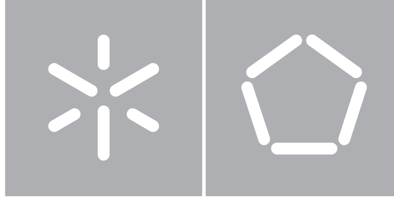
Universidade do Minho
Escola de Engenharia

André Filipe Pereira Félix

Web based player for public displays

Outubro 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Filipe Pereira Félix

Web based player for public displays

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Rui José

Outubro 2014

ACKNOWLEDGEMENTS

First of all, I would like to thank my parents and my family, without their help and support I wouldn't even be able to have the opportunity to undertake this project.

Secondly I would like to thank my supervisor, Professor Rui José, for the opportunity to work with him, for his patience and availability.

A special thank you to my colleagues at DISPLR, for their involvement in the project and their willingness to help, I hope the next months of work with them will be just as good as the past ones.

Last, but not least, to the friends I made since my first year at the University, specially to the ones with whom I shared a home, I thank you, your companionship and all the unforgettable moments we spent together definitely helped me brave the problems during my academic course.

ABSTRACT

The most important element in a network of public displays is a piece of software, the player, it is responsible for interpreting the presentation instructions, which are sent in a specific format, and make the content visible to the users according to those instructions. Usually both the software and instructions format are proprietary.

One of the big issues regarding this type of software is their restrict system requirements. A player is usually conceived having a specific target platform, this creates some issues when deploying new displays. With the increasing development of web technologies emerges a solution to this issue: a web based player with few system requirements and the capability to be deployed in a bigger range of platforms.

The goal of this investigation is to design and implement a web based player using web technologies such as HyperText Mark-up Language 5 (HTML5) and JavaScript. The new Chrome Packaged Apps technology is also being looked at as a way to easily distribute and deploy the software.

Ultimately this web based player aims to increase the reach and availability of the public displays networks by creating a platform to which the non-proprietary developer can create content to.

Keywords: Public displays, public displays networks, player, web technologies, web based player, HTML5, JavaScript, chrome packaged apps.

RESUMO

O elemento mais importante numa rede de ecrãs públicos é um pedaço de software, o player, este software é responsável por interpretar as instruções de apresentação, que são enviadas num formato especifico, e tornar o conteúdo visível aos utilizadores de acordo com essas instruções.

Um dos grandes problemas deste tipo de software são os seus requisitos de sistema restritos, normalmente cada player é concebido para uma determinada plataforma, isto cria alguns problemas quando é necessário implantar novos ecrãs. Com o desenvolvimento das tecnologias web emerge uma solução para este problema: um player baseado em tecnologias web com requisitos de sistema baixose a capacidade de ser implantado numa maior quantidade de plataformas.

O objetivo desta investigação é de desenhar e implementar um player baseado em tecnologias web recorrendo a tecnologias como o HTML5 e o JavaScript, a nova tecnologia Chrome Packaged Apps também está a ser abordada como forma fácil de distribuir e instalar o software.

No futuro este player tem o objetivo de aumentar o alcance e disponibilidade das redes de ecrãs públicos ao criar uma plataforma para a qual programadores não proprietários podem criar conteúdo.

Palavras chave: Ecrãs públicos, redes de ecrãs públicos, player, tecnologias web, player baseado em tecnologias web, HTML5, JavaScript, chrome packaged apps.

CONTENTS

Contents vii
Acronyms xiii
1 INTRODUCTION 1
- 1.1 Motivation 1
- 1.2 Challenges 2
- 1.3 Objectives 2
- 1.4 Document Structure 3
- 1.5 Summary 3

2 STATE OF THE ART 4
- 2.1 Digital Signage Players 4
  - 2.1.1 Ubisign 4
  - 2.1.2 Xibo 4
  - 2.1.3 Rise Vision 5
  - 2.1.4 Concerto 5
  - 2.1.5 OpenSign 6
  - 2.1.6 IAdea 6
  - 2.1.7 Sapo Digital Signage 6
  - 2.1.8 TargetR 6
  - 2.1.9 NoviSign 7
  - 2.1.10 OpenSplash 7
  - 2.1.11 Signagelive 7
  - 2.1.12 Summary 7
- 2.2 Web Based Player 8
  - 2.2.1 Prototype of the Web Based Player 8
- 2.3 Summary 9

3 WEB BASED MEDIA PLAYER FRAMEWORK 10
- 3.1 Player Operation Method 10
- 3.2 Functional Layers 11
  - 3.2.1 Native Layer 11
  - 3.2.2 Web Engine Layer 12
  - 3.2.3 Web Layer 12
  - 3.2.4 Applications Layer 12
- 3.3 Reference Stacks 12

3.4 Common Functionalities From Other Players 14
3.4.1 Fault Tolerance 15
3.4.2 Content Management 15
3.4.3 Logging 16
3.4.4 Execution 16
3.4.5 Security 16
3.4.6 User Interaction 16
3.4.7 Updates 16
3.5 Key Functionalities 16
3.5.1 Native Layer 17
3.5.2 Web Engine Layer 17
3.5.3 Web Player Layer 18
3.5.4 Applications Layer 18
3.6 Player Architecture 19
3.7 Scheduling Principles and Format 20
3.8 Communication and Player Services 23
3.8.1 Player Services 23
3.8.2 Communication Protocol 27
3.8.3 State Diagrams 31
3.9 Libraries 32
3.9.1 Applications Life Cycle 33
3.9.2 Player Specific Library 33
3.9.3 Applications Specific Library 35
3.10 Summary 36
4 CASE STUDIES 38
4.1 Case studies and their targeted platforms 38
4.2 Displr 38
4.2.1 Control Module 39
4.2.2 Server Module 40
4.2.3 Registration Module 41
4.2.4 Scheduling Module 42
4.2.5 Media Handler Module 44
4.2.6 Apps Management Module 46
4.2.7 Logging Module 48
4.2.8 Interaction Module 49
4.2.9 Activity 50
4.2.10 Timer 53
4.3 Ubisign 53

4.3.1 Server Module 54
4.3.2 Registration Module 54
4.4 Android and iOS 54
4.4.1 Media Handler Module 55
4.4.2 Apps Management Module 55
4.5 Samsung SmartTV 55
4.5.1 Media Handler and Apps Management Module 55
4.5.2 Interaction Module 56
4.6 Summary 56
5 TESTING 57
5.1 Planning 57
5.2 Tests 58
5.2.1 Schedule Execution and Precision Tests 58
5.2.2 Stress Tests 59
5.2.3 Memory Profiling 60
5.2.4 Unit Tests 62
5.2.5 Deployment Tests 63
5.3 Summary 64
6 CONCLUSION 65
6.1 Conclusion 65
6.2 Future Work 66
Appendices 68
A EXAMPLE SCHEDULE 69
B EXAMPLE LOG FILE AND TRACE FILE 74
C SCREENSHOTS 77

LIST OF FIGURES

Figure 1 Layered architecture 11
Figure 2 Reference Stacks for Personal Computer (PC) and SFF 13
Figure 3 Reference Stacks for Raspberry Pi and pc-Duino 13
Figure 4 Reference Stacks for Rikomagic, Chromebox and Samsung Smart TV 14
Figure 5 Generic player's architecture 19
Figure 6 Sequence of containers and/or applications 22
Figure 7 A layout, can be composed of containers or applications 23
Figure 8 Selector composed by a list of containers and/or applications and a set of rules 23
Figure 9 A player undergoes this process when it is deployed 24
Figure 10 Periodic synchronization process 24
Figure 11 Process of posting logs to the server 25
Figure 12 Player restarting process 25
Figure 13 Process of unregistering a player and reseting it to factory specs 26
Figure 14 Process of changing a player's domain 26
Figure 15 Process of updating a schedule on the server 27
Figure 16 Player state as seen by the server 32
Figure 17 Player state as seen by itself 32
Figure 18 Applications state diagram 33
Figure 19 Control module class diagram 39
Figure 20 Server module class diagram 41
Figure 21 Registration module class diagram 42
Figure 22 Scheduling module class diagram 44
Figure 23 Media Handler module class diagram 46
Figure 24 Apps Management module class diagram 47
Figure 25 Logging module class diagram 49
Figure 26 Interaction module class diagram 50
Figure 27 Activity class diagram 50
Figure 28 Scheduling sequence diagram 51
Figure 29 Timer class diagram 53
Figure 30 Example of a trace file 58
Figure 31 Example of a result file 59
Figure 32 Memory profilling report 61

Figure 33 Memory profilling report over a week 62
Figure 34 Example result page of the unit test 63
Figure 35 Home screen of the player. 77
Figure 36 Apps being executed. 78

LIST OF TABLES

Table 2 Digital signage solutions comparison 7

Table 3 Dependencies of each of the layers 11

Table 4 Messaging protocol 27

Table 5 Deployment table 64

ACRONYMS

API	Application Programming Interface. 11, 12, 16–18, 32, 33, 39–42, 44, 46, 47, 49, 50, 54, 55, 60, 61
CMS	Content Management System. 7
CSS	Cascading Style Sheets. 2, 5
GPS	Global Positioning System. 6
HTML5	HyperText Mark-up Language 5. iv, v, 2, 3, 6, 54, 55
HTTP	HyperText Transfer Protocol. 6
JSON	JavaScript Object Notation. 6, 20, 28, 29
MIME	Multi-purpose Internet Mail Extension. 28
OS	Operating System. 5–7, 15, 16
PC	Personal Computer. x, 6, 13, 39, 61
PHP	PHP: Hypertext Preprocessor. 4, 5, 58, 59
POS	Point of Sale. 6
RSS	Rich Site Summary. 4
SAAS	Software as a Service. 4, 7
SDK	Software Development Kit. 55
SFFC	Small Form Factor Computer. 1, 5, 6, 66
SMIL	Synchronized Multimedia Integration Language. 6, 53
SQL	Structured Query Language. 4
URL	Uniform Resource Locator. 6, 8, 20, 46
USB	Universal Serial Bus. 66

WPF	Windows Preentation Foundation. 7, 53, 54
XML	eXtensible Mark-up Language. 4, 5
XSLT	eXtensible Style-Sheet Language. 53

1

INTRODUCTION

This work is centred around digital signage, this is the use of displays to communication for marketing or informative purposes. A usual case of digital signage are public displays networks, these are networks of displays used for digital signage. The software behind these displays are the players, they are software responsible to schedule and display content on the screen. This work aims at developing one of these players, using web technologies.

This chapter presents the motivation for this work as well as the challenges and key objectives behind it.

1.1 MOTIVATION

A key element of a public display network is the software that handles the displaying of content on screen. This software, or player, is responsible for receiving a set of content and a set of instructions – the schedule, and, according to the instructions, display the content on the screens. The players are usually proprietary software with very strict system requirements and very closed (usually proprietary as well) scheduling format.

The development of a web based player is motivated by the need to solve the portability issue as a way to spread the usage of public displays. Following the current technological trend centered around web technologies, it is possible to take full advantage of the portability of these technologies and create a software capable of being deployed on a wider selection of systems.

The usage of web technologies also lowers the system requirements of the web based player as opposed to its native software counterparts. This allows the possibility of using new, and cheaper, platforms such as Android pens and Small Form Factor Computer (SFFC), effectively lowering the cost of deploying a public display.

Finally, a web based player aims at taking advantage of the ever growing community of web developers by creating a platform where new content can easily be created.

1.2 CHALLENGES

The main challenges of this work are largely tied to the implications of using web technologies to implement a digital signage player. Considering the characteristics of the web technologies that are going to be used during this work (HTML5, JavaScript and Cascading Style Sheets (CSS)) it is possible to identify the following challenges:

- Integration with the native system, some characteristics of a digital signage player relies on interacting directly with the operative system. Operations like obtaining critical system information for player diagnostics and controlling and scheduling system reboots to ensure a healthy system is something that is expected from a digital signage player. This was trivial when implementing a traditional native application, however web technologies have limited access to the native system.

- Ensuring the player continues to work properly even during periods of no connectivity to the internet. Being a web application, the web based player will require internet connectivity in order to obtain the schedule and download all the necessary resources to display it. Given the nature of a public display it is necessary to ensure that problems in the system, like a lack of internet connectivity, are not shown to the users. These two characteristics of the player create the necessity of an off-line mode for the web based player. This is not something a traditional web application has to deal with, so mechanisms have to be put in place in order to handle the caching and storage of resources to ensure a continuity of the service during periods of no internet connectivity.

- Most of the contents that will be scheduled by the web based player are web applications, given their nature it is necessary that the player is ready to respond positively to crashes originated from those applications. On a native environment this could easily be done with multi-threading, however, the web based player's logic will be developed mainly with JavaScript, which has a single threaded nature.

These challenges can be compared to the ones identified by Taivan et al., and, in summary, they are centred around the limitations of web technologies and how to make use of those technologies to implement a web based player that can offer the same characteristics of a native application while taking advantage of the web technologies' features.

1.3 OBJECTIVES

The main objective of this work is to specify and implement a digital signage player for deployment on public displays network using web technologies.

That objective can be broken down into the following goals:

- Specify a layered architecture that provides the player with the modular nature necessary to implement different versions to be deployed on the identified reference stacks;
- Implement the web based player using web technologies (HTML5, JavaScript and CSS) and design a suitable scheduling algorithm;
- Specify the necessary server side requirements to enable the player with a way to: fetch new schedules, log information and register and unregister instances of the player;
- Obtain quantitative metrics from testing processes to evaluate the validity of the player.

1.4 DOCUMENT STRUCTURE

This document is divided into six chapters, each covering a different part of the work.

- Introduction: this chapter presents the motivation, challenges and objectives of this work.
- State of the Art: the second chapter contextualizes the current state of public displays technologies, the go-to digital signage software solutions and the previous work on this subject.
- Web Based Media Player Framework: the third chapter presents all the work leading to the first implementation of the web based player: the layered format, reference stacks, system requirements and architecture.
- Case Studies: the forth chapter contains the different implementations that were developed during the course of this work for four different case studies.
- Testing: the fifth chapter contains all the information related to the testing of the web based player, from the planning to the results.
- Conclusion: the sixth and final chapter summarizes the entire work, which of the objectives were met and presents the next logical steps in this work.

1.5 SUMMARY

This chapter presented the motivation behind this work: the usage of web technologies to implement a more traditionally native type of software to basically obtain the best of both worlds. Migrating this type of software to web technologies isn't something trivial and raises limitations that did not exist for native software.

The objectives of this work were also presented and can be summarized as the implementation of a digital signage player using web technologies.

2

STATE OF THE ART

This chapter will contextualize where the development of digital signage players is at the moment, the characteristics of existing software and the previous investigative work that introduced this work. This chapter will also feature a quick analysis to the technologies that will be involved in the implementation of the Web Based Player.

2.1 DIGITAL SIGNAGE PLAYERS

Along the past few years a boom in Digital Signage technology could be observed, mainly due to developments in the technologies involved in creating networks of public displays. This chapter will present and analyse eleven of those digital signage software solutions and compare them with each other.

2.1.1 *Ubisign*

Ubisign (2014) provides its Digital Signage solution as a Software as a Service (SAAS), as a web service it can be remotely managed by the customer and provides a friendly, full-featured user interface. It can be fully customizable in terms of screen layouts and media content scheduling, it also supports external dynamic data content like Rich Site Summary (RSS) feeds or eXtensible Mark-up Language (XML). The player provides support to web 2.0 content, creating a great and interactive user experience. It supports content caching and crash recovery systems, providing some level of service in case of connectivity issues. The scheduling format is proprietary.

2.1.2 *Xibo*

Released as an open source digital signage player on 2006, Xibo (2014) follows a client/server paradigm. The server is a PHP: Hypertext Preprocessor (PHP)/MyStructured Query Language (SQL) web application that runs on Windows, Mac or Linux and it is the administration tool of the application, there a user can upload content, design layouts or create content schedules to be displayed on the clients. The client is a display connected to a pc running the client application that is used to show

the content, each of the clients connected to the server can have its own schedule. There are two versions of the client software, a .Net application, the first stable version to be released and more recently a Python version, that's been looked at as a suitable replacement for the .Net version for both Windows and Linux platforms. The client – server configuration is done by exchanging XML Schemas, on each of those is included the scheduling information, layouts and content needed to the client's functionality.

2.1.3 *Rise Vision*

Rise Vision (2014) has so far released two open source Digital Signage Players for Linux and Windows 7, they both work on top of a standardized web browser, in this case Google Chrome. They are able to play any kind of HTML content that is supported by the browser and are also able to run JavaScript. The displays can be managed through a web platform where the schedules can be created without requiring any proprietary format. Each display is managed by a JavaScript application running on the browser that is responsible for showing the content according to the instructions received. The application makes use of several objects during its execution:

- Display, represents any device capable of displaying content;
- Viewer, its the application running on a browser that displays the content on each of the Display objects;
- Player, the native Operating System (OS) application responsible for running the viewer objects;
- Schedule, its a sequence of contents to be displayed;
- Presentation, each of the contents that are going to be displayed is represented by one instance of the Presentation object;
- Placeholder is a fixed area inside a Presentation that contains a list of Gadgets, much like how a Schedule contains a list of Presentations;
- Gadget represents a Google Gadget developed by Rise Vision.

2.1.4 *Concerto*

Concerto (2013) is an open source Digital Signage player, implemented as a web application using PHP, Concerto makes heavy use of CSS, JavaScript and jQuery for the displaying functionality. Each instance of the player requires its own SFFC (Small Form Factor Computer) that is paired on a 1:1 ratio with the display it is meant to control, it is however possible to have 1 SFFC controlling 2 displays with a dual video card configuration. The front-end of the Concerto player only has the responsibility

to display the content as it is passed on by the back-end, all the configuration and scheduling features are hosted on a central server that communicates with all the SFFC that make up the display network.

2.1.5 *OpenSign*

On March 2013 AOpen (2013), a well established digital signage enterprise release OpenSign, a web based digital signage platform for Android devices. The service can be deployed on several devices and controlled by a single PC connected to the web, the service also provides local redundancy in cases of connectivity issues, the software can integrate any kind of external live feed such as Twitter and Point of Sale (POS) data. This is however a proprietary software.

2.1.6 *IAdea*

IAdea (2014) commercializes its own proprietary hardware running a software compatible with Synchronized Multimedia Integration Language (SMIL) W3C (2012b) and HTML5. These SFF (Small Form Factor PC) are Linux based and provide an easy to use interface that allows the user to schedule various types of content, manage layouts and external sensors. It also has some connectivity issues tolerance by providing an offline mode.

2.1.7 *Sapo Digital Signage*

Sapo (2013) was developed internally as a means to support Digital Signage on Sapo's Codebits 2012 to be ran on Raspberry Pi devices, this solution eventually grew into a full-blown client-server application. This software is capable of interpreting JavaScript Object Notation (JSON) objects that consists of a list of content Uniform Resource Locator (URL) to be displayed, these content URLs can be any assortment of HTML pages, live streaming and video, however no local content can be stored on the devices running the software. The client – server communications are done through an HyperText Transfer Protocol (HTTP) pooling system.

2.1.8 *TargetR*

Following a client – server paradigm TargetR (2014) can be deployed on Android, Raspberry Pi and PC. The TargetR server was developed using Java and runs on a Linux OS, the server provides access to the administration interface that features a wide range of configuration, management and scheduling options. The client, as stated above can be deployed on three different platforms, features a crash recovery system and support for camera, Global Positioning System (GPS) and live television functionalities.

2.1.9 *NoviSign*

Distributed as a SAAS, the Novisign (2014) solution is aimed at Android devices, it can be deployed on anything ranging from tablet devices to Android TV devices. It uses a client – server paradigm to operate and offers all the necessary features to create and manage schedules on the server and then propagate them to all the integrated devices.

2.1.10 *OpenSplash*

OpenSplash (2014) is a free, multi-platform open source media player that can be driven by any content management and scheduling system, it is highly extensible with a plug-in architecture. The software receives the displaying instructions as a play-list and media files from the server, it supports video, dynamic screen layouts and overlapping and depth order. It is not a complete Digital Signage solution as it requires a content management system (Content Management System (CMS)) that is not distributed with this software.

2.1.11 *Signagelive*

Signagelive (2014) allows the user to display dynamic content on any browser that's connected to the web, it runs on Windows or Android OS and features a wide array of functionalities. The content and display management and configuration is done through a web interface.

2.1.12 *Summary*

In the following Table 2 we can see a comparison of the eleven digital signage solutions studied.

Player	Platform	Type	Licensing
Ubisign	Windows	Windows Preentation Foundation (WPF)	Proprietary
Xibo	Windows/Linux	Web Browser	Open Source
Rise Vision	Windows/Linux	Google Chrome	Open Source
Concerto	-	Web Browser	Open Source
OpenSign	Android	Web Browser	Proprietary
IAdea	Linux	Web Browser	Proprietary
Sapo Digital Signage	Linux	Web Browser	Open Source
TargetR	Android/Linux	Web Browser	Proprietary
NoviSign	Android	Web Browser	Proprietary
OpenSplash	Windows/Linux	Web Browser	Open Source
Signagelive	Windows/Android	Web Browser	Proprietary

Table 2.: Digital signage solutions comparison

From the information collected in the above table it is possible to observe two major characteristics of the current digital signage solutions: the system dependency and the closeness of some of those solutions. Furthermore the sole purpose of these solutions is to display content on a screen.

The web based player aims at being much more then that: taking advantage of the web technologies behind it, it is possible to create a truly interactive experience with the users. With an open platform ready to display applications that are limited only by the imagination of the developers. This is what separates this work from what was being done before.

2.2 WEB BASED PLAYER

The Web Based Player concept was already explored before, this section will analyse two of the most prominent investigations on the matter.

Lindén et al. (2010) specifies a very similar concept to the web based player this investigation aims to create. The player uses a modular architecture, there are three larger modules: the resources manager, the layouts manager and the visualization module, each of these is in charge of a part of the whole process. The resources manager handles all the content and communicates it through a web service interface to the layouts manager, which in turn is in charge of creating the layouts of the schedules and sending them to the visualization module. The visualization module is essentially a JavaScript application working atop the browser injecting HTML code into a skeleton HTML page.

Taivan et al. states four challenges that must be had in mind when developing a Web Based Player. The first is content management and it evidences three specificities that have to be met in order to present the users with a pleasant experiences when using a public display: avoid idle times, prevent users from noticing the occurrence of errors and support partial disconnection problems. The second is content addressability and it refers to the usage of resources identifiers, providing each web content resource with its own identifier (URL) much like any resource-oriented architecture. The third is visual adaptation and integration and refers to the handling of different display sizes, responsive Web Design has become a standard practice in web development and the same should apply to a Web Based Player, it has be able to cope and handle different display sizes and resize the content to be displayed accordingly. Finally, execution environment refers to the security restrictions and how they can raise issues for the integration with the execution environment, even though workarounds exist to circumvent these restrictions they have to handled carefully as to not create security flaws in the application.

2.2.1 *Prototype of the Web Based Player*

This work is building on a previous work by Carneiro (2013), which originated an early prototype of the web based player. To achieve that it was necessary to thoroughly study the current state of the art and the technologies involved in developing a web application with the nature of the web based

player.

Carneiro (2013) identified a set of reference stacks, which reduced the number of platform/system combinations that would be targeted and supported by the web based player. The system requirements for the web based player based on common features from other players, furthermore he also designed the scheduling format that would eventually evolve to become the format used by the player today.

That investigation was essentially a first input to this work, an initial iteration to the creation of the web based player. Its purpose was to clearly identify the design space and specificities of a software like the web based player and provide this work with an invaluable input with which to start from and eventually reach a state of maturity where the web based player could actually be used in a real digital signage environment.

2.3 SUMMARY

In this chapter the characteristics and limitations of the current solutions regarding Digital Signage were explored, from that study the major issues that were referenced before could be confirmed: strict system requirements and few support to open source developers. It was also presented the solution that aims to fix this problem, the first investigative works and the initial steps towards a functional first version of the Web Based Player.

3

WEB BASED MEDIA PLAYER FRAMEWORK

This chapter will cover all the concepts that led to the development of the web based player. It will present all the information resulting from the specification of the player: reference stacks, system requirements, functionalities and architecture of a generic implementation of the web based player. Furthermore this chapter will also have the specification of the scheduling format and the principles that led to that format. Finally it will cover, in a brief manner, the player services and communications protocol that support the player's execution.

3.1 PLAYER OPERATION METHOD

The Web based player, is meant to receive a schedule from the service it is being managed from. This schedule is essentially a list of content, usually web apps, and a set of instructions. These instructions dictate how the player will display each of the contents on the screen.

Upon receiving a new schedule, the player will validate it and transform it into a scheduling tree. This scheduling tree is essentially the schedule transformed into a logical object to be used by the player modules. The scheduling tree is then parsed and each of the contents and instructions present there will generate events. These events can be: prepare a content, start a content or stop a content, these operations are done to the frame containing the content. These web frames are created, hidden, made visible, and ultimatelly destroyed based on the type of event being execute on them. The full list of events form the event table, which in conjunction with the scheduling tree form an activity.

An activity is a finite class that contain the scheduling information and the methods to display it on the screen. Besides the activity there is another class involved in the process of displaying content: the timer. This class generates a "tick" periodically, each time a "tick" is generated the activity moves forward one position on the event table and executes any event on that position. When the event table reaches its end, the activity is finished and it is destroyed.

This is the basic way of the player to display a schedule, the inherent functionalities of each module will be explained on its respective section.

3.2 FUNCTIONAL LAYERS

The player will present a layered architecture, this will provide it with the ability to serve a wider range of display system and hardware settings.

The following Figure 1 illustrates the layered architecture of the player, it is divided into 4 layers: Applications layer where the displayable content is executed; the Web Player layer that combines generic and system specific javascript modules; the Web Engine layer containing the modules that work as an extension to the system's web engine; and finally the system specific Native layer. Each of these layers offers an Application Programming Interface (API) to the layer directly above it which is used to communicate between those layers.

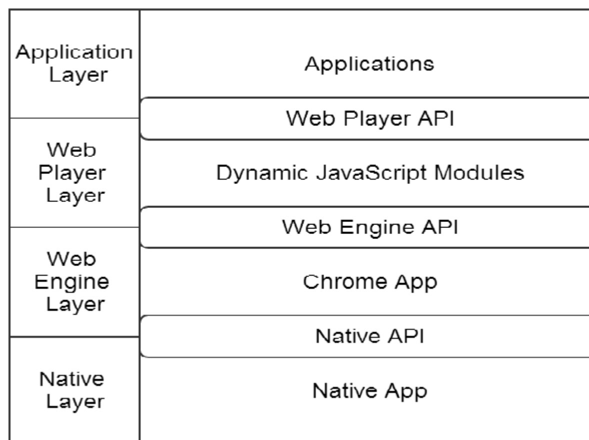


Figure 1.: Layered architecture

The following table 3, shows the layer's dependencies and when each of them require a new implementation upon system change.

Layer	Dependencies	Requires new implementation when
Application Layer	Web Layer API	Rare cases of API changes
Web Layer	Web Engine Layer	Web Engine changes
Web Engine Layer	Web Engine	Web Engine changes
Native Layer	System and platform	Native system changes

Table 3.: Dependencies of each of the layers

3.2.1 *Native Layer*

The native layer includes those modules that need to be implemented for a specific operating systems or hardware platform. Typical functionality at this layer includes fault tolerance procedures, privileged access to system resources, bootstrapping and screen on/off handling. Being platform

specific, the functionality provided by this layer may change considerably based on properties of the underlying platform. A common situation may be the case where this module corresponds to an existing digital signage system or to a particular type of set-top box or a smart TV.

3.2.2 *Web Engine Layer*

The Web Engine layer extends the web engine with functionality that needs to circumvent the limitations imposed on javascript apps that execute on top of the standard web execution environment that is offered by the web engine to web applications. Typical functionality at this layer includes advanced content management operations, such as caching or pre-fetching procedures.

3.2.3 *Web Layer*

This module takes care of all the logic in the application, it integrates the other two modules, using the functionalities they provide to ensure the application is able to run as intended.

The most specific of the layers, every different version of the player has a specific web module containing a scheduler and/or other modules to be loaded and used by the Web Engine module. This app also provides a JavaScript loader so other versions of the player can load and use their specific modules.

3.2.4 *Applications Layer*

The content apps that are to be scheduled and displayed by the player are able to communicate with the scheduler through an API that is made available by a library that can be loaded by the apps.

3.3 REFERENCE STACKS

Even considering the extensive portability of web technologies, it is necessary to consider the differences in the logical layers that make up the execution environment. As such several reference stacks were specified, Figures 2, 3 and 4, each containing a combination of system layers (hardware, operative system and web engine).

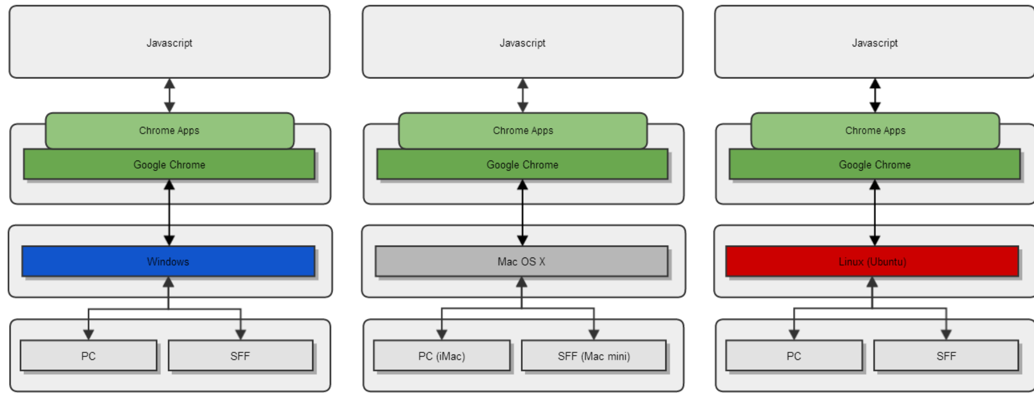


Figure 2.: Reference Stacks for PC and SFF

These reference stacks aim at reducing the number of possible combinations of the system layers, thus defining the real target platforms of the player and creating specific development objectives.

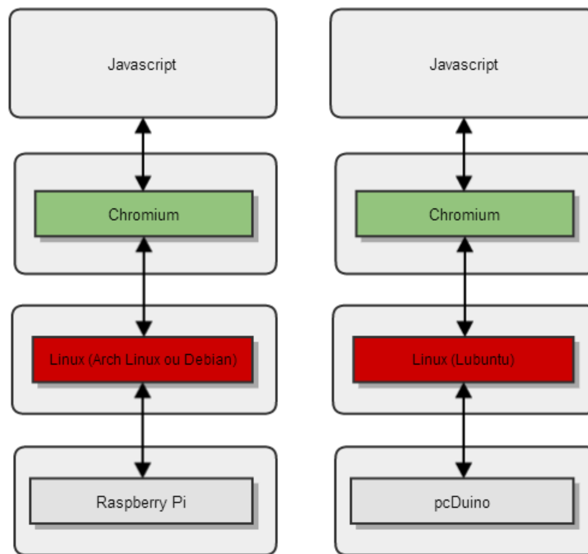


Figure 3.: Reference Stacks for Raspberry Pi and pc-Duino

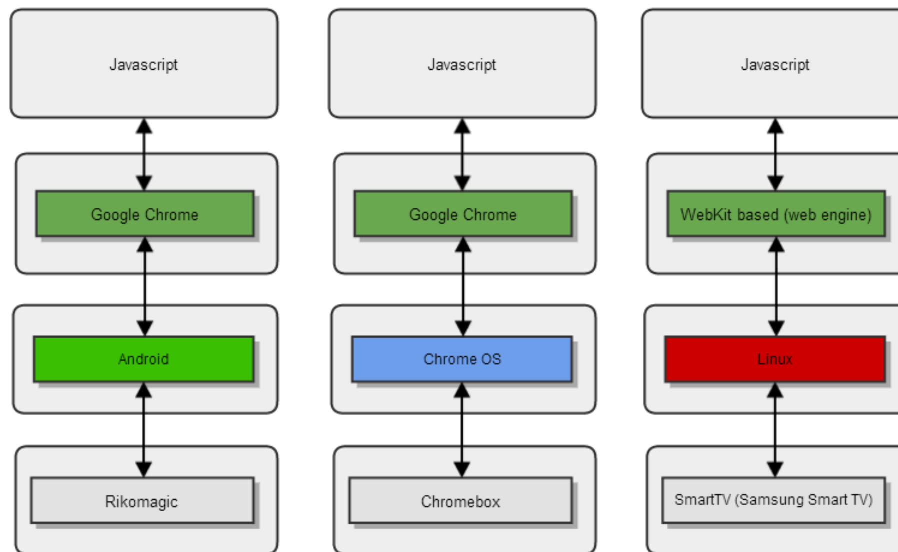


Figure 4.: Reference Stacks for Rikomagic, Chromebox and Samsung Smart TV

3.4 COMMON FUNCTIONALITIES FROM OTHER PLAYERS

The system requirements presented here were identified in Carneiro (2013)'s work in order to provide the player with the necessary tools to ensure it can function properly. The system requirements will be divided into seven categories:

- Fault Tolerance: functionalities responsible to keep the player alive after a faulty behaviour;
- Content Management: preparation and caching of content;
- Logging: functionalities to keep a history of the player's behaviour;
- Execution: functionalities to execute the schedules;
- Security: safe communication with the service and the content that's being displayed;
- User interaction: provide the user with means to interact with the content being displayed;
- Update: ways to handle software updates that the player may suffer.

The following functionalities are the most common among the other digital signage sollutions studied:

- Power on/off displays: ensured during the installation by scheduled events to turn the display on/off at the designated times.

- Enhanced system information: transmit to the service technical informal about the OS, hardware, etc., this allows the application to make adjustments and diagnostics.
- Remote commands: turn the system on, reboot, etc..
- Display control: gives the possibility to turn off the display. It can be a command and have numerous reasons: energy saving, hiding content loading situations, system reboots, etc..
- Sensors: support to connect sensors, like Kinect, to the display.
- Resolution adjustments: the player receives fixed size schedules, this allows to adapt the presentation resolution to the display's resolution. This information is obtained from the OS.

3.4.1 *Fault Tolerance*

- Crash recovery: this functionality consists in detecting application crashes and acting accordingly by rebooting the application, therefore resolving the crash. It is usually supported by a "Watchdog" service, independent from the player application so it won't crash when the player itself does.
- Crash avoidance: this functionality consists in preventing crashes by surveilling the resources consumption and other important system information. It is normally supported by a "Watchdog" service, independent from the player application so it won't crash when the player itself does. It is specially important in machines where memory leaks and driver problems are expected. It may require stopping the player.
- Safe execution: detection of any problems with the execution of applications in the web engine and recover from those situations.

3.4.2 *Content Management*

- Content prefetch: this content activation process allows the scheduler to prepare the contents that are about to be presented, this allows a fluid visualization and no loading times.
- Content caching: in some players, the system pre-downloads all the contents referenced in a schedule before it is presented. This process is usually supported by a module independent from the scheduler that interfaces with the scheduling server. When a new schedule is received the scheduler will check all the content present in it and load the content that still isn't avaiable in that machine before alerting the scheduler to the presence of a new schedule. Links to remote content can be made local before being handed to the scheduler.

3.4.3 *Logging*

- Scheduling logs: support for logging of scheduling events occurring at the web module.
- Screenshots: generate and send screenshots of what's being presented. ”Watchdog” takes the screenshot independently and stores the image on the server.

3.4.4 *Execution*

- Execution control: allows to stop or reboot the scheduler, when, for example, its necessary to change the schedule.
- Scheduler: supports the interpretation and execution of schedules.

3.4.5 *Security*

- Application integrity: assures the application integrity, encryption and checksums.

3.4.6 *User Interaction*

- Interface: allows for direct and local control over the player's operation.

3.4.7 *Updates*

- Automatic updates: automatic and non-assisted software updates. This process can be supported directly by the OS own mechanisms, as long as the software is ready for that. Using Ubisign's player as example, all the process is supported by a speciallized Windows application.

3.5 KEY FUNCTIONALITIES

This section presents the key functionalities tied to each of the layers respective API. This creates the bridge between the system requirements and the API offered by each of the layers.

3.5.1 *Native Layer*

The native module is responsible for all the interaction the player needs with the hosting system. The more prominent functionalities this module has, among others, are management of the web engine module (start-up, reboots, etc.), handle crash recovery/avoidance and display control.

Native API

The native module offers several functionalities to the Web Engine module, most of them are operations that interact directly with the system. The functionalities offered by this API are dependant on the system running the player and as such some methods may or may not be available.

ENVIRONMENT

- Display control: turn the display on/off;
- Systems information: provides information about the system running the player;
- Remote commands: turn the system on/off and reboots;
- Sensors: provides support to handle any sensor available to the player;

FAULT TOLERANCE

- Crash recovery: tools to recover the player state in case of a crash;
- Crash avoidance: tools to monitor the system and check for unusual behaviour that may lead to a crash.

3.5.2 *Web Engine Layer*

The web engine module works as an extension of the web engine and therefore offers functionalities tied to it. Cache management and content prefetch are the most common functionalities to this module.

Web Engine API

This API offers functionalities to be used by the Web Player Layer and focuses mainly on some key aspects of the scheduling of applications: prefetch and caching.

CONTENT MANAGEMENT

- Content prefetch: allows the scheduler to request the preparation of content before it being displayed;
- Content caching: allows the scheduler to request the download and caching of certain content of an app to accelerate it's preparation and presentation process.

3.5.3 *Web Player Layer*

The Web Layer constitutes the execution environment for applications. The functionality offered by this layer includes the ability to coordinate scheduling with applications, access to environment information, logging and interaction. Some of this functionality depends on services offered by lower layers and may not always be available, depending on the specific stack.

Web Player Generic API

This API is offered to the applications being scheduled, it provides functionalities such as logging, access to sensors and some level of user interaction. The methods offered by this API are system dependant and may or may not be available on certain systems.

ENVIRONMENT

- Sensors: allows the apps to make use of a sensor that can be handled by the player;
- Resolution adjustments: request resolution information or a resolution change from the player.

LOGGING

- Scheduling Logs: request a log entry to be added to the execution log;
- Screenshots: request the player to take a screenshot and add it to the execution log.

USER INTERACTION

- Interface: provides access to a user interface.

3.5.4 *Applications Layer*

As part of their execution in the player, applications may interact with their environment to optimise multiple aspects of their operation.

Assumptions

It is assumed that an app implementing this library follows this set of "rules":

- The app should have the necessary methods available for subscription (play, finalize, standby, prefetch, etc);
- The app should be ready to handle a set of parameters sent by the player on the messages, it should, however, be able to properly function without them.
- These parameters include: lease times sent by the player informing how long the app has to display and lease renewals, lease extensions/shortenings and lease renewals.
- The app should be ready to handle situations where no prefetch time is given to prepare the content.

3.6 PLAYER ARCHITECTURE

As mentioned above the functionalities of the player can be decomposed into several categories, to better handle this and to provide the player with a much needed modular architecture, the functionalities were broken down into the modules illustrated in Figure 5.

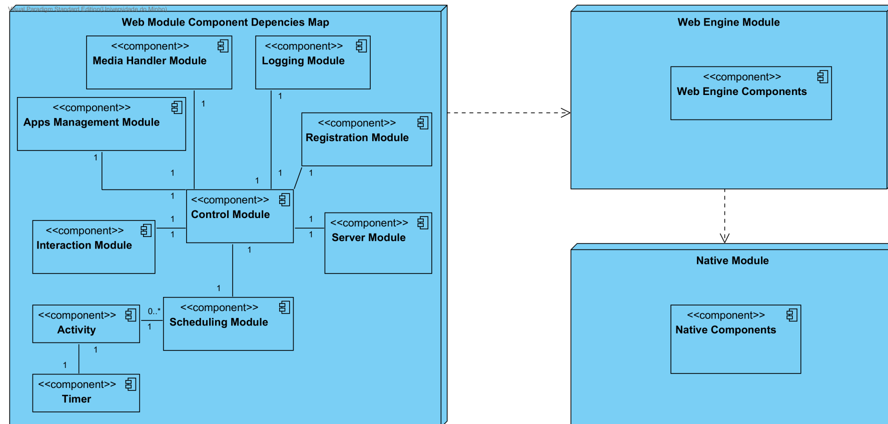


Figure 5.: Generic player's architecture

This modular architecture presents a solutions to the difficulties inherent to the creation of a software deployable across a wide range of systems. Allowing the modules to be interchangeable it provides the possibility to create players deployable on a huge amount of systems by reusing already implemented modules and creating new ones to fit the needs of the new system.

3.7 SCHEDULING PRINCIPLES AND FORMAT

This section will cover all the aspects of the scheduling principles and the format of the schedules that are received and interpreted by the player. On the appendices of this document a full schedule file can be analysed. Each schedule is represented by a JSON object (Group (2014)), an example schedule can be seen on Listing 3.1, it contains information about the schedule itself, the list of applications that will be used by that schedule and the instructions on how, when and for how long to display the applications.

```
{
        "schedule": {
            "id": "sche2",
            "version": "1.2",
            "name": "test schedule",
            "LastUpdate": "15-12-2013",
            "playerId": "001",
            "apps": [... ],
            "content": {
                "rootContainer": "layout",
                "repeatCount": "1",
                "childElements": [.....]
            }
        }
  }
```

Listing 3.1: Example schedule.

The schedule contains information about itself, Listing 3.2, this is used to identify the schedule, its version, name, when it was last updated and which player will be using the schedule. All this is needed for the synchronization process that happens between the player and the server.

```
            "id": "sche2",
            "version": "1.2",
            "name": "test schedule",
            "LastUpdate": "15-12-2013",
            "playerId": "001",
```

Listing 3.2: Extract of a schedule.

The schedule also contains information about the applications that will be displayed during its execution. Each entry of the application's list, Listing 3.3, is composed by: the application's Id, which is a system wide identifier every player displaying this application uses the same Id to identify it; the application's URL where it is hosted; and finally the last time it was updated, this is used by the player to verify the validity any content that might have been pre-downloaded by the player.

```
"apps": [
      {
          "appId": "app001",
          "src": "http://testapplications.displr.com/feeds/?sources=
              http://feeds.jn.pt/JN-Destaques&num=10&delay=20&
              animation=fade&placeId=8",
          "lastUpdate": "15-12-2013"
      },
      {
          "appId": "app002",
          "src": "http://testapplications.displr.com/facebook/index.
              html?type=photo&fbid=spiritocupcakes&title=
              SpiritoCupcakes&delay=10&num=10&placeId=8",
          "lastUpdate": "15-12-2013"
      },
      {
          "appId": "app003",
          "src": "http://testapplications.displr.com/twitter/index.
              html?delay=15&count=12&screenName=cmjornal&method=1&
              placeId=8",
          "lastUpdate": "15-12-2013"
      }
],
```

Listing 3.3: List of applications on a schedule.

Finally, the schedule contains the presentation information, Listing 3.4. This information is mapped into a tree of containers, each container can encapsulate as many others as necessary and have as many levels as necessary.

```
"content": {
    "rootContainer": "layout",
    "repeatCount": "1",
    "childElements": [
                {
                    "container": "app001",
                    "duration": "25",
                    "left": "0",
                    "top": "0",
                    "width": "1",
                    "height": "0.5",
                    "minWidth": "0",
                    "minHeight": "0"
                },
                {
                    "container": "app002",
                    "duration": "25",
```

```
                            "left": "0",
                            "top": "0.5",
                            "width": "0.5",
                            "height": "0.5",
                            "minWidth": "0",
                            "minHeight": "0"
                        },
                        {
                            "container": "app003",
                            "duration": "25",
                            "left": "0.5",
                            "top": "0.5",
                            "width": "0.5",
                            "height": "0.5",
                            "minWidth": "0",
                            "minHeight": "0"
                        }
                ]
            }
        }
```

Listing 3.4: Tree mapping of a schedule.

A container can have three different types: sequence, layout and selector. A sequence, Figure 6, represents a series of content, this content can be applications or other containers, which are to be displayed one after the other, following the order present on the schedule.

Figure 6.: Sequence of containers and/or applications

A layout, Figure 7, is a set of containers or applications that are to be displayed at the same time on the screen, the duration of each must be equal to all the others.

Figure 7.: A layout, can be composed of containers or applications

Finally, the selector, Figure 8, is a composed by a list of containers or applications that will be displayed following a set of rules.

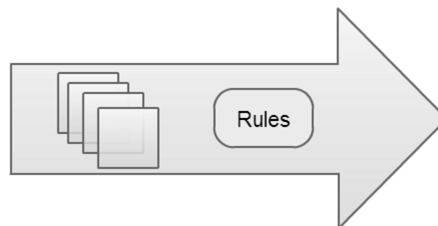


Figure 8.: Selector composed by a list of containers and/or applications and a set of rules

Besides the container type, there is also present a series of other properties: the minimum height and width; the desired height, width and position of the container; the duration, in seconds, of how long that container should be displayed; and, when applicable, the list of the containers and/or applications encapsulated by that container.

3.8 COMMUNICATION AND PLAYER SERVICES

This section will cover all the interaction with the server: the services that are required by the player in order for it to work properly and the communications protocol used to send information to and from the player.

3.8.1 *Player Services*

Player Initiated Communications

These communication processes are initiated by the player and can be started at any time.

FIRST TIME INITIALIZATION

This process is performed whenever a player is intialized by the first time or the first time after a system reset.

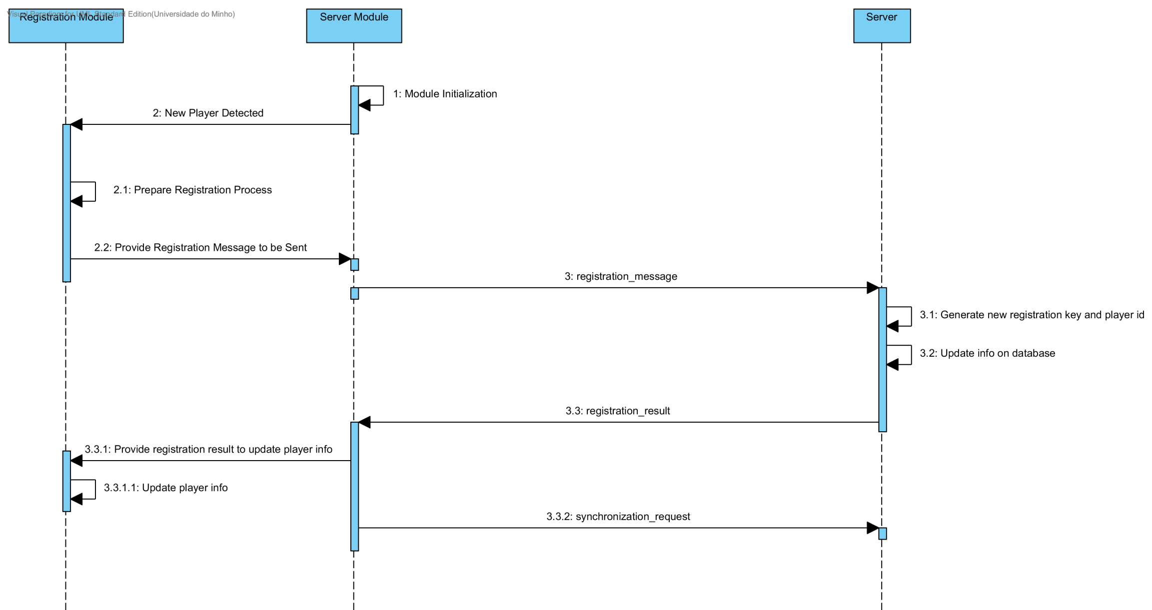


Figure 9.: A player undergoes this process when it is deployed

REGULAR SYNCHRONIZATION

The player periodically sends a synchronization message, requesting the server to check for a schedule update and to inform the server of the player's current state.

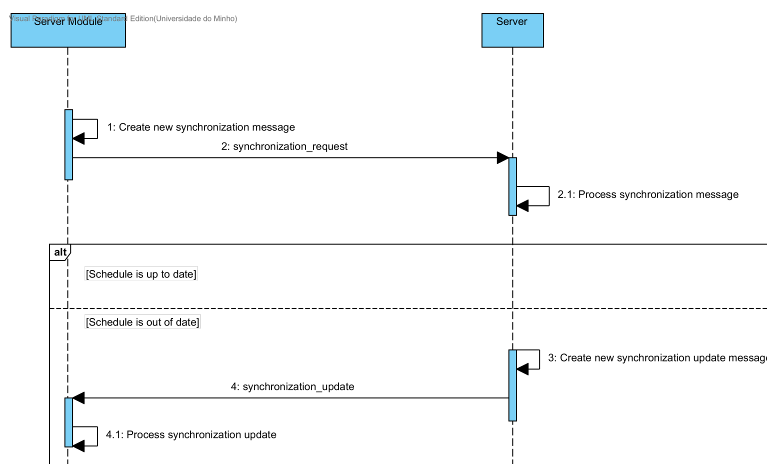


Figure 10.: Periodic synchronization process

LOGGING REPORTS

During the execution of a schedule, the player periodically sends the log entries created so they can be stored on the server.

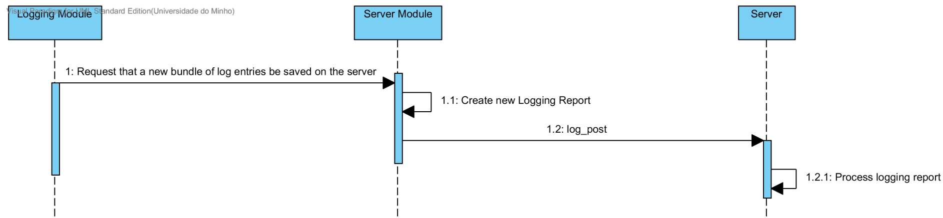


Figure 11.: Process of posting logs to the server

PLAYER RESTART

Whenever the player comes online it is necessary to check it's registration state and if it is not registered a registration request is sent otherwise a synchronization request is sent.

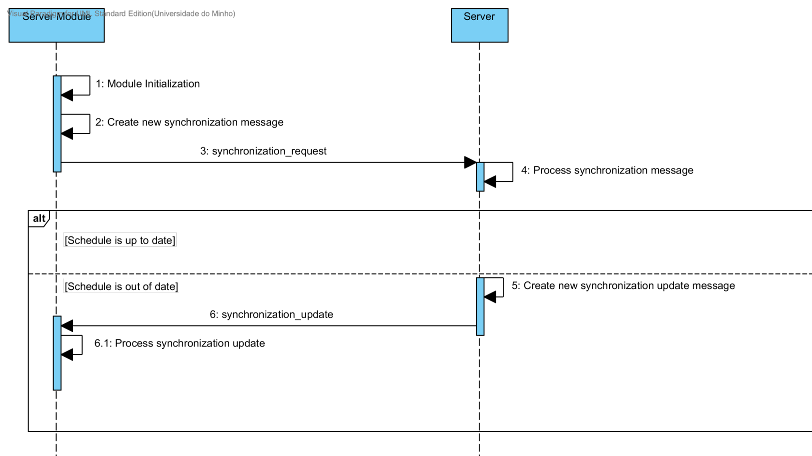


Figure 12.: Player restarting process

SYSTEM RESET

Following a system reset order issued by the user, the player sends an unregistration request to the server and clears any player information present.

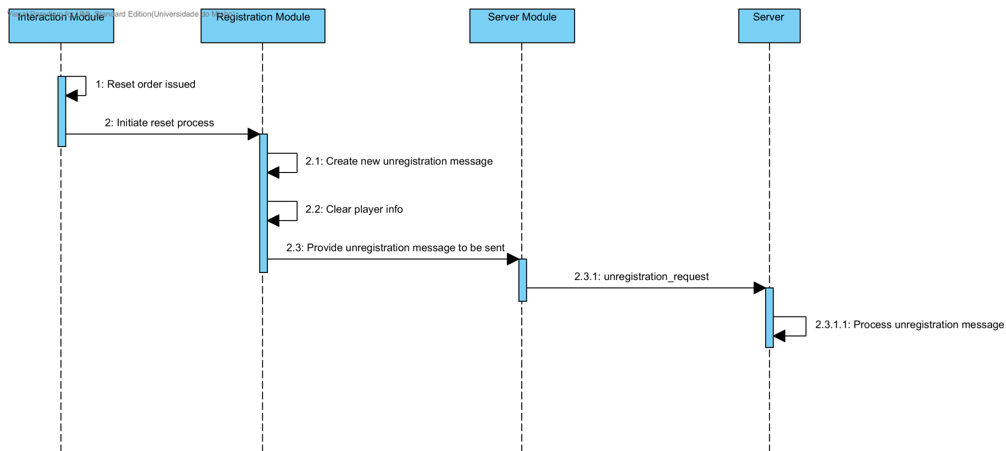


Figure 13.: Process of unregistering a player and reseting it to factory specs

Server Initiated Communications

These communications are dependent on the existence of an open websocket between the player in question and the server. If the player is not currently available, the server will store the message to be sent as soon as the player establishes a connection.

DOMAIN CHANGE

This message is sent by the server when the player's domain is changed by the user.

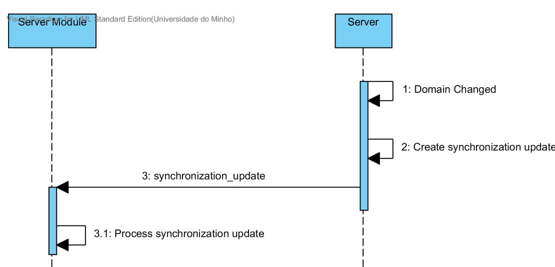


Figure 14.: Process of changing a player's domain

SCHEDULE UPDATE

When the schedule is changed, the server automatically sends a synchronization request to the player associated with that domain.

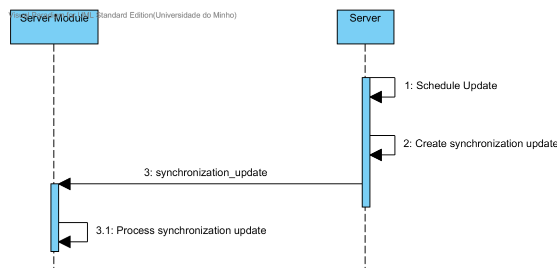


Figure 15.: Process of updating a schedule on the server

3.8.2 *Communication Protocol*

The communication protocol between the player and a control service is based on WebSockets (Corporation (2014)). The format of the messages that support the various interactions between player and server is based on an already defined specification (P2P-SIP (2014)).

Messages Overview

Table 4 displays a list of the messages supported by the player.

Message Type	Direction	Purpose
Logging Post	Player to Server	Send a list of logging events
Registration Request	Player to Server	Request the registration of the player
Registration Result	Server to Player	Send the information necessary to complete a registration process
Unregistration Request	Player to Server	Request the unregistration of the player from the server
Synchronization Request	Player to Server	Request a synchronization of the schedule present on the player
Synchronization Update	Server to Player	Send a notification when a change in the player status occurs (schedule, domain, etc.)
Error Message	Server to Player	Send a notification when an error occurs during communication

Table 4.: Messaging protocol

Most of the messages sent by the player to the server follows a very specific format, composed by four properties:

- method: identifies the http method necessary to be called on the server;

- url: identifies the specific url that can handle this request;
- format: displays what format the contents of the body property is using to make sure the server knows how to handle it, this format follows a syntax based on Multi-purpose Internet Mail Extension (MIME) types (Wikipedia (2014));
- body: contains the information needed to handle the request represented by the message.

The messages sent by the server as a response to the player usually follow a very specific JSON format (Group (2014)) containing the sufficient information needed to handle the message.

Registration

These messages are used when handling the deployment of a new player to notify the server of its existance.

REGISTRATION REQUEST

This message is sent by the player the first time it is initialized, it notifies the server of the existance of a new player and cause the server to generate a new registration key and playerId.

```
{
  "method": "POST",
  "url": "/player/reg
}
```

REGISTRATION RESULT

This message sent by the server as a response to a registration request, it is a JSON object composed of the registration key and the player id generated during the registration proccess.

```
{
  "type": "registration",
  "registration" : {
    "key": "123123",
    "playerId": "123123",
    "schedule": {json_schedule},
  }
}
```

CLAIMING OWNERSHIP

When the ownership of a player changes the server automatically sends a synchronization update containing the new schedule. In case the server does not have a websocket open with the specific player, this information will be requested by the player the next time it connects with the server

Unregistration

This message is used when a player requests to be unregistered from the server, essentially reseting the player to it's factory settings.

UNREGISTRATION REQUEST

The JSON object representing an unregistration request sent by a player as the default player to server message structure defined on this communications protocol, the body property constains the playerId referent to the player that sent the message.

```
{
  "method": "DEL",
  "url": "/player/unreg/{playerId}"
}
```

Synchronization

These messages are used to update the schedule on the player. This message will also be used to determine the status of the player, considering this message will be sent on regular intervals the server can determine either the player is active or not by checking the last time it received a synchronization request.

SYNCHRONIZATION REQUEST

This message is sent by the player to the server, it contains information about the schedule that is being currently and information about the status of the player.

```
{
  "method": "POST",
  "url": "/player/sync/{playerId}",
  "format": "sync/json",
  "body" : {
    "status": "active",
    "domain": "domain1"<GUID>,
    "scheduleId": "sche1",
    "scheVersion" : "v1"
  }
}
```

SYNCHRONIZATION UPDATE

This message is sent by the server to the player whenever there is a change in the player status, as seen by the server. This may include a change of domain or a change in the current version of the schedule. The message contains an indication of the changes, e.g. new domain or new schedule and also the new data that the player needs. In case of a new domain, the message includes the name of the new domain. In case of a as well as the new schedule, the message includes the new schedule itself. Upon receiving this message the player acts accordingly and extracts the new schedule from it.

```
{
  "type": "synchronization",
  "synchronization" : {
    "domain": "place1",
    "scheduleVersion": "v2",
    "scheduleId": "sche2" || null,
    "schedule": {json schedule} || null
  }
}
```

Log Post

This message is sent by the player to the server when it is necessary to save one or more log entries on the data base, the body property of this message can be composed by either one log entry or a list of log entries.

```
{
  "method": "POST",
  "url": "/player/log/{playerId}",
  "format": "log/json",
  "body" : [
"logEntries" {
        "scheduleId":  "sche1",
        "timeStamp": "123456789",
        "level":  "TRACE",
        "message":  "Schedule started."
      },
{
        "scheduleId":  "sche1",
        "timeStamp": "123456789",
```

```
        "level":  "TRACE",
        "message":  "Schedule started."
      }
    ]
  }
}
```

Error Message

This message is sent by the server to the player when something wrong occurs while trying to process a request from a player.

```
{
  "type": "error",
  "error" : {
    "output": "output_string"
  }
}
```

3.8.3 *State Diagrams*

The following diagrams display the possible states assumed by the player during execution, from the moment it is deployed and a registration request occurs to an unregistration request and subsequent reset.

Player as seen by the server

Figure 16 illustrates how the server classifies the player based on the requests the player sends and its changes on the schedule and/or domain.

Player

Figure 17 illustrates how the player sees itself and changes its internal state based on the messages sent by the server.

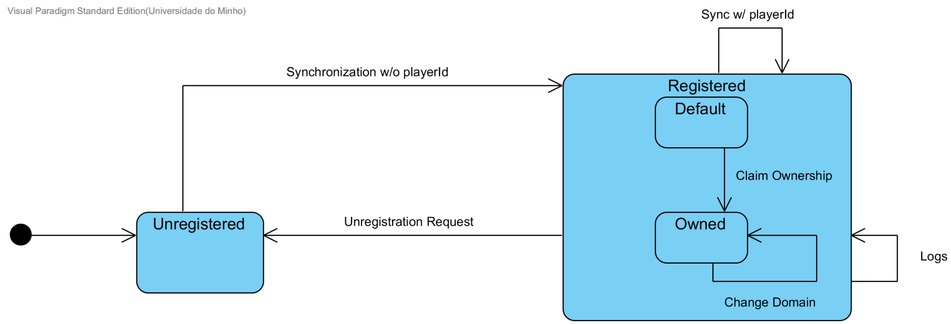


Figure 16.: Player state as seen by the server

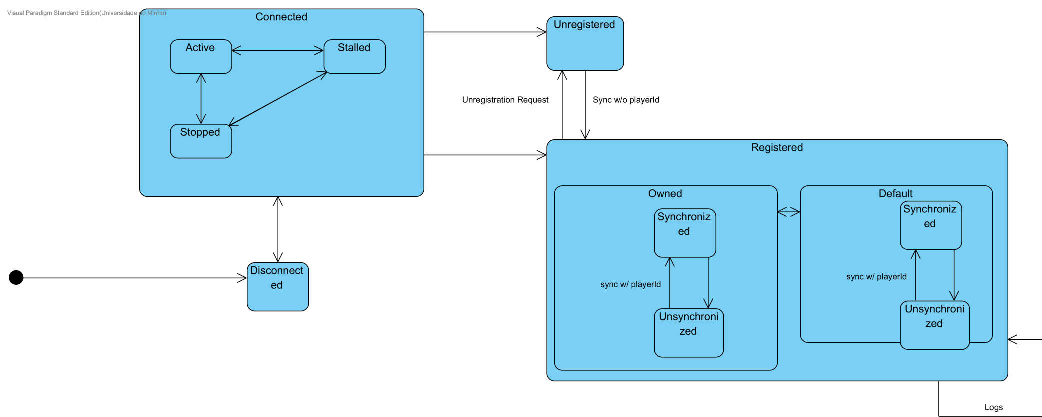


Figure 17.: Player state as seen by itself

3.9 LIBRARIES

The libraries presented in this section represent the web player API, it is the most important of the APIs that were developed and the only one that interacts with outside content. This API was design in two libraries, one loaded by the player (through the web layer) and one loaded by the applications.

3.9.1 *Applications Life Cycle*

The following Figure 18 illustrates the life cycle of an application when it is scheduled to be displayed by the player and how each of the API calls interact with the apps.

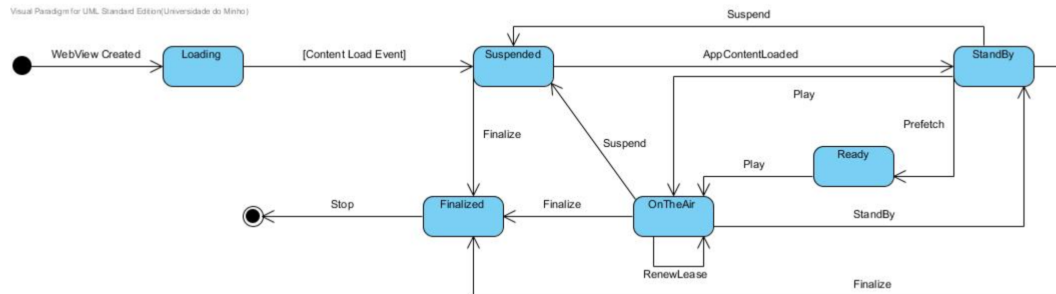


Figure 18.: Applications state diagram

3.9.2 *Player Specific Library*

To allow the player some degree of control over how the applications are being executed it will have access to an API that provides methods to send commands to those applications, parallel to this, the applications have access to a complementary API to inform the player of its internal state.

The player specific library, which is loaded by the player, offers methods for the player to manage the state and execution of the applications being scheduled during their lifetime.

WEB PLAYER COMMUNICATION API

- addApp - called when a new frame is created to encapsulate a new application, the application name, the frame id, the source url of the app and optional parameters can be passed;

```
var messenger = new CommPlayer();
messenger.addApp("appname", "frameId", params, "source");
```

- play - called by the player when it is necessary to start displaying the application content of the application encapsulated on the frame identified by frameId;

```
var messenger = new CommPlayer();
messenger.play("frameId", params);
```

- prefetch - called by the player to notify the app that it has time to prepare the content to be displayed soon;

```
var messenger = new CommPlayer();
messenger.prefetch();
```

- standBy - called by the player when it is necessary to pause the execution of the application encapsulated on the frame identified by frameId;

```
var messenger = new CommPlayer();
messenger.standBy("frameId", params);
```

- finalize - called by the player when it is necessary to stop the execution of the application encapsulated on the frame identified by frameId;

```
var messenger = new CommPlayer();
messenger.finalize("frameId", params);
```

- appList - returns the applications list currently being managed by the library;

```
var messenger = new CommPlayer();
messenger.appList();
```

- subscribe - subscribes a predefined set of actions to trigger when the application encapsulated on the frame identified by frameId changes state.

```
var messenger = new CommPlayer();
messenger.subscribe("frameId", params);
```

3.9.3 *Applications Specific Library*

This library is loaded by each application that is to be displayed on the current schedule. It provides the application with methods to notify the player of changes in the application state and a way for the player to call the methods subscribed by the application.

APPLICATIONS COMMUNICATION API

- subscribe - informs the library of which callback implemented by the application is subscribed to which method (start, standBy, stop);

```
var messenger = new Communication({ appname: 'helloapp', app: this});
messenger.subscribe("method", callback);
```

- isOnTheAir - called by the application to notify the player that the application is displaying content;

```
var messenger = new Communication({ appname: 'helloapp', app: this});
messenger.isOnTheAir(params);
```

- isReady - called by the application to notify the player that all displayable content is ready and there is nothing else for the application to do before going on the air;

```
var messenger = new Communication({ appname: 'helloapp', app: this});
messenger.isReady(params);
```

- isStandBy - called by the application to notify the player that the application has downloaded all the necessary resources;

```
var messenger = new Communication({ appname: 'helloapp', app: this});
messenger.isStandBy(params);
```

- isStopped - called by the application to notify the player that the application is stopped and will no longer display any content from this point forward;

```
var messenger = new Communication({ appname: 'helloapp', app: this});
messenger.isStopped(params);
```

- isSuspended - called by the application to notify the player that the application is not ready to display content, the application assumes this state when it is still downloading resources or something required the application to stop abruptly.

```
var messenger = new Communication({ appname: 'helloapp', app: this});
messenger.isSuspended(params);
```

3.10 SUMMARY

This chapter covered all details of the specification and generic implementation of the player. In order to have a valid and robust piece of software it was necessary to identify the systems it would target. Even with the compatibility options of web technologies it is necessary to focus on four key systems and work towards having those implementations of the player working seamlessly. To reach that goal a set of key functionalities and system requirements were identified, these aimed at providing the player with the tools necessary to properly execute the tasks it was created for.

Having in mind that the player would need to keep most of its key functionalities when it was deployed in different platforms, a modular architecture was designed, this allowed the player to be deployed on distinct platforms while changing only the strictly necessary modules.

A scheduling format was also designed in order to provide the player with the information needed to display whatever type of schedule a user might want. Finally, to keep the player synchronized with the current schedule and to have some degree of control over the player, a communication protocol and a set of player services were created to be deployed on a server.

4

CASE STUDIES

In this chapter all the different case studies of implementations of the player are going to be analysed. This modular architecture didn't happened by chance, it was the result of an iterative research started before this work that culminated in the current architecture. The goal of this architecture is to allow an easy replacement of a module to facilitate the implementation of new versions of the player. All the modules that serve as the building blocks for each of the implementations of the player will be thoroughly examined.

Being the base implementation of any version of the player, the Displr case study will be the first to be covered in this chapter. Considering all other versions derive from the Displr case study, for each of the other case studies only the modules that had to be reimplemented will be analysed.

These modules were all developed using the requireJS framework, this framework allows to easily implement and manage any extremely modular application such as the web based player.

4.1 CASE STUDIES AND THEIR TARGETED PLATFORMS

In the first stages of this investigation, it was necessary to identify a set of reference stacks to narrow the platform/system combinations the web based player would be able to target. After developing a prototype framework for the player it was possible to start thinking of applying it to some of those platforms by implementing different versions of the player to meet the needs of several case studies.

The solutions created for those case studies were both to upgrade older, more restrictive digital signage software into something more ubiquitous and less system dependant and to develop solutions to brand new platforms in an effort to augment the reach of an already existing digital signage service.

During this stage it was possible to create solutions for Windows, Linux and MacOS based platforms, Android and iOS platforms and Samsung SmartTVs (Samsung (2014b)).

4.2 DISPLR

Displr is a service that aims at taking the user interactive experience with public displays to a whole new level. The service revolves around the content that's being displayed and how the user

interacts with it using his smart phone. The content being displayed will feed a content stream that in turn allows the user to actively influence what is displayed on screen by interacting with the Displr app on his smart phone.

The Displr case study is the basic implementation of the web player: a Chrome Packaged App composed by the modules displayed on the player's architecture presented on the previous chapter. This implementation is ready to communicate with the Displr service and consume the services it provides. It is prepared to be deployed on all systems that can run a desktop version of Google Chrome v32 or greater.

In this specific case the platform used will be a PC running an Ubuntu distribution.

4.2.1 *Control Module*

The control module, Figure 19, is the central module of the framework. It instantiates all other modules and serves as a communications bridge between them. It is also responsible for obtaining some system info such as the resolution of the screen that is displaying the content.

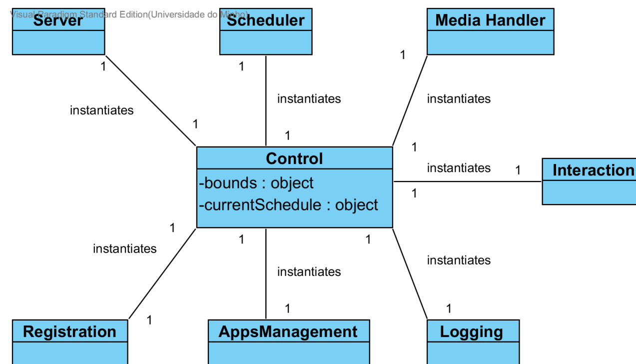


Figure 19.: Control module class diagram

4.2.2 *Server Module*

The server module, Figure 20 is responsible for handling all messaged received and sent. It contains info about the player: its id, the registration key used to claim ownership over the player and the domain to which the player is associated.

Being the module responsible for communications it uses the socketIO API (Contributors (2014)) to create, open and manage a socket connection to the server. The server's address must be defined within the connString variable.

API

This module implements three different methods that are called by the control module:

- init

  The init method is called by the control module during the players initialization. It prepares the server module for execution by opening a socket connection, requesting the registration module to check the player state and, when everything is ready, it starts the schedule synchronization periodic requests.

  ```
  var serverModule;
  serverModule.init(loggingModule, currentSchedule, schedulingModule,
      registrationModule);
  ```

- close

  This method simply closes the current socket connection with the server.

  ```
  var serverModule;
  serverModule.close();
  ```

- saveLogs

  This method is called to send a batch of log entries to be saved on the server, this is done periodically by the logging module.

  ```
  var serverModule;
  serverModule.saveLogs(logBundle);
  ```

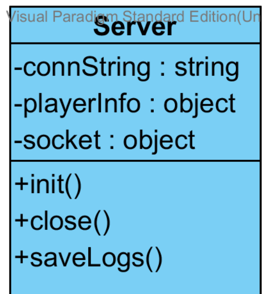


Figure 20.: Server module class diagram

4.2.3 *Registration Module*

The registration module, Figure 21, contains all the logic related to the player registration, unregistration and player information updates. It checks the player registration status during the player's initialization and acts according to the information stored on the system, by either requesting a registration of the player, when a player is first deployed, or simply by telling the server module everything is ready and the schedule synchronization should start when a player is already registered on the server.

API

- checkState

  The checkState method is called by the server module when the player is initialized, it checks the local storage for information about the player and acts according to what is found: if there is a valid player id, the registration module tells the server module everything is ready and the schedule synchronization should start; if there is no player id present or it is not a valid one (this can happen when the player is initialized but cannot communicate with the server) this module issues a registration request.
  The registration request, already explained in the previous chapter, requests the server to generate a new id for this player and a new registration key.

  ```
  var registrationModule;
  registrationModule.checkState(socket, playerId, loggingModule);
  ```

- updateRegInfo

  This method is called by the server module when it receives a registration result message, it updates the player information with the new information contained in the message object and saves it on the local storage.

```
var registrationModule;
registrationModule.updateRegInfo(messageObj, playerId);
```

- unregister

  The unregister module simply resets the player to factory specifications, it clears all information present on local storage and generates an unregistration request to be sent to the server so it is updated on the status of the, now reset, player.

```
var registrationModule;
registrationModule.unregister();
```

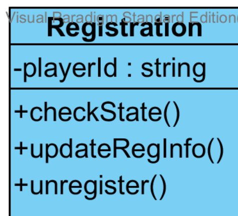


Figure 21.: Registration module class diagram

4.2.4 *Scheduling Module*

The scheduling module, Figure 22, is responsible for creating, initializing, managing and terminating all the activities. Each of these activities represents a schedule received from the server.

The module contains a list of all the activities that are queued to be executed and an object representing the activity that is currently being executed. It offers methods to initialize and manage these activities, they are usually called by the server module when a new schedule is received and by the interaction module when the user needs to interact with the player's behaviour.

API

- init

  This initializes the module by passing to it the instances of the modules that are required. The currentSchedule object is also passed so that it can be managed by the scheduling module.

```
var schedulingModule;
schedulingModule.init(mediahandlerModule, appsmanagementModule,
    loggingModule, currentSchedule);
```

- newActivity

  This initializes a new activity based on the json object passed as argument. It creates the scheduling tree, requests that the activity create its own eventTable and queues it on the activity list.

```
var schedulingModule;
schedulingModule.newActivity(scheduleJSON);
```

- pauseCurrent

  Pauses the execution of the current activity. The content being displayed continues with its execution but the player will not advance on the event table until it receives a resume order.

```
var schedulingModule;
schedulingModule.pauseCurrent();
```

- resumeCurrent

  Can only be called when an activity is paused, doing so will make the player resume its normal execution of the activity.

```
var schedulingModule;
schedulingModule.resumeCurrent();
```

- stopCurrent

  Stops the current activity, can be called at any time when an activity is being executed. This forces the player to stop the execution of the activity and start the execution of the next activity in the queue, if there is any.

```
var schedulingModule;
schedulingModule.stopCurrent();
```

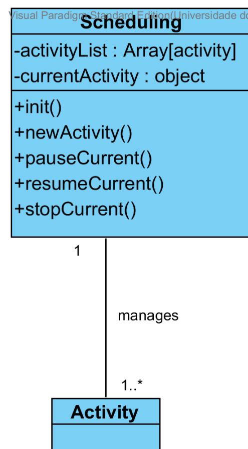


Figure 22.: Scheduling module class diagram

4.2.5 *Media Handler Module*

The media handler module, Figure 23, is responsible for creating, managing and destroying the frames that execute all the content described in each schedule. It contains a list of the frames that currently exist and offers a range of frame manipulation methods, these are mostly called by the activities during their execution.

These frames are WebView HTML elements Google (2014b) and, as stated before, they are a Chrome specific API.

API

- init

  The initialization method is used to provide the media handler module with the tools needed to perform its job. This method receives as arguments the instance of the logging module and the bounds object from the control module.

  ```
  var mediahandlerModule;
  mediahandlerModule.init(loggingModule, boundsObj);
  ```

- requestFrame

  The requestFrame method is used by the activities to, like the name suggests, request a frame to execute a piece of content in. When this method is called the media handler module checks the frame list for an already existing frame, currently not being used, with the characteristics needed and if there is one, that frame is returned otherwise, if no such frame exists the module

creates a new one using the createFrame method. Upon reaching a maximum number of frames permitted, this method will start destroying the oldest, not being used frame in the list.

```
var mediahandlerModule;
mediahandlerModule.requestFrame(width, height, left, top, persistent,
    containerId, appId);
```

- createFrame This method complements the requestFrame method, it does not check the frame list to try and recycle an old frame, it simply creates a new one if the maximum number of frames wasn't reached.

```
var mediahandlerModule;
mediahandlerModule.createFrame(width, height, left, top, persistent,
    containerId, appId);
```

- releaseFrame

  This method is called by the activities during their execution and notifies the media handler module that a frame is no longer needed and can be reused. The module then updates that fram'es information for future use.

```
var mediahandlerModule;
mediahandlerModule.releaseframe(frameId);
```

- resetData

  This method is called once per activity execution, when the activity naturally finishes its execution or is manually stopped. Upon being called, the resetData method will clear all the frames from the list and destroy the html tags represented by those frames.

```
var mediahandlerModule;
mediahandlerModule.resetData();
```

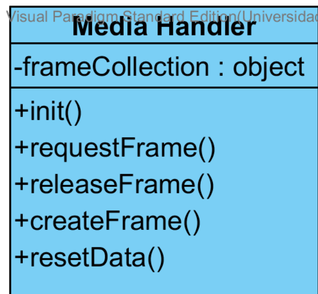


Figure 23.: Media Handler module class diagram

4.2.6 *Apps Management Module*

The apps management module, Figure 24, contains and manages information about all the content that was displayed on that player. It implements a list of applications (content), each of these applications is an object containing information regarding the application it represents: id, the source URL, if the application is cached, the number of times it was played by this player, the last time it was played and its current state (being played, pre fetched or stopped). This information is updated by the activities during its execution using the methods this module implements and stored on using the ChromeStorage API Google (2013).

API

- load

  The load method, is called by the control module during the player's initialization. It checks the local storage for any information previously saved regarding the applications that were executed by this player, if anything is found it is loaded into the application list.

```
var appsmanagementModule;
appsmanagementModule.load();
```

- save

  Called every time a schedule stops being executed, this method saves the current state of the application list on the local storage.

```
var appsmanagementModule;
appsmanagementModule.save();
```

- updateList

  This method is used to add new applications to the application list. When the scheduling tree of an activity is created, the scheduling module checks the app list of the schedule that generated that activity, if any new app is found, the scheduling module adds it to the apps management module application list.

```
var appsmanagementModule;
appsmanagementModule.updateList(appId, appSrc);
```

- updateApp

  During the execution of an activity it is necessary to keep the information the application list updated. The activity uses this method to update that information on the list according to which operation is being perform on the application (play, stop or pre fetch).

```
var appsmanagementModule;
appsmanagementModule.updateApp(appId, status);
```

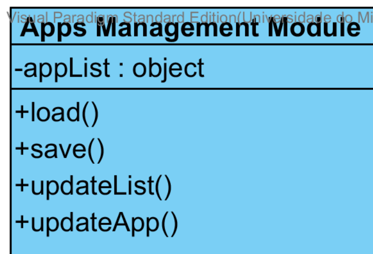


Figure 24.: Apps Management module class diagram

4.2.7 *Logging Module*

The logging module, Figure 25, was implemented based on the specification of the log4JS framework Down (2014). It offers methods to log the behaviour of the player depending on the severity of the message.

The module contains a list of log entries that are periodically sent to the server to be saved.

API

- init

  This method receives the object that represents the batch of log entries that have to be saved, this object is shared with the server module so it can be sent to the server.
  This method also starts the periodic request to save a new batch of logs, if there are any.

  ```
  var loggingModule;
  loggingModule.init(logBundleObj);
  ```

- trace

  This method allows the player to log a message with the trace severity, this kind of messages are usually generated during the execution of an activity.

  ```
  var loggingModule;
  loggingModule.trace(message, scheduleId);
  ```

- info

  The info message are used to log generic events of the player's execution: initialization, connection to the server, messaged received and sent and similar events.

  ```
  var loggingModule;
  loggingModule.info(message);
  ```

- error

  The error messages, just like the name suggests, are used to log unexpected behaviour of the player such as a failure to parse a schedule, or a drop of the socket connection.

```
var loggingModule;
loggingModule.error(message);
```

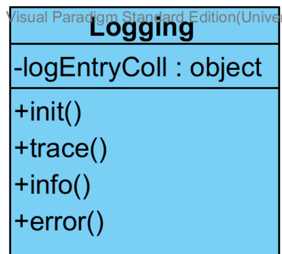


Figure 25.: Logging module class diagram

4.2.8 *Interaction Module*

The interaction module, Figure 26, generates the user interface that allows the user for some external control over the behaviour of the player. Using the interface provided by this module, a user can pause, resume or stop the execution of the current activity.

API

- init

  The init method is called during the player's initialization, it orders the module to generate the HTML elements that compose the UI and inject them into the base frame of the player. It is also used to pass, as argument, the necessary modules and bounds object present on the control module.

```
var interactionModule;
interactionModule.init(loggingModule, boundsObj, schedulingModule);
```

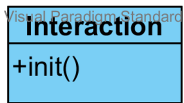


Figure 26.: Interaction module class diagram

4.2.9 *Activity*

The activity class, Figure 27, represents a schedule, it contains all the instructions necessary to display the content present there and is able to parse it into an event table in order to be played.

The event table created based on the instructions present on the schedule that originated the activity is travelled, one position at a time, according to the ”ticks” generated by the timer class. Each ”tick” received represents a position on the event table array, during the execution, the activity will check the event table for any events on that specific position and act accordingly.

The events can be of one of three different types: pre-fetch, play and stop. During the pre-fetch of an app, its frame is created, hidden, and the app is allowed to prepare itself by downloading any necessary information until the play event is fired. When the play event for that app is launched, that frame’s visibility is changed to visible until the stop event is fired. When that happens the frame is again hidden and its status is updated on the media handler module until it is requested by the activity to be played again or the frame is either recycled or destroyed.

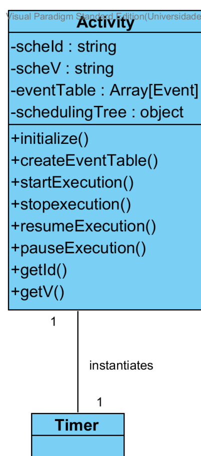


Figure 27.: Activity class diagram

On the following Figure 28 the full scheduling process can be seen in detail.

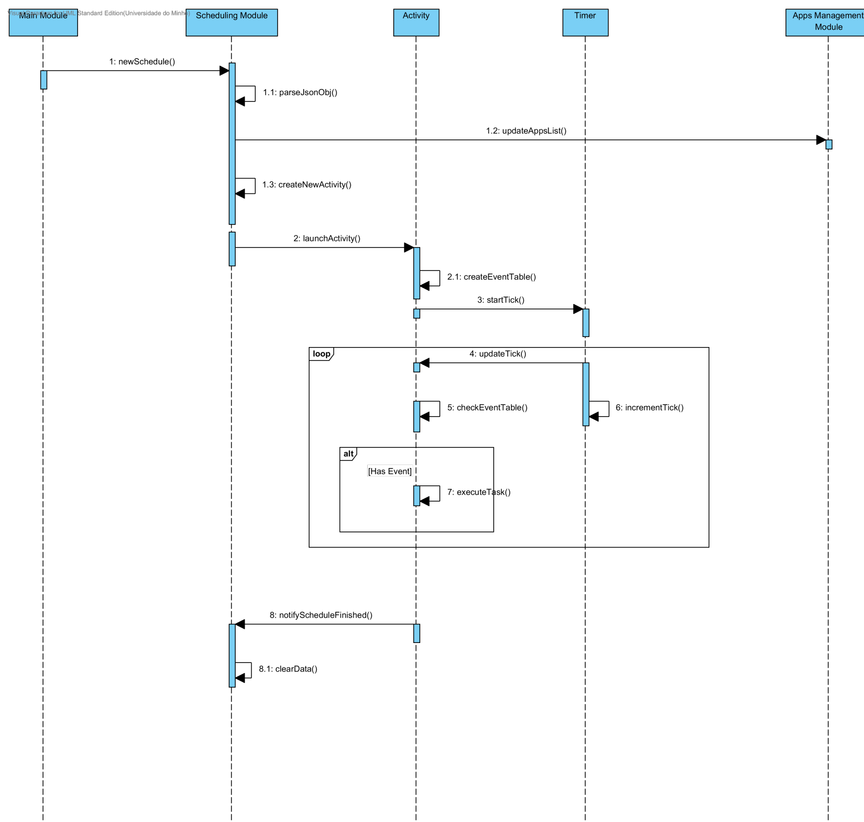


Figure 28.: Scheduling sequence diagram

API

- createEventTable

  This method transforms the information from the scheduling tree into an event table. This event table maps the pre-fetch, play and stop events for each of the applications (or content pieces) to positions on an array. These positions will be iteratively travelled according to the ”ticks” generated by the timer class.

```
var activity = new Activity(schedulingTree, mediahandlerModule,
    appsmanagementModule, loggingModule);
activity.createEventTable();
```

- startExecution

  This method notifies the timer to start sending ticks to the activity and starts checking the event table positions for events. A currentActObj is passed as argument, this object is used to keep track of which activity (or schedule) is currently being played.

```
var activity = new Activity(schedulingTree, mediahandlerModule,
    appsmanagementModule, loggingModule);
activity.startExecution(currentActObj);
```

- stopExecution

  This is the counter part of the previous method, the stopExecution method tells the timer to stop its work and performs the tasks connected to the end of an activity: reseting the media handler module data and saving the information present on the apps management module.
  This method is both called when the activity naturally reaches its conclusion or when a user intentionally requests the current activity to be stopped.

```
var activity = new Activity(schedulingTree, mediahandlerModule,
    appsmanagementModule, loggingModule);
activity.stopExecution(currentActObj);
```

- pauseExecution

  The pauseExecution method can only be called by a user and simply tells the timer to stop generating ”ticks” until further notice. This does not reset the ”ticks” count like the stopExecution does and when necessary the execution can be resumed by called the following method.

```
var activity = new Activity(schedulingTree, mediahandlerModule,
    appsmanagementModule, loggingModule);
activity.;
```

- resumeExecution

  Like stated above this method is called to recover the normal execution after a pauseExecution request is issued. This notifies the timer to resume the generation of ”ticks”.

```
var activity = new Activity(schedulingTree, mediahandlerModule,
    appsmanagementModule, loggingModule);
activity.resumeExecution();
```

- getId

  Returns the id of the schedule that generated this activity.

```
var activity = new Activity(schedulingTree, mediahandlerModule,
    appsmanagementModule, loggingModule);
var scheduleId = activity.getId();
```

- getV

  Returns the version of the schedule that generated this activity.

```
var activity = new Activity(schedulingTree, mediahandlerModule,
    appsmanagementModule, loggingModule);
var scheduleVersion = activity.getV;
```

4.2.10 *Timer*

The timer class, Figure 29, is responsible for generating the “ticks” that allow the activity to move forward on the event table. The “tick” is basically an integer that is incremented by one unit at regular intervals and it is based on SMIL (W3C (2012b)).

This class is instantiated by the activity as a JavaScript worker (W3C (2014)) and they communicate using the cross-window messaging system (W3C (2012a)). When the timer receives the order to start incrementing the counter it creates a periodic event, based on the interval variable, that increments the counter and sends the new value to the activity.

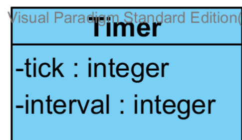


Figure 29.: Timer class diagram

4.3 UBISIGN

The Ubisign case study of the player aims at being deployed on already existing Ubisign digital signage networks. Considering that the player should retain most of its deployment capabilities, only the modules that interface with the network’s back-office or server (server and registration modules) are required to be changed.

4.3.1 *Server Module*

As stated above, on the Ubisign case study, the player is required to connect and communicate to an already existing infrastructure of servers. The Ubisign back-office implements several services in a WPF programming model (Microsoft (2014a)) this forces a change in the socket communication model already implemented on the player.

To allow the communication between the JavaScript player and the WPF server a web service (W3C (2004)) interface was created. The basic services were implemented on that interface and the player could use standard jQuery (jQuery Foundation (2014)) GET requests (RFC 2616 Fielding (2004)) to communicate with it. During an initial stage only a registration and synchronization services were created and tested successfully.

The synchronization service required a bit of finesse to implement as the Ubisign scheduling format was completely different of what the player was expecting. The solution was the implement a eXtensible Style-Sheet Language (XSLT) style-sheet (W3C (1999)) on the web service that would transform an Ubisign schedule into a schedule with the standard scheduling format developed for the web player.

4.3.2 *Registration Module*

The registration module was simpler to port to Ubisign specifications. The logic of the module remained the same only changing the registration message into a format that the WPF service could understand.

4.4 ANDROID AND IOS

The Android and iOS versions are, at this moment, only a proof of concept. The idea is to deploy the player on these platforms using the specific app store for each of the systems, therefore this requires different apps for each of these cases. They will be analysed together due to the fact that for the moment the apps only emulate a webview that runs an HTML5 version of the player deployed on a server.

This HTML5 version runs all the same modules as the Displr implementation except for the ones with visualization and storage capabilities (media handler and apps management), they require different APIs from the ones used on the original Chrome Packaged App since these APIs are chrome specific.

The future of this implementation is moving the execution of the player into the android/iOS apps.

4.4.1 *Media Handler Module*

As stated before, the media handler module is responsible for managing the frames where the content is displayed. The module generates and injects onto the HTML5 base frame the frames as they are created. On the Chrome Packaged App implementation these frames represented WebView HTML elements (Google (2014b)), however, being a Chrome specific API they are not usable on an HTML5 only player.

The solution was to switch the usage of WebViews to iFrames. The iFrame API (Community (2014)) is fully supported by HTML5 and allows the player top have a very similar behaviour to its WebView counterpart.

As mentioned above, the next logical step on this case study is to move the whole execution of the player to the android or iOS app itself, this will, once again, allow the use of WebViews since it is a fully supported API on those environments (Ogden (2012)).

4.4.2 *Apps Management Module*

Much like the media handler module, the apps management module required an API change when migrating the player to android/iOS systems. The standard Chrome Packaged App implementation of the player relied on the chromeStorage API (Google (2013)) to store all the player and applications information, being a chrome specific API it was necessary to rely on the localStorage API (Pilgrim (2011)) for the HTML player.

Since both storage APIs have similar characteristics it was very easy performing the migration to this API.

4.5 SAMSUNG SMARTTV

The Samsung SmartTV case study is aimed at creating a Samsung Smart App that runs the web based player directly on a Samsung SmartTV. This app, like the Chrome Packaged App the Displr version of the player is based on, can be distributed through the Samsung App Store accessible on any Samsung SmartTV.

Much like the android and iOS case study requires a different media handler and apps management modules, exactly for the same reasons as stated above. Besides these two modules, the interaction module also had to be changed to allow for tv remote usage.

4.5.1 *Media Handler and Apps Management Module*

These two modules had to suffer the exact same modifications as the android/iOS case study. The Samsung SmartTV implementation is a straight HTML5 implementation and could not use any of the

Chrome specific APIs.

A small detail with this case study is that in the android/iOS case study, when the player completely integrated into the android/iOS app, it will be possible to once again use the WebView API, this is not true for the Samsung SmartTV.

4.5.2 *Interaction Module*

The interaction module of the Samsung SmartTV player had to suffer a small implementation change in order to accommodate the input of the TV's remote command. However, the Samsung Software Development Kit (SDK) (Samsung (2014a)) offers a library to handle key presses on the remote command, it was only necessary to map these into the already existing user interface oand the player became responsive to user interaction from the remote command.

4.6 SUMMARY

This chapter covered the four main case studies where the web based player framework was applied. Having an extremely modular nature, it is possible to create players that can be deployed on all identified reference stacks.

The migration to different systems is not a straight forward mechanism, the necessity to develop and implement the specific modules to each platform/system combination is still present, however the architecture of the player and most of its characteristics are maintained, furthermore it shouldn't be necessary to replace all the modules when contemplating the identified reference stacks. Most of the already implemented modules can be mix and matched in order to answer the requirements of those systems.

Finally, this chapter attests to the portability of this framework in which, with minimum changes, it was possible to deploy the web based player in several different platform/system combination.

5

TESTING

This chapter will present the planning, execution and results of the testing process. The purpose of these tests is to prove that the player is ready to be deployed on a real digital signage network and can handle without problems all the tasks required from a digital signage player.

The tests were executed on a controlled environment and all the logs and results created during the testing process were thoroughly examined and validated.

5.1 PLANNING

In order to ensure that the player is working as intended and can properly function on a normal working environment a series of tests have to be conducted to validate it. These tests will consist of:

- Scheduling execution tests, they consist on running several dummy schedules and making sure the output log of resulting from the execution of those tests matches the traces for each schedule.
- Schedule precision tests, they are focused on ensuring the timing of the execution of a real schedule is correct and there is no unexpected delays.
- Stress tests, the player will be subject to a battery of tests that aim to place the application under a stressful execution situation to make sure it is capable of maintaining it's functionalities during those periods.
- Memory profiling, this process will involve tests aimed at finding memory leaks and understand what is causing them.
- Unit tests aimed at proving the consistency of the application's code.
- Deployment tests, test the deployment and registration process in various target platforms.

5.2 TESTS

5.2.1 *Schedule Execution and Precision Tests*

The following items describe the steps taken during this process:

- Create dummy schedules and their expected traces;
- Execute the dummy schedules several times while logging the execution;
- Compare the resulting logs with the traces.

For every different schedule a trace file Figure 30 is created containing the expected behaviour of the player while executing the schedule. This file is created when the schedule is parsed. After the execution of the schedule is over, the player saves the log associated to that schedule into a file. This is done by POSTing the trace and log results to a PHP script, which handles the information and creates the files. The usage of PHP rather then JavaScript for handling this information is justified due to PHP's ease of access to the FileSystem. There is also another PHP Script to read all the trace and log files present in the folder and compare each of the schedule log files to it's corresponding trace file. The result of this operation is saved to a file, where every line of a log that does not match to the trace file is enumerated.

```
0 PREFETCH app001
0 PLAY app001
0 PREFETCH app002
20 PLAY app002
20 STOP app001
20 PREFETCH app003
40 PLAY app003
40 STOP app002
40 PREFETCH app004
80 PLAY app004
80 STOP app003
80 PREFETCH app005
110 PLAY app005
110 STOP app004
135 STOP app005
```

Figure 30.: Example of a trace file

Results

After running a number of different schedules and comparing the resulting logs to the traces generated we can conclude that the player is executing the schedules with precision and with the correct behaviour. The PHP script used to compare the trace with the log files, compares line by line each of the files and generates a file Figure 31 with the result of this comparison. On it the information of the schedule ID and which files were compared can be observed. On the appendices of this document more detailed information about the logs and trace files can be found.

```
dummy1 -> LOG_dummy1_1400068181640.txt - TRACE_dummy1.txt
Log and Trace file match.
dummy2 -> LOG_dummy2_1400068189484.txt - TRACE_dummy2.txt
Log and Trace file match.
dummy3 -> LOG_dummy3_1400068208265.txt - TRACE_dummy3.txt
Log and Trace file match.
precision1 -> LOG_precision1_1400073103633.txt - TRACE_precision1.txt
Log and Trace file match.
precision1 -> LOG_precision1_1400081395044.txt - TRACE_precision1.txt
Log and Trace file match.
precision2 -> LOG_precision2_1400073111563.txt - TRACE_precision2.txt
Log and Trace file match.
```

Figure 31.: Example of a result file

5.2.2 *Stress Tests*

The stress tests will be executed in two steps:

- Run the player for a large period of time;
- Run the player with schedules that require a large amount of frames open simultaneously.

This will provide the information necessary to evaluate the player's behaviour under extreme or abnormal situations.

Results

The first portion of these tests was done in conjunction with the memory profiling tests. The player was deployed on a machine and was left running the player, with the same schedule, for a period of one week. During the time, the machine's state was checked regularly and nothing out of the ordinary was found.

Player and machine both handled well the initial value of maximum 20 frames open at any time. These tests were done using the usual tick frequency of 1000ms, to speed up the testing process it was experimented lowering the tick frequency to 100ms. With smaller, lighter schedules the player and machine were able to handle the faster ticking rate. However when a bigger schedule, where 10 frames were required to be open at the same time while prefetching another 10, both the player and the machine were unable to cope. Performance wise it was clear the machine wasn't ready to handle that amount of frames/operations in such a small time window. Player wise it was noticeable a few operations were not executed: a couple of splash screens were not hidden and when the execution of the schedule was over the player failed to clear all the existing frames, leaving a few behind.

This was, however, an extreme case, under normal circumstances it shouldn't be necessary to execute such an amount of operations in such a small time period, during this test the player had to prefetch, play and stop a total of 90 frames under 1 minute.

5.2.3 *Memory Profiling*

These tests will be executed on three diferent levels to ensure any possible memory leaks are identified:

- Apps running on their own;
- Apps running alone on the player;
- Apps running with other apps simultaneously on the player.

The player will also be fitted with a memory profiling tool that will periodically map the memory usage to the player's logs using Chrome's Memory Profiling API.

Results

To log the memory usage Window's Performance Monitor (Microsoft (2014b)) was used, this generated reports such as the following Figure 32.

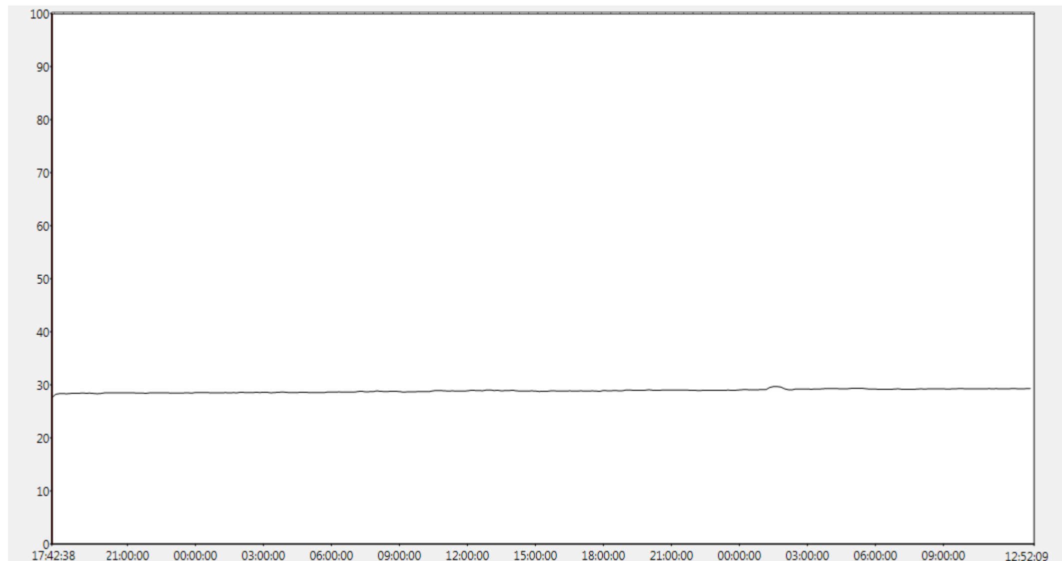


Figure 32.: Memory profilling report

This graph shows the memory allocated by the machine during one of the tests. This particular test was performed using a schedule with real apps for a period of 43 hours on a very controlled environment with only the memory monitor, the player and the system minimum processes using the system's resources. The next test, Figure 33 is a much more real one, it was taken over the course of one week having the player running the same schedule on a PC competing for resources with other processes.

This time the tool used to monitor the memory consumption was the performance monitorization API for Chrome Packaged Apps (Google (2014a)), periodically measuring the amount of used memory and logging that result to a file.

This test shows us a lot of different events during its course, considering this is a shared environment these results are expected. We can observe spikes of used memory, with some periodicity, this was not replicated during the controlled environment tests so it can be ruled out as being caused by other processes. Overall the memory usage did not increment into an unbearable amount so this leads to a conclusion that the player is ready to be deployed on any kind of system. Be it a machine running exclusively the player or a machine that is running other background processes.

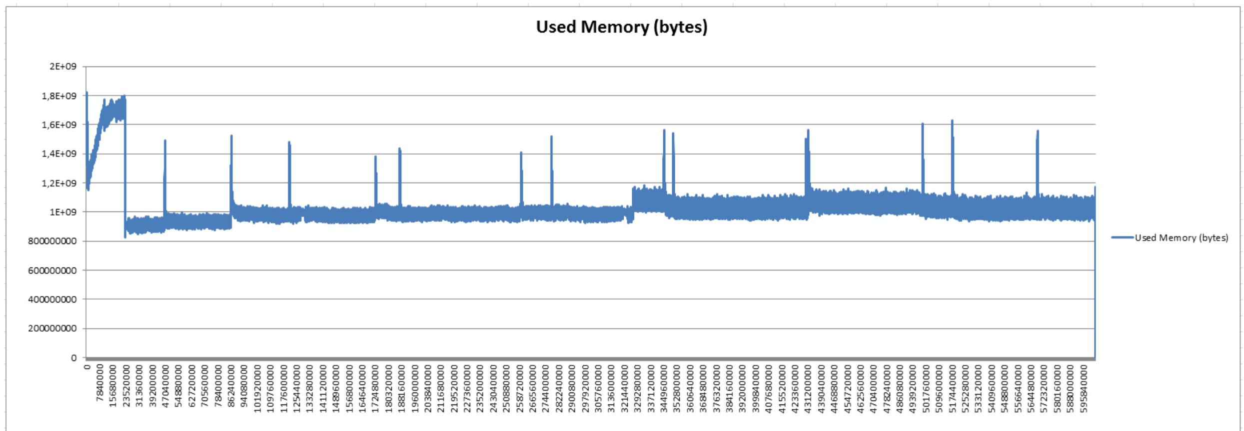


Figure 33.: Memory profilling report over a week

5.2.4 *Unit Tests*

These tests will ensure the code is correct and fit for use, passing these tests with a green light proves that the application is ready to be distributed and used. The testing framework used is Jasmine (Jasmine (2014)). Each of the player's modules will have its own testing suite, containing the atomic tests related to that module. These atomic tests check each of the individual methods used by the module that implements them to ensure they are performing as expected.

Results

After implementing and running the testing suites necessary, each containing multiple atomic tests such the code snippet on Listing 5.1.

```
describe("Reset data clears all existing frames and information.",
    function () {
    it("Empties the framelist.", function () {
        expect(mediaHandler.frameList.length).not.toBe(0);

        mediaHandler.resetData();

        expect(mediaHandler.frameList.length).toBe(0);
    });
});
```

Listing 5.1: Example atomic test

This example checks if the media handler module correctly clears all unnecessary frames and information tied to a schedule that is no longer being displayed. Running the testing suites generates a results page such as Figure 34.

```
Jasmine 2.0.0

. . . . . . . . . . . . .

0 specs, 0 failures
0 specs, 0 failures
13 specs, 0 failures

  Update list adds a new app.
    App list length should be greater then before the update.
  Update app changes the state of an app.
    Changes the status of an existing app.
    Increments the play counter if status is PLAY.
    Does not increment the play counter is status is not PLAY
  The app list is saved successfully into the local storage.
    Saves the info into the local storage.
  The app list is loaded successfully from the local storage.
    Loads the info from the local storage.
  Create frame returns the id of an existing frame.
    Creates a frame in the html base page.
    Return the frameId of an existing frame.
  Release frame changes the busy state of a frame to not busy.
    Changes the state of the frame model.
  Reset data clears all existing frames and information.
    Empties the framelist.
  It launches a new activity based on a JSON schedule.
    The activity has a scheduling tree.
    The activity has a valid scheduling tree.
    The activity has a populated event table.

Jasmine 2.0.0
```

Figure 34.: Example result page of the unit test

As it can be seen the test results were all positive thus making the player able to move to the next stage of testing: deployment.

5.2.5 *Deployment Tests*

These tests are aimed towards the deployment of the player on several different platforms. This will allow us to identify the deployment procedures and dependencies for each of those platforms.

The platforms being tested are the following:

- Windows
- Linux
- MacOS
- ChromeOS
- Android (raspberryPi, rikomagic, android smartphones)
- iOS (iPad, iPhone)
- Samsung SmartTv

Results

The different builds of the player were deployed on each of their respective platforms according to Table 5 . All of them were successfully deployed and executed.

	Player Builds			
Platforms	Chrome App	Android App	iOS App	Samsung App
Windows	X			
Linux	X			
MacOS	X			
Android		X		
iOS			X	
Rikomagic		X		
Raspberry Pi		X		
Samsung SmartTV				X

Table 5.: Deployment table

5.3 SUMMARY

The first step during the testing process was to identify which tests were needed, and how to implement them, to unsure the player is ready to be deployed on a real digital signage network. After the planning of the testing process it was time to gradually implement the tests, once the player passed the first type of tests it was ready to be tested with the next type of tests and so on. This process started with simple execution and precision tests and ended with deployment tests where the player was deployed on real platforms and its behaviour was documented. Having successfully passed the testing process, the player is now ready to be used on a digital signage network.

6

CONCLUSION

In this final chapter, a conclusion to this investigation will be presented containing a synthesis of the work completed it will also take a look into how the objectives defined previously were, or not accomplished.

Finally, the proposed future work will be presented. That section will identify and contextualize the next logical steps in this project.

6.1 CONCLUSION

The objective of this thesis was to specify and implement a web based digital signage player with multi-platform deployment capabilities based on the specifications of a previous investigation. The initial stages of this project was focused on updating those specifications, identifying new requirements and specifying all the necessary aspects of the player such as an updated architecture, server communication and player to application communication.

These are the specific milestones of the project that were successfully concluded:

- Update to the architecture design and the layered structure of the player, having in mind the necessity of multi-platform support and ease of access to implement other versions of the player;
- Update to the scheduling format, it was redefined to become a self-contained, tree structured format without imposing limits to the creation of schedules;
- Development of capable scheduling and media handler modules capable of interpreting the instructions from the schedule and translate them into visual content.
- Creation of a server – player communication protocol in order to deploy a back-office to the player so it could be fitted with server specific services;
- Specification and implementation of a player – application communication library, it became clear it was necessary to provide the player with a tool to communicate with the applications in order to have a better control of the scheduling process;

- Development of modules to allow deployment on other systems. This allowed the implementation of web based players for platforms such as Android and Samsung SmartTV based on the initial Chrome Packaged App implementation;
- Testing, the player was submitted to a number of tests to assure it was capable of performing under the necessary conditions. These tests encompassed execution tests, precision tests, stress tests, memory profiling tests, unit tests and deployment tests.

Finally, looking at the objectives proposed by this thesis and the concluded milestones presented above, it is apparent that they were successfully accomplished. In the end this project resulted in a well defined specification of the web based player, i's server side services, communication protocols, scheduling format and most importantly a working, tested and valid software.

6.2 FUTURE WORK

Upon reaching the final stages of this work it became clear that even thought its objectives had been met, much more can be made to improve the framework that resulted from the development of the web based player.

- Implement modules to allow deployment on other systems: at this point the player can be deployed on the systems identified on the reference stacks during the specification process. However, considering the modular nature of the framework it is possible, with minimal coding effort, to create other versions of the player for systems similar to the Samsung SmartTV or even other SFFC to move towards a scenario where deploying a digital signage network becomes as easy as downloading an app or connecting an Universal Serial Bus (USB) device to a display.
- Revise the scheduling format into a more dynamic one: for now the scheduling format is a somewhat rigid format, the player receives the instructions and can do very little to interact with it. The next logical step is to move from this paradigm into something where the player can interact more with the schedule, still obeying the rules set by the schedule, but capable of dynamic alterations to the schedule requested by the users or the apps themselves.
- Develop native applications to grant support on more systems: for now there are native applications for Linux and Windows systems with minimal functionalities. It is important that other native applications are developed to provide the players targeting other systems with functionalities specific to those systems.
- Improve the system to player communication in order to augment the native application capabilities: the web based player deployed as a Chrome Packaged App relies on the Chrome native messaging API, while this is reliable for that version of the player the same is not true for players deployed outside of the Chrome Packaged Apps environment. For those cases it

is necessary to investigate and develop a messaging system capable of providing the necessary communication methods between the player and the native application.

Appendices

EXAMPLE SCHEDULE

This example schedule was used throughout the whole project to test and verify the behaviour of the player in several occasions. It contains at least one container of each type.


```
{
    "schedule": {
        "_id": "sche1",
        "version": "1",
        "name": "Default Schedule",
        "LastUpdate": "16-07-2014",
        "playerId": "-",
        "apps": [
            {
                "appId": "app001",
                "src": "http://testapplications.displr.com/feeds/?sources=http://
                    feeds.jn.pt/JN-Destaques&num=10&delay=20&animation=fade&
                    placeId=8",
                "lastUpdate": "15-12-2013"
            },
            {
                "appId": "app002",
                "src": "http://testapplications.displr.com/facebook/index.html?
                    type=photo&fbid=spiritocupcakes&title=SpiritoCupcakes&delay
                    =10&num=10&placeId=8",
                "lastUpdate": "15-12-2013"
            },
            {
                "appId": "app003",
                "src": "http://testapplications.displr.com/twitter/index.html?
                    delay=15&count=12&screenName=cmjornal&method=1&placeId=8",
                "lastUpdate": "15-12-2013"
            },
            {
                "appId": "app004",
                "src": "http://testapplications.displr.com/presences/?placeId=8",
                "lastUpdate": "15-12-2013"
            },
```

```
        {
            "appId": "app005",
            "src": "http://testapplications.displr.com/posters/?placeId=8&fx=
                fade&interval=20&transition=1000",
            "lastUpdate": "15-12-2013"
        },
        {
            "appId": "app006",
            "src": "http://testapplications.displr.com/stream/?placeId=8&
                interval=5000&lang=pt",
            "lastUpdate": "15-12-2013"
        }
    ],
    "content": {
        "rootContainer": "seq",
        "repeatCount": "-1",
        "childElements": [
            {
                "container": "layout",
                "repeatCount": "1",
                "childElements": [
                    {
                        "container": "app001",
                        "duration": "25",
                        "left": "0",
                        "top": "0",
                        "width": "1",
                        "height": "0.5",
                        "minWidth": "0",
                        "minHeight": "0"
                    },
                    {
                        "container": "app002",
                        "duration": "25",
                        "left": "0",
                        "top": "0.5",
                        "width": "0.5",
                        "height": "0.5",
                        "minWidth": "0",
                        "minHeight": "0"
                    },
                    {
                        "container": "seq",
                        "repeatCount": "1",
                        "left": "0.5",
                        "top": "0.5",
                        "width": "0.5",
```

```
                    "height": "0.5",
                    "minWidth": "0",
                    "minHeight": "0",
                    "childElements": [
                        {
                            "container": "app003",
                            "duration": "10"
                        },
                        {
                            "container": "app004",
                            "duration": "15"
                        }
                    ]
                }
            ]
        },
        {
            "container": "layout",
            "repeatCount": "1",
            "childElements": [
                {
                    "container": "app005",
                    "duration": "20",
                    "left": "0",
                    "top": "0",
                    "width": "0.5",
                    "height": "1",
                    "minWidth": "0",
                    "minHeight": "0"
                },
                {
                    "container": "app006",
                    "duration": "20",
                    "left": "0.5",
                    "top": "0",
                    "width": "0.5",
                    "height": "1",
                    "minWidth": "0",
                    "minHeight": "0"
                }
            ]
        },
        {
            "container": "seq",
            "repeatCount": "1",
            "childElements": [
                {
```

```
            "container": "layout",
            "childElements": [
                {
                    "container": "seq",
                    "left": "0",
                    "top": "0",
                    "width": "0.5",
                    "height": "1",
                    "minWidth": "0",
                    "minHeight": "0",
                    "childElements": [
                        {
                            "container": "app004",
                            "duration": "15"
                        },
                        {
                            "container": "app001",
                            "duration": "15"
                        }
                    ]
                },
                {
                    "container": "app002",
                    "duration": "30",
                    "left": "0.5",
                    "top": "0",
                    "width": "0.5",
                    "height": "1",
                    "minWidth": "0",
                    "minHeight": "0"
                }
            ]
        },
        {
            "container": "app003",
            "duration": "20",
            "left": "0",
            "top": "0",
            "width": "1",
            "height": "1",
            "minWidth": "0",
            "minHeight": "0"
        }
    ]
},
{
    "container": "selector",
```

```
                    "left": "0",
                    "top": "0",
                    "width": "1",
                    "height": "1",
                    "minWidth": "0",
                    "minHeight": "0",
                    "selectorType": "weighted",
                    "repeat": "0",
                    "childElements": [
                        {
                            "container": "app001",
                            "weight": "3",
                            "duration": "15"
                        },
                        {
                            "container": "app002",
                            "weight": "1",
                            "duration": "15"
                        },
                        {
                            "container": "app003",
                            "weight": "2",
                            "duration": "15"
                        }
                    ]
                }
            ]
        }
    }
}
```

B

EXAMPLE LOG FILE AND TRACE FILE

This was one of the files generated during the testing phase. The trace file is the file created prior to the schedule's execution and is used to compare with the atual log file to find any mismatch and identify any possible problem with schedule execution.

```
Tick: 0 || Task: PREFETCH || Container: c11 || App: app001
Created frame for container: c11
Tick: 0 || Task: PLAY || Container: c11 || App: app001
Tick: 0 || Task: PREFETCH || Container: c13 || App: app003
Created frame for container: c13
Tick: 1 || Frame for container c13 loaded.
Tick: 2 || Frame for container c11 loaded.
Tick: 15 || Task: PLAY || Container: c13 || App: app003
Tick: 15 || Task: STOP || Container: c11 || App: app001
Tick: 15 || Task: PREFETCH || Container: c11 || App: app001
Container c11 already prepared, event skipped.
Tick: 30 || Task: PLAY || Container: c11 || App: app001
Tick: 30 || Task: STOP || Container: c13 || App: app003
Tick: 30 || Task: PREFETCH || Container: c13 || App: app003
Container c13 already prepared, event skipped.
Tick: 45 || Task: PLAY || Container: c13 || App: app003
Tick: 45 || Task: STOP || Container: c11 || App: app001
Tick: 45 || Task: PREFETCH || Container: c11 || App: app001
Container c11 already prepared, event skipped.
Tick: 60 || Task: PLAY || Container: c11 || App: app001
Tick: 60 || Task: STOP || Container: c13 || App: app003
Tick: 60 || Task: PREFETCH || Container: c12 || App: app002
Created frame for container: c12
Tick: 65 || Frame for container c12 loaded.
Tick: 75 || Task: PLAY || Container: c12 || App: app002
Tick: 75 || Task: STOP || Container: c11 || App: app001
Tick: 75 || Task: PREFETCH || Container: c21 || App: app004
Created frame for container: c21
Tick: 76 || Frame for container c21 loaded.
Tick: 90 || Task: PLAY || Container: c21 || App: app004
Tick: 90 || Task: STOP || Container: c12 || App: app002
Tick: 90 || Task: PREFETCH || Container: c23 || App: app001
```

```
Created frame for container: c23
Tick: 91 || Frame for container c23 loaded.
Tick: 105 || Task: PLAY || Container: c23 || App: app001
Tick: 105 || Task: STOP || Container: c21 || App: app004
Tick: 105 || Task: PREFETCH || Container: c21 || App: app004
Container c21 already prepared, event skipped.
Tick: 120 || Task: PLAY || Container: c21 || App: app004
Tick: 120 || Task: STOP || Container: c23 || App: app001
Tick: 120 || Task: PREFETCH || Container: c23 || App: app001
Container c23 already prepared, event skipped.
Tick: 135 || Task: PLAY || Container: c23 || App: app001
Tick: 135 || Task: STOP || Container: c21 || App: app004
Tick: 135 || Task: PREFETCH || Container: c21 || App: app004
Container c21 already prepared, event skipped.
Tick: 150 || Task: PLAY || Container: c21 || App: app004
Tick: 150 || Task: STOP || Container: c23 || App: app001
Tick: 150 || Task: PREFETCH || Container: c22 || App: app005
Created frame for container: c22
Tick: 150 || Frame for container c22 loaded.
Tick: 165 || Task: PLAY || Container: c22 || App: app005
Tick: 165 || Task: STOP || Container: c21 || App: app004
Tick: 180 || Task: STOP || Container: c22 || App: app005
```

Listing B.1: Log File

```
0 PREFETCH app001
0 PLAY app001
0 PREFETCH app003
15 PLAY app003
15 STOP app001
15 PREFETCH app001
30 PLAY app001
30 STOP app003
30 PREFETCH app003
45 PLAY app003
45 STOP app001
45 PREFETCH app001
60 PLAY app001
60 STOP app003
60 PREFETCH app002
75 PLAY app002
75 STOP app001
75 PREFETCH app004
90 PLAY app004
90 STOP app002
90 PREFETCH app001
105 PLAY app001
```

```
105 STOP app004
105 PREFETCH app004
120 PLAY app004
120 STOP app001
120 PREFETCH app001
135 PLAY app001
135 STOP app004
135 PREFETCH app004
150 PLAY app004
150 STOP app001
150 PREFETCH app005
165 PLAY app005
165 STOP app004
180 STOP app005
```

Listing B.2: Trace File

C

SCREENSHOTS

Here are a two screenshots of the DISPLR player in action. Figure 35 shows the home screen of the player, this screen is presented while the player connects itself to the service.

Figure 35.: Home screen of the player.

On figure 36 it is possible to see the player displaying and executing three applications.

Figure 36.: Apps being executed.

BIBLIOGRAPHY

AOpen. Opensign digital signage, 2013. URL http://www.aopen.com/.

Marco Pereira Carneiro. Um player web para redes de ecrãs públicos. 2013.

WHATWG Community. The iframe element, 2014. URL http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html.

Concerto. Concerto digital signage, 2013. URL http://www.concerto-signage.org/.

Open-Source (MIT) Contributors. Socketio, 2014. URL http://socket.io/.

Kaazing Corporation. Websockets, 2014. URL https://www.websocket.org/.

Tim Down. log4javascript, 2014. URL http://log4javascript.org/.

Google. Chrome storage, 2013. URL https://developer.chrome.com/apps/storage.html.

Google. Chrome packaged app apis, 2014a. URL https://developer.chrome.com/apps/system_memory.

Google. Webview api, 2014b. URL https://developer.chrome.com/apps/tags/webview.html.

JSON Group. Json, 2014. URL http://json.org/.

IAdea. Iadea digital signage, 2014. URL http://www.iadea.com/.

Jasmine. Jasmine, 2014. URL http://jasmine.github.io/2.0/introduction.html.

The jQuery Foundation. jquery, 2014. URL http://jquery.com/.

Thomas Lindén, Tommi Heikkinen, Timo Ojala, Hannu Kukka, and Marko Jurmu. Web-based framework for spatiotemporal screen real estate management of interactive public displays. 2010.

Microsoft. Windows presentation foundation, 2014a. URL http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx.

Microsoft. Windows performance monitor, 2014b. URL http://technet.microsoft.com/en-us/library/cc749249.aspx.

Novisign. Novisign digital signage, 2014. URL http://www.novisign.com/android/android-based-digital-signage/.

Max Ogden. Building webview applications, 2012. URL http://maxogden.com/building-webview-applications.html.

OpenSplash. Opensplash digital signage, 2014. URL http://www.opensplash.net/.

P2P-SIP. Restful communication over websocket, 2014. URL http://p2p-sip.blogspot.pt/2011/06/restful-communication-over-websocket.html.

Mark Pilgrim. The past, present and future of local storage for web applications, 2011. URL http://diveintohtml5.info/storage.html.

et al. RFC 2616 Fielding. Http requests, 2004. URL http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html.

Samsung. Samsung developers, 2014a. URL developer.samsung.com/.

Samsung. Samsung smarttv, 2014b. URL http://www.samsung.com/pt/smart-tv-o-futuro-agora/.

Sapo. Sapo digital signage, 2013. URL https://github.com/sapo/digital-signage-client.

Signagelive. Signagelive digital signage, 2014. URL http://signagelive.com/.

Constantin Taivan, Rui José, and Bruno Silva. Understanting the use of web technologies for applications in open display networks.

TargetR. Targetr digital signage, 2014. URL http://www.targetr.net/technology.

Ubisign. Ubisign digital signage, 2014. URL http://ubisign.com.

Rise Vision. Rise vision digital signage, 2014. URL http://www.risevision.com/.

W3C. Xsl transformations (xslt), 1999. URL http://www.w3.org/TR/xslt.

W3C. Web services architecture, 2004. URL http://www.w3.org/TR/ws-arch/.

W3C. Html5 web messaging, 2012a. URL http://www.w3.org/TR/webmessaging/.

W3C. Smil, 2012b. URL http://www.w3.org/AudioVideo/.

W3C. Web workers, 2014. URL http://www.w3.org/TR/workers/.

Wikipedia. Mime, 2014. URL http://en.wikipedia.org/wiki/MIME.

Xibo. Xibo digital signage, 2014. URL http://xibo.org.uk/.