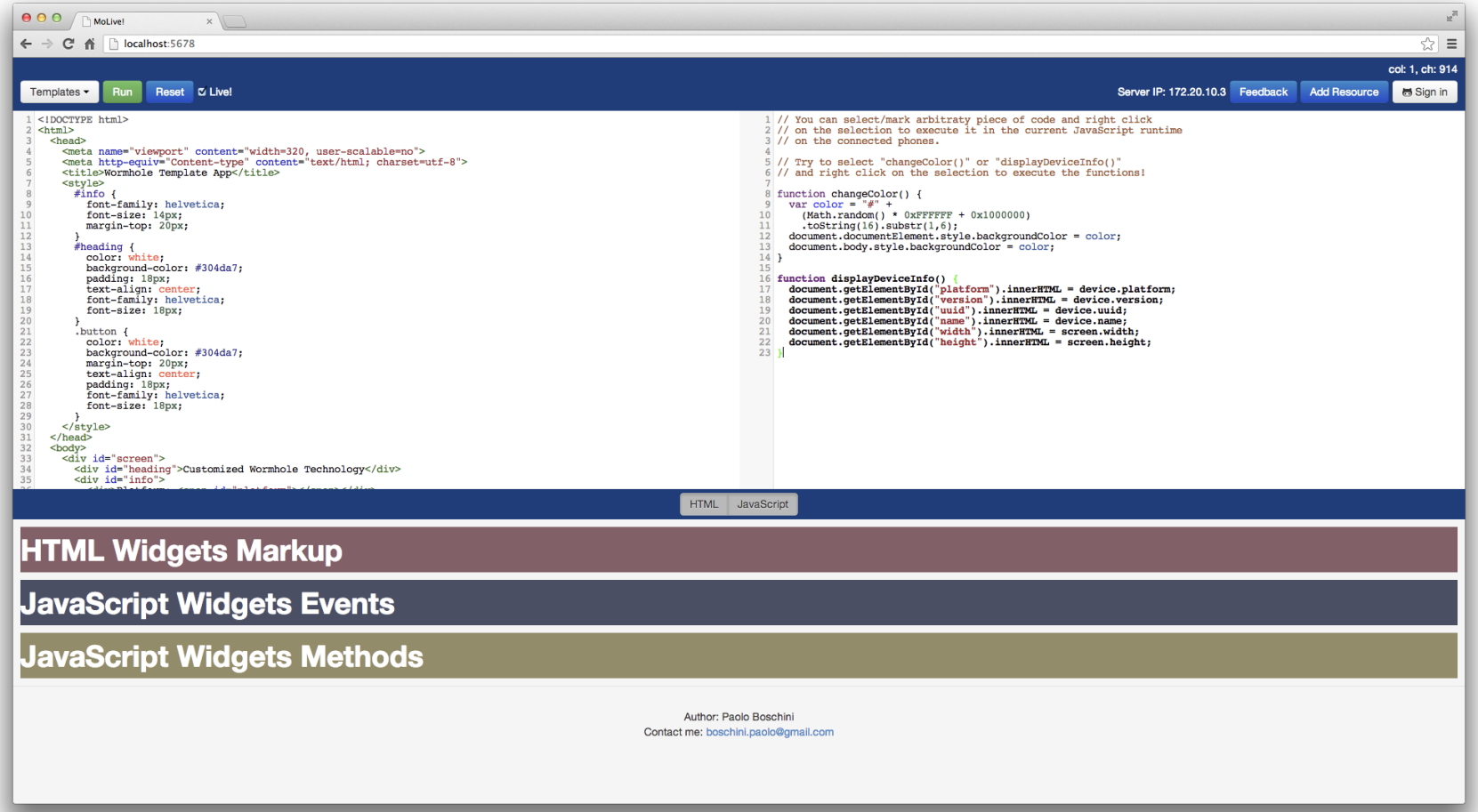

Figure 4.3: Web Application in live mode. The header of the web application contains buttons for controlling the code execution, information about the server address and the status of GitHub authentication. In the middle of the page are the code editors, while on the bottom is the documentation for native widgets.

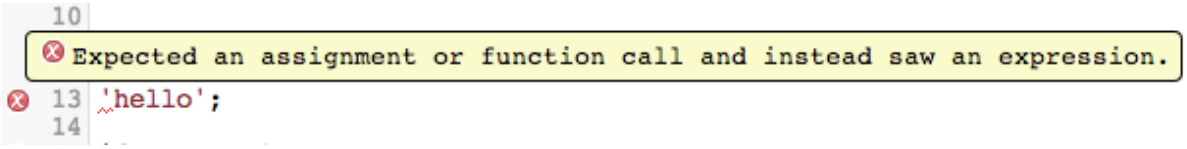

Figure 4.4: Feedback on errors.

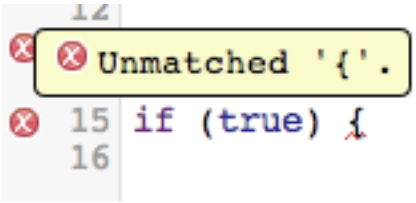

Figure 4.5: Feedback on errors.

Figure 4.6: Autocompletion in action when creating a Native UI component using HTML5.

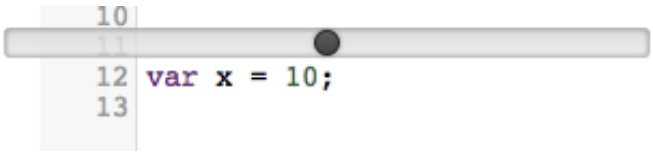

Figure 4.7: Number picker.

Figure 4.8: Colour picker.

interface quickly, change the size of a picture or modify colours in an interface. By dragging with the mouse the user can see what happens in the mobile application continuously, for a better control of the final result. These widgets are shown in Figure 4.7 and in figure 4.8.

In this prototype, the textual code itself does not contain any visual feedbacks, but users can see the changes made to the code instantaneously reflected on the connected mobile phones. A central aspect when building a mobile application is that the UI is continuously visible and updated during the development process.

4.5.1 Executing code

In addition to the aforementioned visual aids, the web interface lets the developer control how code is executed on the target devices. Before executing any user actions, the system first performs a check on the JavaScript code through the JavaScript linter JSHint [36], a tool used to see if the program contains errors. If no errors are detected, the system will execute the action taken by the user. The most simple way to execute code is to click on the run button. Using this method, the system will fetch both the HTML5 markup and JavaScript code and will send it to the mobiles to be executed. This procedure will re-run the program from the beginning, resetting the state of the current runtime on the connected mobiles. Another way to execute code is to selectively run it. This involves selecting the portion of text code that one wants to execute and clicking on the execute code button activated by right clicking on the portion of the selected code. This action, illustrated in Figure 4.9 and heavily inspired by Smalltalk, will solely send the portion of the selected code to the targets and the received code will be injected in the current runtime. This implies that the application is not restarted, but will instead be updated and changed live. A simple example would be to change a function in the JavaScript code, select it, and execute it to redefine it in the runtime of the mobiles. To note is that the function will not be executed if the program calls it somewhere in the rest of the code, since the user has not explicitly called the function, but only redeclared it. This action can be only performed in the JavaScript editor, since this language is dynamic and can execute arbitrary pieces of code with the *eval* code instruction. At this stage of the system, the HTML5 code is instead always re-executed in its whole. HTML5 is a declarative language, and injection of portions of code would have required a more extensive work not possible for this project. The last approach to execute code is to manually activate the live checkbox situated

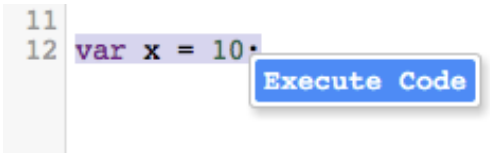

Figure 4.9: Selectively run a portion of code.

on the header of the web application document. Activating the live option, anytime a developer strokes on the keyboard, the system will send a message to the connected devices with some code, depending on in which editor the changes were made. If the user edits the HTML5 code, the application is restarted with the new HTML5 and JavaScript code, while if JavaScript was edited, only the current edited top-level expression is executed immediately in the runtime of the mobile device. To give a visual feedback of what exactly is being sent to the mobiles, the relevant portion of the code is highlighted with a bolder font, as shown in Figure 4.10. This is done by parsing the JavaScript code with Acorn [27], a JavaScript parser written in JavaScript. The parser takes JavaScript code as input, and returns among other useful information the top level expression of a program.

4.5.2 File System

Another central functionality that the web application provides is the interaction with menu-like graphic components that let the user make use of GitHub's Gists as a file system. This functionality lets the user manage projects and files remotely in an easy and intuitive way. Files can be grouped in projects and can be created, saved and easily recalled by the system. The integration with GitHub was possible by making use of *everyauth* [40] and *node-github* [16], two plugins for managing GitHub authentication and interaction with the GitHub API interface. In addition to Gist support, the

```
 8 function changeColor() {
 9   var color = "#" +
10     (Math.random() * 0xFFFFFF + 0x1000000)
11     .toString(16).substr(1,6);
12   document.documentElement.style.backgroundColor = color;
13   document.body.style.backgroundColor = color;
14 }
15
16 function displayDeviceInfo() {
17   document.getElementById("platform").innerHTML = device.platform;
18   document.getElementById("version").innerHTML = device.version;
19   document.getElementById("uuid").innerHTML = device.uuid;
20   document.getElementById("name").innerHTML = device.name;
21   document.getElementById("width").innerHTML = screen.width;
22   document.getElementById("height").innerHTML = screen.height;
23 }
```

Figure 4.10: Live coding in the JavaScript editor. The code within the caret is highlighted and re-executed every time a change is made to that expression.

prototype provides a simplistic template system for recalling ready-to-use projects for getting started with mobile development. The interface for this functionality is shown in Figure 4.11. A GitHub icon and a user name are shown indicating that the user is authenticated on GitHub and can read and write Gist projects and files. Upon a selection of a Gist project, a list of HTML5 and JavaScript files is downloaded and displayed into two drop-down menus for further selection.

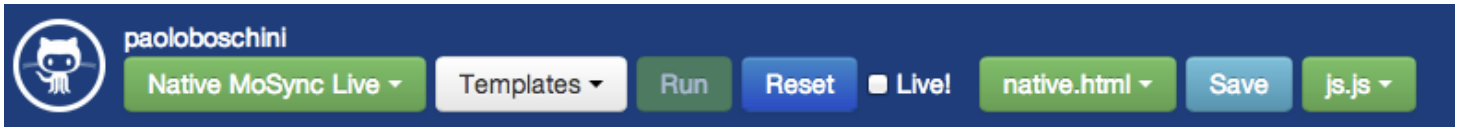

Figure 4.11: Gist support.

4.6 Discussion

One of the major challenges during the implementation of this prototype was to define how the system would implement Live Programming and how the user would perceive instant feedback from the system. A lot of experimentation was done to understand how JavaScript could be used to provide code injection on the connected devices, and especially to understand if there was a way to always have the text code representation in the web application to match the output on the mobiles. This turned out to be very complicated to achieve, since the state of an executing program has to be preserved when changing code in realtime.

Preserving the runtime state of a program while Live Programming is difficult. While editing a piece of code, the desirable effect is not to re-execute the whole program, but to only execute the portion of the code that was changed. In this way, the current state of the program such as variables stored in memory, is not altered. This could be done by keeping track of all the objects (such as variables and functions) currently loaded in memory, and update their value when the code is updated. However, if we consider the code in listing 4.2, a change to x also affects y, making it difficult to find the objects dependency chain.

```
1   var x = 1;
2   var y = 2;
3   y = x;
```

Listing 4.2: Variable assignment.

Since JavaScript links objects by reference, a workaround to this issue would be to preprocess the source code and create copies of all variables and functions and embed them in objects, so that a change to a variable or to a function will be propagated to all other linking references. The implemen-

tation for listing 4.2 would create a dummy object for x and one for y, so that a change to the value of x will be propagated to y. The implementation developed for the Khan Academy platform uses a similar approach [47], where all the global objects (variables and functions) are serialised as strings in a global container object, and evaluated in a temporary runtime at each code change. The new values are then compared to the old values, and what differs is updated accordingly.

Two alternative solutions were proposed in this prototype, namely to either manually select a portion of code to execute as it is done in Smalltalk, or to inject the top-level expression that is being edited. These two simplifications were adopted due to the difficulty of both updating a program live, and keeping its state at the same time.

In this prototype, *eval* is used as a mean to execute code and inject it in a program runtime. According to common JavaScript guidelines, usage of *eval* should be avoided since it can open for security issues if the source code that is being executed is not a trusted one [60] [14] [13]. In this prototype, the source of the code is known, and use of *eval* opens up for live code injection that allows for Live Programming.

Languages such as JavaScript are not designed for Live Programming, so changing arbitrary code and injecting it can be difficult to make it work the way we intend to. Functions that normally work with the standard JavaScript API behaves differently when programming live. An example is *setInterval(code, millisec)*, a method that calls a function or evaluates an expression at specified intervals. Editing such function live would trigger new instances of intervals, breaking its original functionality. A workaround would be to offer the developer an ad hoc *setInterval(timerId, fun, ms)*, such that it takes a timer id, the function to run and the interval in milliseconds. In this way the new API can stop the old timer and restart a new one with

the new code. An example of this implementation is shown in listing 4.3.

```
1 var MyTimers = {};
2
3 function mySetInterval(timerId, fun, ms) {
4   if (MyTimers[timerId]) {
5     // stop this timer
6   }
7
8   MyTimers[timerId] = setInterval(fun, ms);
9 }
```

Listing 4.3: New setTimer API example.

In this case, the design of the language API influences Live Programming. For this reason, the standard implementation of JavaScript becomes limited for this kind of programming. Hence, a more interesting approach would be to design a subset of JavaScript or an entirely new language from scratch that specifically targets Live Programming.

Another problem is how the program should behave when functions are edited live. An approach would be to redefine the new function in the current runtime, or re-run the whole program, causing the state to be lost. Alternatively, together with the redefinition of the function, the system could execute the calls to that function in the rest of the program. This opens a number of issues in the execution flow of the program, such as calls that depend on conditional statements. This prototype provided a few simple solutions to deal with this problem, letting the user chose what to run, or giving visual feedback of what is being re-evaluated in the runtime of an application.

Another challenge was to decide how to go about the execution of HTML5 markup code. As it is now, there is no way to automatically add arbitrary HTML5 code to an existing web page without traversing the nodes of the

document and manually changing the nodes that need to be updated. Due to time limitations this was not possible to implement, so at the current state the system simply rewrites the whole HTML5 document as the user changes it. Moreover, since a requirement of the system is to support the creation of native UI widgets using solely HTML5 elements and attributes, this would have involved a thorough extension of the MoSync JavaScript Wormhole library. Instead, when updating HTML5 code, the system uses the document.write(html) JavaScript instruction that replaces the current document with the new passed one.

4.7 Alternative implementations

An alternative design experimented during this project was the use of PubNub [45], a service for real-time messaging. PubNub is a publish-subscribe network where users can send messages in real-time. An implementation of this system was made using PubNub, using the service to send code as messages between the web application and the mobile application. However, this implementation turned out to be very inflexible, due to the size limitation of each PubNub message. PubNub's great network performance relies on compressed messages that are smaller than a single IP datagram, making the communication very fast [44]. This limitation was a clear barrier for this system, since pieces of code longer than 1800 characters would not reach the destination point. Figure 4.12 illustrates the the system using NubNub.

Another alternative solution for this system was to use already implemented pastebins (web application coding playgrounds) such as JS Bin [50], capable of rendering HTML5 and JavaScript to an arbitrary URL in real-time, so that changes made to code in a web browser can be shown in any other web browser pointing to the same URL. Moreover, at each code change the out-

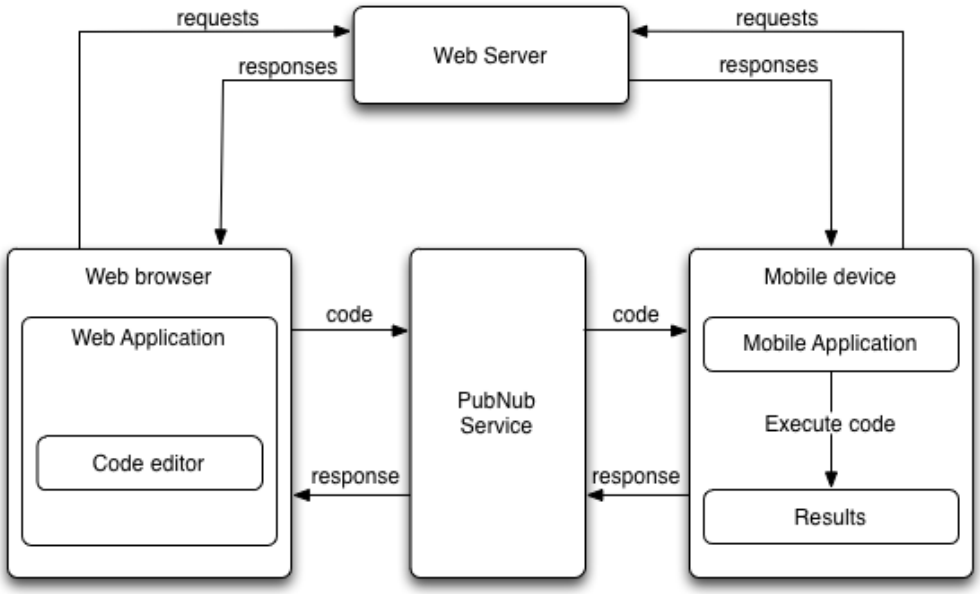

Figure 4.12: System using PubNub service.

put is immediately updated without the need of refreshing the page. Using this tool would eliminate the need for a web server, since JS Bin is already deployed as a web service, and the coding would be done in its editors and pushed automatically to the mobile devices. A downside with this approach would be the lack of flexibility for the user, since the system would be dependent on an external service and no functionality could be added to the system. Figure 4.13 illustrates the the system using JSBin.

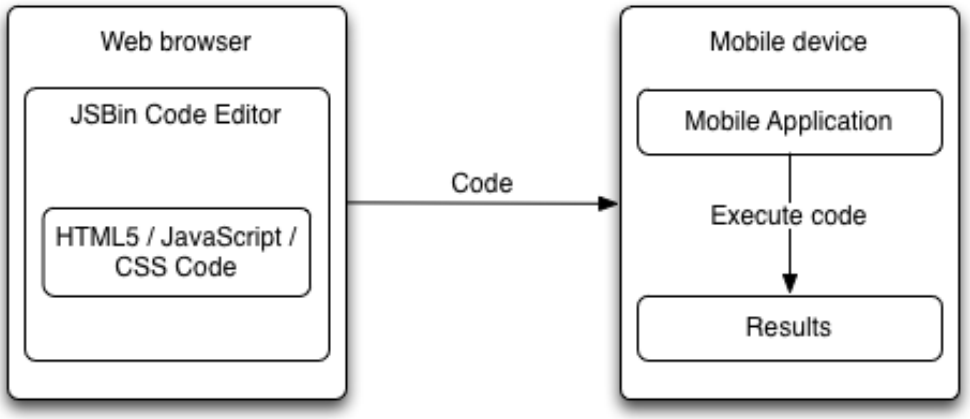

Figure 4.13: System using JSBin.

Chapter 5

User evaluation

5.1 Purpose of the study

The purpose of the user test is to evaluate the prototype to study if Live programming can have advantages for programming mobile devices. Another purpose is to find out if there are any differences developers experience when using Live Programming in comparison to conventional mobile development.

5.2 Method

To get user feedback, a blog post [10] with a video demonstration [8] was published on the internet. The blog post was announced on mailing lists and on Twitter. In total seven users responded and tried out the prototype.

The users were instructed to freely experiment with the system, and then answer a questionnaire with both open and multi-choice questions (see A). The questions ranged over programming related topics and Live Programming

topics. The users were also asked to give a review of the features the prototype provides, and assess the relevance of Live Programming with respect to mobile development.

Five of the users who tested the environment were experienced developers with more than ten years of experience, with the exception of two users who had five years of experience. Five of them are or have been mobile developers, while only two of them had never created a mobile application before. However, all of the testers were already familiar with HTML5 and JavaScript.

5.3 Results

Six users saw the use of HTML5 and JavaScript as a natural approach for mobile development. One user asserts for example that *"This is a very good approach for developing both platform specific apps and cross-platform apps. I think hybrid apps is the way to go. JavaScript, since it is standard, is the choice of a dynamic language for mobile development".* As pointed out by another user, these technologies work well to quickly test ideas: *"It is an easy and cross-platform way to create mobile apps. Very useful for rapid prototyping."*

One user stated that HTML5 and JavaScript are promising technologies, but *"they still lag behind in performance and features on devices".* Two users also experience HTML5 and JavaScript as tools for creating high quality mobile applications, although with a steep initial leaning curve that is worth the effort: *"The initial learning curve for creating a true cross-platform app with these technologies might be steep but I consider it being worth the effort. I have in my own company developed and published cross-platform mobile apps*

using these technologies".

All users appreciated the option for enabling live code editing, a functionality that instantly update the code on the connected mobile devices. One user thinks that the tool is very visual and quick to get started with: *"It is quick to get started, and it is very visual and immediate to see the results on the device as you edit".*

Two questions asked what features in the environment could be improved. Users perceived the UI to be sometimes sluggish, and most of them would like to be able to edit multiple files at once. They also wished a better integration of the documentation with more examples. An aspect that one users brought up is the incremental evaluation of JavaScript when live coding: *"The incremental evaluation of JavaScript is somewhat fishy (since I have insight into this and we have discussed this several times). It feels a bit strange to evaluate a function by typing the name and then closing parens to run it".*

The option for selecting and evaluating the code was more appreciated, since it felt more "distinct". An easy approach suggested by one user is to evaluate everything as done with HTML5, since you are always sure of what gets evaluated. Then, being able to switch to the Smalltalk style is a very good option to have. None of the users experienced any major delay when programming live, and all of them thought the live updating was surprisingly fast.

Users also gave feedback on what could be added to the system, such as debugging to be able to inspect the running code at any time. Moreover, two users wished there could be a way to pack and publish a mobile application. Local file storage was also wished by some users, who perceived the integration with GitHub a bit complicated to use, with too many steps to open and

save files.

A question regarding the benefits of Live Programming was also asked in the questionnaire. Most of the comments were about how quick it was to test out code snippets and make quick prototypes. In this way, once a developer's graphical vision is transferred to all supported platforms, the code can be adjusted and fine tuned appropriately. Most of the users also pointed out that rapid feedback is a great way of learning. Live programming can make the development process much faster: *"You cut the turn-around time of the development cycle and can test things much more quickly. I think this is very helpful for both beginners and experts".* For all users a central benefit was immediate feedback, useful to spot errors and educate novice developers. Scenarios where the users said they would use this programming environment include education as a learning tool, for prototyping any mobile applications written in HTML5/JavaScript, and for full scale application development.

On the whole, the tool was received positively and five users can think of using it for future projects, since it was easy to set up. Two users pointed out the lack of a better file management system.

Chapter 6

Summary and Discussion

6.1 Summary of results

A working Live Programming environment for mobile development has been developed and tested among users with several years of programming experience. The implementation showed how a client-server architecture that uses a combination of asynchronous and pushing communication could enable live updating of code on mobile devices. Also, suitable web technologies were chosen to provide a user friendly interface for encouraging experimentation with Live Programming.

6.2 Discussion

This work set out to investigate the feasibility of achieving Live Performance for mobile development and to study if a Live Programming environment could have benefits for mobile developers. By researching different tools and

architectures a web environment together with a mobile application container were built and tested. Developing a programming environment for the web provided great flexibility in terms of customisation of both the interface and the functionality, making it possible to tweak components for improving the Live Programming experience. Online textual code editors can be used to code mobile applications live, giving the users the possibility to continuously link their ideas and work to a visible result. According to Tanimoto [55], this environment offers level three of liveness, since the representation of the program provides a direct feedback as new changes are made when editing code. At this level, the system executes what is changed instantaneously, and becomes idle when the user stops typing. These components together with suitable network communications showed that it was possible to achieve live performance across different platforms and devices for mobile development.

Both converging and different opinions were gathered when test users experimented with the system. Most of them perceived Live Programming as a great way to approach mobile development. A general feeling was that Live Programming can really help people to improve their productivity and to get good support when programming. By programming live, they appreciated the speed of creating application prototypes, and how quick it was to make changes. Almost all of them pointed out the importance of Live Programming as an educational aid. Early error detection and instant feedback are invaluable elements for having more control and understanding the meaning of a program.

While some users found web technologies to be a natural approach for mobile development, they also thought that these technologies still lag behind in performance and features on devices. In fact, a lot of effort is still needed to provide a definitive tool for creating hight performance hybrid applications comparable to native ones.

6.3 Future work

Many improvements can be done to enhance Live Programming for mobile development. A interesting aspect that constantly emerged during this work was how to achieve incremental evaluation of code and show the correct representation of a program to the users. Different solutions were developed for this prototype, such as re-executing the whole program at code changes for HTML5, or to execute user selections of JavaScript code. An alternative solution would be to manually parse HTML5 code to be able to only inject those nodes that got changed during the development process. Moreover, the design of a custom language developed with Live Programming aspects in mind could address many issues that today are solved with workarounds.

The prototype could also benefit from better documentation integration, such as interactive support for language constructs and hints on how to use external libraries. Live help support could be added to the environment to enhance both the learning process and productivity.

This project aims to give support for Live Programming through text based programming. Mobile development relies heavily on building graphical interfaces (a mobile application without a graphic interface is hard to use and to interface to), and a graphical development environment could be built to further assist the user to realise ideas more easily. Graphical user interface builders are an example of such tools [58].

Chapter 7

Conclusion

This report has presented a prototype for creating mobile applications through Live Programming. The project has been the result of the effort to combine the vision of MoSync for solving the mobile market fragmentation and a new environment that aims to bring Live Programming to mobile development. This project has shown that Live programming can be integrated into mobile development, and that it is possible to provide programmers with tools to realise their ideas quickly.

To summarise the results of the project:

- Is it possible to achieve "Live Performance" when developing on mobile devices?

  A prototype for enabling Live Programming when developing mobile devices has shown that it is possible to achieve "Live Performance".

- Which (network) architectures are suitable for Live Programming of mobile devices?

  By using push technologies it is possible to enable live programming

for mobile devices, since code can be arbitrarily sent and run to any devices listening to a central server.

- Does Live Programming help developers to program mobile devices?

  According to the user test, Live Programming seems to enrich the development process by giving valuable support when prototyping new ideas. Rapid feedback can be a great way of learning how to program and useful to spot errors.

- What are the main differences developers experience compared to conventional mobile development?

  Live Programming makes the process of development faster than conventional mobile development, since it gives immediate feedback and the results of a program can be observed in real-time.

The system prototype was tested with developers coming from different backgrounds. They gave feedback about its potentials, strengths and weaknesses. This was essential to understand what benefits Live Programming can bring to mobile development, and what can be improved. The use of popular languages like HTML5 and JavaScript for creating mobile applications gives the possibility to a large user base to approach mobile development. The use of frameworks such as the MoSync SDK make it possible to use these languages to create hybrid applications that can work on many devices.

Bibliography

[1] Sam Aaron. Quick intro to live programming with overtone. `http://vimeo.com/22798433`. Accessed: 01/04/2013.

[2] Khan Academy. Khan academy. `https://www.khanacademy.org/cs`. Accessed: 01/04/2013.

[3] Apple Computer, Inc. Hypercard. `http://download.info.apple.com/Apple_Support_Area/Manuals/software/0340617AHYPERCARDI.PDF`, 1998. Accessed: 01/04/2013.

[4] Apple Computer, Inc. Introduction to quartz composer programming guide. `https://developer.apple.com/library/mac/#documentation/graphicsimaging/conceptual/quartzcomposer/qc_intro/qc_intro.html`, october 2008. Accessed: 01/04/2013.

[5] J. L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *In XIII International Switching Symposium*, pages 2–7, 1990.

[6] Alan Blackwell and Nick Collins. The programming language as a musical instrument. In *In Proceedings of PPIG05 (Psychology of Programming Interest Group*, pages 120–130, 2005.

[7] Paolo Boschini. Molive. https://github.com/paoloboschini/MoLive. Accessed: 01/07/2013.

[8] Paolo Boschini. Molive. http://www.youtube.com/watch?v=uqsxRTx0Iv8. Accessed: 01/07/2013.

[9] Paolo Boschini. Moliveserver. https://github.com/paoloboschini/MoLiveServer. Accessed: 01/07/2013.

[10] Paolo Boschini. A new live programming tool for mobile app development. http://boschini.se/live.html. Accessed: 01/07/2013.

[11] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, March 1995.

[12] Ricardo Miguel Cabello. glsl. http://glsl.heroku.com/. Accessed: 01/04/2013.

[13] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[14] Angus Croll. How evil is eval? http://javascriptweblog.wordpress.com/2010/04/19/how-evil-is-eval/. Accessed: 01/04/2013.

[15] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[16] Mike de Boer. Javascript github api for node.js. https://github.com/ajaxorg/node-github. Accessed: 01/04/2013.

[17] filearts. Plunker helping developers make the web. http://plnkr.co/. Accessed: 01/04/2013.

[18] Andrew Fischer. Circa a language for live coding. http://circa-lang.org/. Accessed: 01/04/2013.

[19] Gabriel Florit. Live coding interactive sketchpad. http://livecoding.io/. Accessed: 01/04/2013.

[20] Ben Fry and Casey Reas. Processing.org. http://www.processing.org/. Accessed: 01/04/2013.

[21] Adele Goldberg. Why smalltalk? *Commun. ACM*, 38(10):105–107, October 1995.

[22] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[23] Luke Gorrie. Slime: The superior lisp interaction mode for emacs. http://common-lisp.net/project/slime/. Accessed: 01/04/2013.

[24] Chris Granger. Light table - a new ide concept. http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept, 2012. Accessed: 01/04/2013.

[25] T. R G Green and A.F. Blackwell. Thinking about visual programs. In *Thinking with Diagrams (Digest No: 1996/010), IEE Colloquium on*, pages 5/1–5/4, 1996.

[26] Christopher Michael Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, 2003. AAI0805688.

[27] Marijn Haverbeke. acorn.js. http://marijnhaverbeke.nl/acorn/. Accessed: 01/04/2013.

[28] Marijn Haverbeke. Codemirror. http://codemirror.net/. Accessed: 01/04/2013.

[29] GitHub Inc. Gists. https://gist.github.com/. Accessed: 01/04/2013.

[30] Joyent Inc. node.js. http://nodejs.org/. Accessed: 01/04/2013.

[31] Dan Ingalls. Lively kernel. http://lively-kernel.org/. Accessed: 01/04/2013.

[32] Ian Johnson and EJ Fox. Rapid prototyping with d3.js. http://tributary.io/. Accessed: 01/04/2013.

[33] K. Kahn. Seeing systolic computations in a video game world. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 95–101, 1996.

[34] Alan C. Kay. A personal computer for children of all ages. In *Proceedings of the ACM annual conference - Volume 1*, ACM '72, New York, NY, USA, 1972. ACM.

[35] Julie E. Kendall and Kenneth E. Kendall. Information delivery systems: an exploration of web pull and push technologies. *Commun. AIS*, 1(4es), April 1999.

[36] Anton Kovalyov. Jshint, a javascript code quality tool. http://www.jshint.com/. Accessed: 01/04/2013.

[37] John H. Maloney and Randall B. Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the*

8th annual ACM symposium on User interface and software technology, UIST '95, pages 21–28, New York, NY, USA, 1995. ACM.

[38] Alex McLean and TopLap. Toplap - the home of live coding. `http://toplap.org/`. Accessed: 01/04/2013.

[39] MoSync. Javascript, html5, css, c/c++ all in one application. `http://www.mosync.com/docs/sdk/js/guides/wormhole/html5-javascript-wormhole/index.html`. Accessed: 01/04/2013.

[40] Brian Noguchi. everyauth. `https://github.com/bnoguchi/everyauth`. Accessed: 01/04/2013.

[41] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995.

[42] Nicolas Petton. Amber. `http://amber-lang.net/`. Accessed: 01/04/2013.

[43] D. J. Power. A brief history of spreadsheets, 2004. version 3.6.

[44] PubNub. How many characters fit into one message? `https://help.pubnub.com/entries/21100116-How-Many-Characters-Fit-into-One-Message-`. Accessed: 01/04/2013.

[45] PubNub. Real-time network - push real time data to mobile, tablet, web. `http://www.pubnub.com/`. Accessed: 01/04/2013.

[46] Guillermo Rauch. Socket.io: the cross-browser websocket for realtime apps. `http://socket.io/`. Accessed: 01/04/2013.

[47] John Resig. Empirejs: Khan academy computer science. `http://www.youtube.com/watch?v=2sEvlJKAuos#at=1396`. Accessed: 01/04/2013.

[48] Jeff Rose, Sam Aaron, and Fabian Aussems. Overtone. http://overtone.github.com/. Accessed: 01/04/2013.

[49] Erik Sandewall. Programming in an interactive environment: the "lisp" experience. *ACM Comput. Surv.*, 10(1):35–71, mar 1978.

[50] Remy Sharp. Collaborative javascript debugging app. http://jsbin.com/. Accessed: 01/04/2013.

[51] Cincom Smalltalk. Refactoring steroids. http://www.cincomsmalltalk.com/files/bobw/screencasts/Refactoring_Steroids/. Accessed: 01/04/2013.

[52] David Canfield Smith, Allen Cypher, and Larry Tesler. Programming by example: novice programming comes of age. *Commun. ACM*, 43(3):75–81, mar 2000.

[53] Andrew Sorensen. Day of the triffords. http://vimeo.com/2735394. Accessed: 01/04/2013.

[54] Andrew Sorensen. Impromptu. http://impromptu.moso.com.au/. Accessed: 01/04/2013.

[55] Steven L. Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, jun 1990.

[56] Sherry Turkle. *Life on the Screen: Identity in the Age of the Internet.* Simon & Schuster Trade, 1995.

[57] Bret Victor. Inventing onprinciple. http://www.youtube.com/watch?v=PUv66718DII, 2012. Invited talk at the Canadian University Software Engineering Conference (CUSEC), Jan. 2012.

[58] Wikipedia, the free encyclopedia. Graphical user interface builder. http://en.wikipedia.org/wiki/Graphical_user_interface_builder. Accessed: 01/07/2013.

[59] Wikipedia, the free encyclopedia. Read-eval-print loop. http://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop, 2013. Accessed: 01/04/2013.

[60] Simon Willison. Don't be eval(). http://24ways.org/2005/dont-be-eval/. Accessed: 01/04/2013.

Appendix A

User Test Questionnaire

- For how long have you been a developer?
- Are you a mobile app developer?
- Have you used HTML and JavaScript before?
- What do you think of using HTML and JavaScript for creating mobile applications?
- What feature of this tool did you like most?
- What features do you think can be improved?
- What features do you think can be added?
- What features do you think are useless?
- Did you use the checkbox to enable or disable live programming?
- In live mode, how was the latency between a code change and the result displayed on device?

- What advantages do you think live programming could give you for mobile app development?
- For which tasks/projects would you use a tool like this?
- Any other comments/suggestions?

Appendix B

Links to Code and Demo

- A New Live Programming Tool for Mobile App Development

  `http://boschini.se/live.html`

- Video Demonstration

  `http://www.youtube.com/watch?v=uqsxRTx0Iv8`

- Web application/Server code repository

  `https://github.com/paoloboschini/MoLive`

- Mobile application code repository

  `https://github.com/paoloboschini/MoLiveServer`