Integrating the Google Cast Technology in a Second-screen Solution

by

Niklas Lavrell

LIU-IDA/LITH-EX-A–14/031–SE

2014-06-23
Final thesis

Integrating the Google Cast Technology in a Second-screen Solution

by

Niklas Lavrell

LIU-IDA/LITH-EX-A–14/031–SE

2014-06-23

Supervisor: Anders Fröberg (Linköpings University), Deng Shengchun (Harbin Institute of Technology), Peter Steiner (Accedo Broadband AB)

Examiner: Erik Berglund (Linköpings University)
Abstract

The newly released Google Chromecast has generated an increasing amount of interest for so called second-screen experiences in the market. Although the technology behind such experiences has existed for a couple of years, end users are now truly starting to grasp the concept and benefits of multi-screen. The company, at which the thesis was performed at, provides a cloud-based messaging solution for Internet connected devices, which enables multi-screen use cases. To increase the amount of supported platforms in the solution, new technologies frequently needs to be integrated. In this thesis I have performed an exploratory research & development project with the aim to integrate the Google Cast technology in this multi-screen solution. The fundamental difference in how the two ecosystems were designed resulted in a companion device framework that acted as a wrapper over the technologies. The framework was implemented on the Android platform together with a set of demo applications. The proposed solution should be seen as a starting point for integrating different multi-screen technologies within a single companion device framework. While combining these technologies, a fundamental difference in the user experience between them became apparent. The Google Cast ecosystem relies on the companion device as the interaction point for the end user, whereas television (TV) applications usually have the main interaction point on the actual TV itself via a dedicated remote control. Having this kind of inconsistency within the same set of applications increases the risk of confusion among end users. Therefore I suggest that development of such multi-screen experiences, that combines these technologies, should strive for a high consistency throughout the whole user experience, independent of platforms and technology.
Addendum

This master thesis has been performed in parallel with a thesis performed by Emil Bergwik. The two theses have concerned the same product, wherefore they have had an impact on each other. Consequently the two thesis reports might share some conceptual and artifactual similarities.
I would like to begin with a big thanks to Accedo Broadband, for the opportunity to perform this master thesis in this truly exciting and fast growing market, especially thanks to my supervisor Peter Steiner and Fredrik Sandberg for feedback and support during the thesis. Also, thanks to my supervisors and examiners at Linköping University and Harbin Institute of Technology, Anders Fröberg, Erik Berglund and Deng Shengchun. Also, for feedback and opposition I thank my classmate David Buö.

Further, I would like to extend my gratitude to my good friend and colleague Emil Bergwik for all support and the exciting time spent together. Finally, I would like to thank my family for all support during my studies and especially my beloved Jonna.

Niklas Lavrell
Stockholm, May 2014
## Contents

1 Introduction ............................................. 2
   1.1 Background of Study ............................... 3
   1.2 Purpose & Aim .................................... 4
   1.3 Problem Definition ................................. 4
   1.4 Scope & Limitations ............................... 4
   1.5 Approach ......................................... 5
   1.6 Terminology ...................................... 6
   1.7 Structure ......................................... 6

2 Theoretical Background .................................. 8
   2.1 TV Interactivity .................................. 8
       2.1.1 Companion Devices .......................... 9
   2.2 Media Consumption Trends ......................... 11
   2.3 Current Challenges ............................... 11

3 Google Chromecast ..................................... 13
   3.1 Use Cases ........................................ 13
       3.1.1 Remote Playback ............................ 14
       3.1.2 Secondary Output ............................ 14
   3.2 Technology ....................................... 15
       3.2.1 Discovery of Devices ....................... 15
       3.2.2 Communication with Devices ............... 15
   3.3 Implementation .................................. 16
       3.3.1 Receiver .................................... 16
       3.3.2 Sender ...................................... 18

4 Accedo Connect ......................................... 22
   4.1 Use Cases ........................................ 22
       4.1.1 Pairing Process ............................. 23
   4.2 Technology ....................................... 24
       4.2.1 Connection States ........................... 24
       4.2.2 Messaging .................................. 25
       4.2.3 Host & Clients .............................. 25
       4.2.4 Library Injections ......................... 26
   4.3 Implementation .................................. 26
## CONTENTS

4.3.1 Connecting to the Accedo Cloud ............... 27
4.3.2 Connecting to a Channel .................... 27

5 Development ........................................ 30
5.1 Process ........................................... 30
5.2 Pre-study Results ................................. 31
5.2.1 Conceptual Solution Idea .................... 31
5.3 Use Cases & Requirements ....................... 33
5.4 Architecture ....................................... 33
5.4.1 Refining the Architecture .................... 34
5.5 Implementation .................................... 38
5.5.1 Framework Classes ............................ 38
5.5.2 Google Cast .................................... 42
5.5.3 Accedo Connect ................................. 44
5.5.4 The DIAL Protocol ............................. 45
5.6 Demo Applications ................................. 46
5.6.1 Android Application ........................... 46

6 Result & Analysis .................................... 49
6.1 Framework Development ............................ 49
6.2 Application Development .......................... 50
6.3 End User Perspective ............................... 51

7 Discussion .......................................... 52
7.1 End User Interaction Point ....................... 52
7.2 Communication Technique ....................... 53
7.3 Alternative Solutions .............................. 54
7.4 Future Work ....................................... 55

8 Conclusion .......................................... 56

Bibliography ........................................... 56

Appendix ............................................... 61

A The Google Chromecast ............................. 62

B RemoteMediaPlayer Initialization .................. 63

C Solution Idea ....................................... 64

D Discovery and Device Abstraction .................. 65

E The MediaState Object .............................. 66

F The CastManager Class ............................. 68
G  A BroadcastReceiver Implementation  77
Acronyms

API  application programming interface
CP   content provider
CSS  Cascading Style Sheets
DIAL Discovery and Launch
DLNA Digital Living Network Alliance
DRM  digital right management
EPG  Electronic Programming Guide
GUI  graphical user interface
HLS  HTTP Live Streaming
HTML HyperText Markup Language
HTTP Hypertext Transfer Protocol
IP   Internet Protocol
mDNS multicast Domain Name System
NUI  natural user interface
OS   operating system
OTT  Over the Top Technology
QR   Quick Response
RAMP Remote Application Media Protocol
SDK  software development kit
SSDP Simple Service Discovery Protocol
STB  set-top box
TV   television
UC   Universal Control
UDLR up-down-left-right
UI   user interface
UPnP Universal Plug and Play
URL uniform resource locator
VOD video on demand
Chapter 1

Introduction

The number of devices connected to the Internet has increased exponentially in the latest years. Nowadays it is not only desktop computers, laptops, and smartphones; it is also cars, locks, thermostats, and TVs (Cramer, 2014). The era of the so called Internet of Things is in its rising years and a quite apparent change for the end users, during the recent years, has been the growth of the smart TV market. The smart TVs are still quite young though, wherefore great potential exists for device manufacturers and content providers (CPs) to win the battle of consumers. A lot of players in the market of the living room results in a very fragmented one, both in a hardware and software perspective, where everyone is trying to become the mainstream choice. As for now, the majority of TV manufacturers have their own operating system (OS) powering their devices. This is not an unusual behavior in such new markets, as seen in the phone market before the rise of iOS and Android, the question is though: how long time will it take before some players eliminates the competition?

Even though arising markets often achieve great technology advancements, the market fragmentation in the living room could be a real hassle for application developers and in this case also the media companies and CPs. If a CP can not provide their content to the end users, they can not really do any business. A key for the CPs is therefore to be able to deliver their content to as many end users as possible, in other words having their content available on as many platforms as possible. Given this technology fragmentation, developing applications becomes a very time consuming task and only the companies with great resources can reach out to the majority of end users.

In the meantime as TV manufacturers are making moves, other big players such as Apple and Google are also interested in the potential of the living room context. Apple entered the market in 2007 with the first generation of their Apple TV (Wikipedia, 2014a), whilst Google entered in 2010 with their counterpart Google TV (Wikipedia, 2014c). Google TV was not the
success that Google had hoped for though (GigaOM, 2014), so after rework-
ing the concept they released the Chromecast in 2013 (Wikipedia, 2014b).
The Chromecast has its main interaction point on the companion device, enabling intuitive second-screen use cases. The Chromecast can become a game changer in the market, with a competitive price compared to the smart TVs, but as for now it is just another device which contributes to even more technology fragmentation for end users as well as the CPs.

Also, the first screens (TVs) are getting smarter in the aspect of companion device interactions. Big players are releasing solutions, such as the Apple AirPlay and the Samsung MultiScreen, however they usually are bound to their own platforms, which limits the connectivity with other platforms and can only be used by those with a full setup of devices from that specific company. If the market fragmentation is an issue for regular application development, presumably it is an even bigger problem for development of multi-screen experiences which ends up in the middle of all the available platforms.

### 1.1 Background of Study

This thesis is performed as a final thesis on a master’s double degree pro-
grame in international software engineering. The programme is a coop-
eration between Linköping University in Sweden and Harbin Institute of Technology in China. The thesis will be supervised, evaluated and exam-
ined by both universities; therefore requirements from both universities must be fulfilled within the thesis.

Accedo Broadband (henceforth referred to as Accedo) is the market lead-
ing enabler of TV application solutions. Accedo’s extensive experience and product portfolio reach out to a variety of customers, including media com-
panies, operators, and device manufacturers. Accedo employs around 200 people around the world, with offices in Europe, North and South America, Asia, and Australia. The thesis is performed at Accedo’s product depart-
ment at the headquartered office in Stockholm, Sweden, with a collaboration with the product development at the Honk Kong office.

Given the market fragmentation, Accedo’s experience of developing plat-
form independent solutions is a main reason for their success in the business, since it allows faster deployment of applications to more platforms. This is nowadays considered a core competence. Accedo has great experience of application development for smart TV, set-top box (STB), game consoles (like Xbox and PlayStation) etc., with more than 1000 applications deployed on more than 40 different platforms (Accedo Broadband, 2013). To cope with the fast delivery of application implementations, Accedo provides var-
ious products to ease the development of applications. A common theme in their product portfolio is to act as a abstraction layer above the frag-
mentation in the market. The interesting product for this thesis is Accedo Connect (referred to as Connect) which is a white label cloud-based mes-
saging solution for connected devices. Connect offers the possibility to build multi-screen experiences by connecting applications on different platforms with each other. Supported platforms are a variety of smart TVs, STBs, smartphones, tablets; in other words the majority of Internet connected consumer electronic devices. The primary use case to achieve with the Connect solution is the swipe to play feature, which includes connecting a companion device to the TV, browse a media library on the companion device, and play the media on the TV. Connect will be described more thorough in chapter 4.

The newly released Chromecast from Google is a streaming device that connects to the TV and enables multi-screen experiences by communicating with companion devices within the same Wi-Fi network. Compatible companion device platforms include Android, iOS, and the Chrome web browser, which together reaches the majority of end users. The primary use case, here as well, is to easily send content to the big screen without the need for other remote controls. A more detailed description of the Chromecast can be found in chapter 3.

1.2 Purpose & Aim

The fact that Connect offers a comprehensive platform support is one of the unique selling points for the product. The application developer could implement Connect is his set of applications and reach a scenario where the same multi-screen experience is achieved whether the end user has a Samsung smart TV and a iOS device; or a LG smart TV and an Android device. After the release of the Chromecast, presumably some end users (of this application powered by Connect) would like to be able to use their Chromecast as the first screen platform. Consequently, the purpose of this thesis is to evaluate the possibilities of adding support for Chromecast devices in the Connect ecosystem, and also to provide recommendations on how to achieve this. The aim is to develop a solution that does not limit any functionality of neither Connect nor Cast, yet keep a simple interface towards implementers of the solution.

1.3 Problem Definition

Given the purpose of this thesis, the problem definition that the thesis should answer is as follows: How can the Google Cast technology be integrated with a cloud-based messaging solution?

1.4 Scope & Limitations

Due to the time limitation and to be able to achieve a sufficient technological depth in the thesis, the scope must be narrow enough and limitations must be clarified. The following is set up for my thesis:
1.5. APPROACH

As stated before, the thesis is performed at Accedo and the cloud-based messaging solution is the previously described Accedo Connect. To be able to propose a solution on how to integrate Chromecast support into Connect, thorough studies of the Cast and Connect technologies must be performed. These studies might be performed in parallel or in a serial manner, depending on the circumstances such as resources at Accedo, events regarding the technologies, new releases of the technologies, and so forth. After the technologies are examined, the process of developing a solution that integrates them is started. The development and the actual implementation of the solution is performed with an agile development methodology since it is used within Accedo, which means that the solution will be improved and refined as the thesis proceeds. When the time limitation of the implementation phase is due, an evaluation of the solution is to be performed. The conclusions of the studies and the result of the implementation will be the foundation of the recommendations that will be given to Accedo at the end of the thesis. To concretize the process, the following list summarizes the thesis process:

- Analyzing the Google Cast technology.
1.6 Terminology

This section gives explanations of some frequently used terms in the report, whose meaning might not be obvious for the reader.

- First screen: This term refers to a bigger screen in some context, often the TV in the living room, but can also be for example the projector in a meeting room.
- Second screen: The smaller screen in some context, often a smartphone or a tablet. Can be seen as a personal screen. In this thesis, seconds screen and the term companion device is treated synonymously.

1.7 Structure

The master thesis report is divided into eight main chapters:

Chapter 1: Introduction This first chapter gave the reader an introduction to the thesis and why it is performed.

Chapter 2: Theoretical Background This chapter will introduce the theoretical background of the context.

Chapter 3: Google Chromecast This chapter will describe the Chromecast and Google Cast technology in detail.

Chapter 4: Accedo Connect This chapter will describe the Accedo Connect solution and its technology.

Chapter 5: Development This chapter will describe the performed development work; concept, design and implementation.

Chapter 6: Result & Analysis This chapter will handle the results from the performed project and analyze them.

Chapter 7: Discussion This chapter will contain a discussion regarding the thesis, its findings and suggestions on future work.
Chapter 8: Conclusion  This last chapter will highlight the main conclusions from the performed thesis.
Chapter 2

Theoretical Background

This chapter will introduce the reader to the relevant theoretical background from the research in the field.

2.1 TV Interactivity

During the latest century, the complexity of the TV and devices connected to the TV has increased rapidly. But even though the features and options to consume media on the first screen (TV) have increased, the classical TV remote control has not seen same evolution. When linear TV was the standard way to consume media in the living room, the up-down-left-right (UDLR) remote control was adequate to the available navigation options (mainly change channel and volume etc.). To reach the desired media, the user simply had to: turn on the TV; change to the desired channel. But as the TV systems got more and more complex, the limitations of the UDLR remote control emerged. Gritton (2013) describes the term ”hundred-button remote” which explains the phenomenon seen in classical remote controls when new functionality was added to the system; new buttons was simply added to the remote, ending up in an overload of buttons. The new functionalities were not only feature rich and complex user interfaces (UIs) on the TV itself, but also on devices connected to the TV, such as a STB or a game console. Reaching ones desired media nowadays usually requires the user to turn on the TV, as well as the STB, switch to the correct input source on the TV, then navigate to the correct channel/program on the STB. A consequence of this was that the end user often found himself with an overload of remote controls in the living room, to be able to control all devices. To summarize, Bernhaupt and Pirker (2013) identified three areas where the classical UDLR remote was insufficient:

- **Usability**: The need for multiple remote controls make the user experience inadequate
• **Adaptability**: The classical remote can not adapt itself to the context; it can not be changed

• **Scalability**: The ”hundred-button-remote” becomes an unsustainable solution

Later, the smart TVs were introduced in the market, enabling a multitude of new features and applications to be experienced on the big screen. However, studies have shown that most users with a smart TV in their living room, still mostly consume linear TV (Gritton, 2013). The reason for this phenomenon seems to be the complexity of the UI in combination with the now inadequate UDLR remote control.

Many researchers have proposed solutions that will enhance the interactions with the TV in new innovative ways. Gritton (2013) proposes a solution based on a motion sensitive remote control to achieve a so called natural user interface (NUI) design pattern. Lee et al. (2012) proposed a long-range touch interface for controlling the TV (an imaginable screen in front on the end user, a camera on the TV registers the hand movements). This solution has also been proposed by Fan et al. (2013). Lee et al. (2013) later improved his proposal, including face recognition for viewer authentication. These relatively new ideas are starting to be introduced in the market as this thesis is written. The idea of using a secondary screen for controlling the TV has not seen the same growth even though the idea is relatively old.

### 2.1.1 Companion Devices

Already in the 1990s researchers suggested that more research should focus on the interaction between the TV and other devices (Coffey and Stipp, 1997). Using a second screen (companion device) was first suggested by Robertson et al. (1996), but has become more popular in recent years. Probably because the fast growth of handheld devices such as smartphones and tablets, has made the multi-screen use cases more apparent.

Even though many viewers use a smartphone in parallel with their TV consumption, they often use it for things not related to the actual TV content (such as browsing the web or checking e-mails) (Evelien and Paulussen, 2012; Google Mobile Ads, 2012). Tsekleves et al. (2009) identifies two main categories for how a secondary screen can be used as companion to the TV. The first category mainly covered the controlling of the first screen with the companion device, whilst the second category was to enrich the viewer’s experience in front on the TV. Cesar et al. (2008) more thoroughly identified four main usage areas for companion devices to the TV: control, enrich, share, and transfer.
Control

The controlling usage area for a companion device is simply to control the UI on the first screen. Experiments have been made where end-users have shown a strong preference for interacting with the TV via a second screen instead of a classical TV remote (Tsekleves et al., 2007), and the majority of people already know how to interact with a touch-screen (Bernhaupt and Pirker, 2013). This feature can also be extended to achieve a shared and personal UI in a multi-user context. Cesar et al. (2008) describes that this could be realized by using a shared view on the first screen and a personalized view on the second screen. Kim et al. (2013a,b) proposes a solution for controlling the first screen via a screen mirroring technique. The user navigates in the UI on the companion device which he probably would be familiar with since controlling via a touch interface, like a companion device, has become an acceptable way of interacting with a system (Bernhaupt et al., 2012).

Enrich

Content enrichment can be achieved by using a secondary screen that shows additional information in conjunction with the first screen (Basapur et al., 2011; Hess et al., 2011). Example use cases can be: using the second screen to preview other channels (Bernhaupt et al., 2012); showing additional information in parallel with a TV show (Basapur et al., 2011), such as a story-map (Murray et al., 2012); enable the viewers to play the quiz show against the studio participants (Jolly and Evans, 2013). Commercial attempts to enrich content via companion devices has been performed, but little is known in the literature about the resulting experience of these applications (Basapur et al., 2011).

Share

TV viewing can be a very social experience and new technologies further increase the possibilities. Social TV is the concept of bringing TV experiences and social interactions via the computer closer together (Ducheneaut et al., 2008). A pretty straightforward use case that implements this category is sharing one’s thoughts about a TV show via social media (Doughty and Lawson, 2012). For instance, Vinayagamoorthy and Kramskoy (2013) present a prototype that helps viewers refer to the media that they are consuming. Another idea for a use case is a social TV Electronic Programming Guide (EPG), where content and recommendations can be based on the users social relationships (You et al., 2013).

Transfer

More and more Internet connected devices also give rise to cross-device media consumption, which enables end users to move the content from one
2.2 Media Consumption Trends

When the TV first was introduced in the market, the available content to consume was fairly limited. But ever since then, the amount of content has increased massively. Narasimhan (2011) describes another trend of the X-shifting category, namely the time-shifting trend. This trend is driven at first by personal video recorders, letting users record their favorite TV shows and watch them at a later time. Nowadays it is also driven by the available video on demand (VOD) services, letting users watch content whenever they like (Hess et al., 2011). This results in more individualized consumption trends that are tailored to users interests and schedule. Since using a smartphone is a personal experience, combining these new media consumption trends with a companion device experience, could create great synergy effects.

2.3 Current Challenges

The big challenge for developers of companion device applications is to get a wide user acceptance. Wherefore the companion device to be accepted as a TV interaction device, needs to fulfill some end user requirements. Bernhaupt and Pirker (2013) provide a set of recommendations to application developers that wish to successfully implement a companion device experience:

- The companion device should be able to control all devices in the living room.
- There should exist support usage-oriented scenarios.
- The application should enable personalization and personal usage.
- It should provide meaningful functionalities to enhance the overall experience.
- There should be support for touch and speech interactions.
- The application should reach all user groups.
Studies shows that the today’s smartphone applications fails at replacing the classical TV remote control since they fail at achieving the first and foremost requirement, that is controlling all devices in the living room (Bernhaupt et al., 2012). Those applications simply become yet another remote for the TV experience and wherefore do the opposite to solving the initial issue.

In the market, there exist solutions for coping with these kinds of interoperability problems, for instance Universal Plug and Play (UPnP) and Digital Living Network Alliance (DLNA). However, Jolly and Evans (2013) claims none of them meet the required needs without making any extensions or modifications to the technologies. Instead, they propose an application programming interface (API) called Universal Control (UC), with the aim to solve this issue. UC is a RESTful web-based API that lets connected devices be controlled remotely from other applications (Barrett et al., 2011). Another solution containing a whole infrastructure including back- and front-end architecture have also been proposed (Jin et al., 2013). Cortez et al. (2012) proposes another protocol called Device Communication for inter-device communication on the local network. It provides a full-duplex communication (two-way communication) between devices which is favorable over other solutions. No solutions have yet become the choice of the market, but there exists a obvious interest and desire to reach more interoperable inter-device communication standards.
Chapter 3

Google Chromecast

The Chromecast is a media streaming dongle from Google (seen in Appendix A), which connects to the TV via the HDMI port, and connects to the Internet via a Wi-Fi connection. The Chromecast is controlled remotely from applications on companion devices within the local network, such as a smartphone or a laptop, wherefore it enables multi-screen experiences between them. Supported companion platforms include Android, iOS and the Chrome web browser which is available on Windows, Mac OS, and Chrome OS. Compared to other solutions in the market, the Chromecast is considered a cheap and easy way to consume online videos and music on the TV. The Chromecast implements the Google Cast technology and is therefore often referred to as a Cast device. By the time of writing this thesis report, the Chromecast is the only Cast device available in the market.

3.1 Use Cases

The primary use case for the Chromecast is to send content from a second screen to a big screen. The Chromecast streams media, over the Internet Protocol (IP), and the playback can be controlled from the companion application, as shown in Figure 3.1. Since the Chromecast is controlled only by companion applications, no other remote control is needed. The general flow of sending content from a sender application to a Chromecast is as follows:

1. Launch the sender application, for example an Android application.
2. Press the Cast button and select the device which you like to connect to.
3. Choose some content on the sender application that will start playing on the Chromecast.
There are two fundamental ways of sending content to a Chromecast device: as remote playback, and as secondary output.

### 3.1.1 Remote Playback

Examining Figure 3.1, one of the Chromecast’s biggest strength over other streaming solutions can be seen. Streaming techniques that mirrors the screen of the companion device rely on the processing power of the companion device itself since it is the source of the stream. The Chromecast removes this dependency, since it can retrieve content directly from its source on the web, decode and play it by itself. This solution results in a lot of saved processing power on the companion device compared to a mirroring approach. Control of the playback is performed via messages that are sent to the Chromecast via a proprietary protocol over the local Wi-Fi network. So usually, the companion device browse a media library on the web and then sends a ”play this specific media” message to the Chromecast, which in turn streams the media it by itself.

### 3.1.2 Secondary Output

The second alternative to send content to the Chromecast is actually to mirror parts of the companion device screen. The Google Cast extension in the Google Chrome web browser lets users mirror a specific tab in the browser. There are also experimental functions that give the opportunity to
show the entire computer screen on the Chromecast. This type of streaming requires a lot of processing power in the sender device, which can contribute to a bad user experience on low-end devices. Some settings can be adjusted to achieve a better performance due to lower quality but as for now, this functions is stated as beta and only available for the Google Chrome web browser. This option is not seen as the primary use case for the Chromecast.

These use cases is possible due to the Cast ecosystem, which contains receiver applications, running on a Cast device, and sender applications, running on Android, iOS, or in a Chrome web browser. The sender applications can discover Cast devices on the local Wi-Fi network, launch the receiver application on it, establish a channel between them, and communicate over that channel.

3.2 Technology

No official documentation describes the internal techniques of the Cast software development kit (SDK). However, some reverse engineering and research on the web gave some interesting clues. The technologies being used within the Cast SDK can be divided in two categories: discovery of devices and communication with a device.

3.2.1 Discovery of Devices

The initial versions of the Cast SDK were using the DIAL protocol for discovering Cast devices on the local Wi-Fi network and also launching application on the device (Nicholls, 2013a). DIAL is a co-developed protocol by YouTube and Netflix that simply aims to discover devices and launch applications on them. It is built on the UPnP and Simple Service Discovery Protocol (SSDP) techniques over a Hypertext Transfer Protocol (HTTP) connection (Netflix, Inc., 2012). Application developers can in other words use the Cast SDK or use custom SSDP requests (in line with the DIAL specification) to launch applications on the Chromecast. In newer versions of the SDK, the discovery of devices is done via multicast Domain Name System (mDNS) (Dutta and Nicholls, 2014). However, Google is recommending application developers to only use the provided APIs for communicating with Cast devices.

3.2.2 Communication with Devices

For communication with a device, the Cast SDK uses a proprietary protocol on the local network called Remote Application Media Protocol (RAMP) (Nicholls, 2013b). The RAMP messages are sent over a WebSocket connection between the devices. A WebSocket connection provides a so called full-duplex communication between a web browser and a web server (two-way communication), unlike HTTP, which often is seen as preferable when
3.3 Implementation

To implement a Cast enabled application one need to consider both the sender and the receiver side of the Cast environment. This section will give an introduction to the ecosystem from an application developer viewpoint.

3.3.1 Receiver

The receiver application running on a Cast device is a web application that is developed using regular HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript. The Cast receiver API is a JavaScript library which needs to be included in the application. As seen in Figure 3.2, the Chromecast runs a stripped-down version of Google’s Chrome web browser. But even though it runs web applications, the hardware of the Chromecast is highly optimized for media playback, so implementing an application with a lot of heavy animations will not result in a good user experience even though it is totally doable from an implementation point of view. Clearly, the purpose for the Chromecast is not to provide full-fledged applications with heavy UIs (like the ones running on a smart TV), but to provide a simple way to consume content on the big screen.

The receiver application, which is supposed to be hosted somewhere on the web, must be declared within Google’s Cast SDK Developer Console. The developer console is a self-administration tool for application developers where they can declare receiver applications and development devices.
When declaring a web application as a receiver application, one receives an application ID that is needed in the companion application when establishing a connection to a Cast device. When the Cast device receives a request with a specified ID, it looks up the application ID on Google’s servers, and browse the corresponding website. Consequently, there are no applications that need to be downloaded to the Chromecast in beforehand. There are three types of receiver applications that can be implemented by the developer: the default media receiver, the styled media receiver, and the custom receiver.

**Default media receiver**

The default media receiver is a pre-built receiver application provided by Google. The application is a non-branded media player that supports the basic media playback commands such as play, pause, seek, volume changes etc. Any developer can use the default media receiver without any restrictions.

**Styled media receiver**

The second kind of receiver is the styled media receiver. This is basically the same application as the default media receiver but it allows the developer to override some CSS to brand the application with his own logos, images, colors, and so forth. To use the styled media receiver, the developer needs to register his application in the Cast Developer Console and provide a uniform resource locator (URL) to a CSS file. Listing 3.1 shows some sample CSS code for the styled media receiver.

```css
.background {
  background-image: url(background.png);
}

.logo {
  background-image: url(logo.png);
}

.progressBar {
  background-color: rgb(0, 100, 200);
}

.splash {
  background-image: url(splash.png);
}

.watermark {
  background-image: url(logo.png);
  background-size: 20%;
}
```

Listing 3.1: Sample CSS for a styled media receiver
3.3. IMPLEMENTATION

3.3.1 Custom receiver

The third and last receiver is a custom receiver, which is a regular web application that the developer can develop using HTML, CSS and JavaScript. The developer must implement the Cast JavaScript API and must handle all messages that are received by the application and develop corresponding functionality. This option is required if the developer wants to add more functionality than the default or styled media receiver can provide, for example support for HTTP Live Streaming (HLS) or level 1 digital right management (DRM) support such as Widevine or PlayReady. The receiver application can manage multiple connections to different senders simultaneously, enabling great multi-user experiences such as games or queuing of media. But as stated before, the hardware is limited and such implementations should be developed with a rigorous quality assurance process.

Debugging of the receiver applications running on a Cast device can be performed with the Chrome Developer Tools. This is done by browsing the IP address of the Cast device on port number 9022, in a Chrome web browser.

3.3.2 Sender

As stated before, the sender application is an application that runs on Android, iOS, or in a Chrome browser. In every case, the developer must implement a specific API for the platform. As stated earlier, this thesis will focus on the Android platform, but the other platforms behave in a similar way. The following text will introduce the Android Cast API for the reader, and does not have the purpose of giving an extensive walkthrough on how to develop a production ready application with Cast support. The general flow to establish a connection with a receiver application from an Android application is as follows:

- Initiate a search for devices on the network
- The application receives a Cast device chosen by the end user
- Establishing a connection to the Google Play services
- Launch an application on the receiver
- Establish a communication channel to the receiver application

To initiate a search for devices, the developer can use the Android Media Router framework which is a support library that enables communication with external playback devices, such as screens or speakers. This can be seen in Listing 3.2. To start, we initialize the MediaRouter itself, the MediaRouteSelector which acts as a filter for devices, and the MediaRouterCallback object in which we receive the chosen device. As seen, the application ID
3.3. IMPLEMENTATION

CHAPTER 3. GOOGLE CHROMECAST

(APP_ID) received in the Cast Developer Console is entered in the initialization. The actual scan is then started when the MediaRouter.addCallback method is called, which is recommended to do in the onResume method of the Android Activity class.

```
...  
private MediaRouter mMediaRouter;
private MediaRouteSelector mMediaRouteSelector;
private MediaRouter.Callback mMediaRouterCallback;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...
    mMediaRouter = MediaRouter.getInstance(appContext);
    mMediaRouteSelector = new MediaRouteSelector.Builder()
        .addControlCategory(
            CastMediaControlIntent.categoryForCast(APP_ID).build());
    mMediaRouterCallback = new MyMediaRouterCallback();
}

@Override
protected void onResume() {
    super.onResume();
    mMediaRouter.addCallback(mMediaRouteSelector,
        mMediaRouterCallback, MediaRouter.CALLBACK_FLAG_PERFORM_ACTIVE_SCAN);
}
```

Listing 3.2: The initialization of the MediaRouter API and search for devices

The Media Router framework includes graphical components that let the user select a device. The selected device is then received in the onRouteSelected method of the MediaRouterCallback class as shown in Listing 3.3.

```
private CastDevice mSelectedDevice;

private class MyMediaRouterCallback extends MediaRouter.Callback {
    @Override
    public void onRouteSelected(MediaRouter router, RouteInfo info) {
        mSelectedDevice = CastDevice.getFromBundle(info.getExtras());
        launchReceiver();
    }
    ...
}
```

Listing 3.3: Receiving of the selected Cast device

Since the Google Cast SDK is a part of the Google Play services, the next step is to connect to the Google API Client. The GoogleApiClient is the main interaction point for an application that wants to use functionality
from the Google Play services. How to establish this connection is shown in Listing 3.4.

```java
mConnectionCallbacks = new ConnectionCallbacks();
mConnectionFailedListener = new ConnectionFailedListener();
Cast.CastOptions.Builder apiOptionsBuilder = Cast.CastOptions.builder(mSelectedDevice, mCastListener);
mApiClient = new GoogleApiClient.Builder(this)
    .addApi(Cast.API, apiOptionsBuilder.build())
    .addOnConnectionFailedListener(mConnectionFailedListener)
    .build();
mApiClient.connect();
```

Listing 3.4: Initialization and connection to the Google API Client

As seen, there are a lot of callback objects for handling different types of callbacks from the Google Play services as well as the actual Cast device. We assume that the connection was successfully established, so the next step is to launch the actual application on the receiver. This is done via the `Cast.CastApi.launchApplication` method as shown in Listing 3.5. Attached to the launch method call is a callback class `ResultCallback` which asynchronously receives an answer from the API when the action is performed. The `Cast.CastApi.launchApplication` will automatically try to join a session with the receiver application if it is running at the time. So for instance, if a user is casting a video via the YouTube application and a second user connects to the receiver, also via the YouTube application, the second user will join the session of the first user. If the second user launches a new application, the running application will be stopped and the new application will be launched. When developing certain multi-user experiences, the `Cast.CastApi.joinSession` method can be usable since it only successes if trying to join another session.

```java
Cast.CastApi.launchApplication(mApiClient, APP_ID, false)
    .setResultCallback(
        new ResultCallback<Cast.ApplicationConnectionResult>() {
            @Override
            public void onResult(Cast.ApplicationConnectionResult result) {
                if (result.getStatus().isSuccess()) {
                    /* Successful launch */
                } else {
                    /* Error */
                }
            }
        });
```

Listing 3.5: Launching the receiver application on the Chromecast

The next step is to establish a channel on which the application can send messages to each other. The Cast API allows developers to create own
channels to structure messages by themselves. If developing an application for the media playback use case, it is favorable to use the RemoteMediaPlayer which is a pre-built channel that handles media playback messages. The RemoteMediaPlayer object let developers call methods such as play, pause, and requestStatus which can be seen in Listing 3.6. The Remote Media Player is compatible with the default media receiver, the styled media receiver, as well as the custom receiver if the developer handles the messages correctly. Initializing the channel and setting up the callback methods can be seen in Appendix B.

```java
mRemoteMediaPlayer.requestStatus(mApiClient)
    .setResultCallback(
        new ResultCallback<RemoteMediaPlayer.MediaChannelResult>() {
            @Override
            public void onResult(MediaChannelResult result) {
                if (!result.getStatus().isSuccess()) {
                    // Failed to request status
                }
            }
        });
```

Listing 3.6: Establishing a channel with the RemoteMediaPlayer
Chapter 4

Accedo Connect

Accedo Connect (referred to as Connect) is a cloud-based messaging solution for devices connected to the Internet, such as smartphones, tablets, smart TVs, STBs, game consoles etc. Devices can be paired with each other to establish a communication channel in the cloud. In this channel can text messages be sent and received by every device. The pairing process uses codes for authentication of devices, that is: the first screen shows a four digit code that the end user inputs on the companion device and the devices gets paired in the Accedo cloud. This process can also be simplified by using Quick Response (QR) codes (that simply represents the pairing code) that is shown on the first screen and which the user scans with the camera of the companion device. An overview of the Connect solution is seen in Figure 4.1.

4.1 Use Cases

One of the most desired use cases that can be achieved using Connect is the *swipe to play* functionality, which means that the user can browse a media
library on the companion device and then perform a swipe motion against the first screen (meaning; play this media on the TV) that in turn retrieves the play command and plays the requested media. The media playback can then be controlled by the companion device. As a note, the companion device can be a desktop computer or a laptop and not only a smartphone as one might think. This use case can be seen in Figure 4.2. Other use cases that can be implemented with Connect can be:

- Replacing the remote control with a companion device application
- Enabling enhanced experiences with a second screen in conjunction with the first screen, such as showing additional information (on the second screen) about the playing media (on the first screen)
- Multi-screen games, for example where people have a personal and a shared view; poker, mahjong etc.

Theoretically any kind of multi-screen experiences can be achieved by using Connect. The Accedo cloud provides messaging channels for devices, and which type of messages that is actually sent is all up to the application developer to implement.

4.1.1 Pairing Process

Connecting devices requires some interaction from the end user. As mentioned before, devices get paired with each other using pairing codes. This can be done either via manually entering the pairing code in the companion device or by scanning a shown QR code on the first screen. Pairing devices manually usually requires the following flow of interaction:

1. Launch first screen application, e.g. a Samsung smart TV application.
2. Navigate to the pairing screen in application, where a pairing code is shown for the user.

3. Launch the second screen application, for example an iPhone application.

4. Input the pairing code in the second screen application and press the "connect" button; and the devices is now paired.

Using the QR code process instead would replace the last step with; scanning the QR code with the camera of the companion device. With the user experience in mind, this alternative is more seamless. The Accedo cloud also provides a persistent pairing state between devices, which means that the connection process only needs to be performed once by the user.

4.2 Technology

Connect is a cloud-based messaging solution, meaning that all communication is done through the Internet. This is the case even if devices should be on the same local network. The Connect solution consists of three main parts: an application powered by Connect, the Connect service, and a third party messaging service. The architecture of Connect can be seen in Figure 4.3. The application need to implement the Connect API for that specific platform, which handles communication with both the Connect service and the third party messaging service. An upgrade to the client library does not require a new deployment of the application; this is described more thoroughly in subsection 4.2.4. The messaging server is the third party server cloud that manages all communication between devices that are connected. The Accedo service provides additional functionalities to the messaging server such as RESTful APIs, pairing features, persistent pairing, and so forth.

4.2.1 Connection States

Devices can have different steps of connection between each other. When the pairing process has been successfully completed, the devices get paired. This is a persistent pairing, until one of the devices consciously sends an unpair request. In other words, devices will still be paired even though they disconnect from the current session. This means that devices also can be connected with each other, meaning that they are online in the communication channel and listens to messages being sent. To clarify; if devices are connected, it implies that they are paired with each other and online in the channel; if devices are paired, they can be either connected or disconnected to their communication channel. A detailed view of the communication flow is shown in Figure 4.4. The application initializes the communication by sending a handshake, and receives a reply withholding a library injection.
4.2. TECHNOLOGY

4.2. ACCEDO CONNECT

Figure 4.3: The architecture of Accedo Connect

The application now performs a lookup against the Accedo service to find out whether a persistent pairing exists or not, whichever the application receives a communication channel to use. As Figure 4.4 also expose, messaging history is sent from the messaging service to the Accedo service which enables analytics.

4.2.2 Messaging

If a successful connection has been made, any kind of serializable messages, for example formatted as JSON data, can be broadcasted within the channel. A note here is that Connect does not support any kind of streaming between devices. So, implementing swipe to play functionality in practice means that the second screen application sends a message to the first screen application telling it to retrieve a certain media stream from a web address.

4.2.3 Host & Clients

Within Connect, devices are divided into host and client devices. A communication channel must contain a host device, which is "responsible" for the channel, implying that the a channel must be initiated by a host. Should that host device unpair from the channel, the channel is going to cease to exist. A host device logically represents the first screen application in the context, usually a TV, and technically this can be an application on for example a smart TV, a game console, a STB, or a Roku device. A client device logically represents the companion device, which usually is a Android or iOS powered device. In theory though, any type of device can acts as a host or a client within Connect.
4.3. IMPLEMENTATION

CHAPTER 4. ACCEDO CONNECT

4.2.4 Library Injections

Many platforms, like the smart TVs or STBs, often run web applications meaning that applications can be developed using HTML, CSS, and JavaScript. But even though one could think that cross-platform development should be fairly easy against such platforms, that is not the case. Each platform often runs its own OS with its own API for reaching features requiring system level calls. To support the variety of platforms that Connect does, the APIs uses library injections from Accedo’s servers. The JavaScript API requests code from the servers and receives code specifically designed for the OS on which the application is running. This means that the platform fragmentation is abstracted away from the application developer. This advantage is a unique selling point for Connect. The application developer can instead focus on the front-end development, resulting in a faster time to market, on multiple platforms with shorter development time.

4.3 Implementation

The Connect SDK includes APIs for JavaScript (e.g. enabling web applications), Java (e.g. enabling Android), Objective-C (e.g. enabling iOS), and BrightScript (e.g. enabling Roku applications) which makes the solution to be implemented on the majority of devices in the market. As this thesis have its focus on the Android platform, the following text will give an introduction to implementing Connect support in a Android application. The purpose is not to give a deep explanation of the API, rather to give a brief
presentation on the inner workings of Connect. In the Android library, there are four objects that the developer needs to manage:

- **Connect**: The main interaction point of the Connect API.
- **ConnectCallback**: Callback interface for Connect method calls.
- **Channel**: Interaction point for a specific communication channel.
- **ChannelCallback**: Receives messages that is broadcasted in the channel.

The callback interfaces contains methods that receive messages from corresponding class. Implementing the Connect APIs consist mainly of two steps, connecting to the Accedo cloud and connecting to a channel.

### 4.3.1 Connecting to the Accedo Cloud

Establishing a connection to the Accedo cloud is done by calling the initialization function of the `Connect` object. Listing 4.1 shows how to achieve this in Java.

```java
private Connect connect;
...
public initConnect()
{
    connect = new Connect(deviceId, apiKey, serverAdress, connectCallback);
    connect.init();
}
```

Listing 4.1: Initialization of the Connect object in Java

The `ConnectCallback` object must be implemented to receive the callback messages sent from the Accedo cloud. When the `ConnectCallback.onInit` method has been called, the Connect library is connected and one can start connect to a specific channel.

### 4.3.2 Connecting to a Channel

The first thing one should do after establishing a connection is to check if there is some existing channel in which the device is already paired with. This is done with calling the `Connect.connect` function. The library will answer with a callback to the `ConnectCallback.onConnect` function as seen in Listing 4.2.

```java
private Channel channel;
...
@Override
public void onConnect(Channel channel, ResponseCode code) {
    if (code.equals(ResponseCode.OK)) {
        this.channel = channel;
        /** Reconnected to a channel **/```
If the ResponseCode has a value of ResponseCode.OK, it means that we managed to reconnect to an already existing persistent pairing that has been established in some previous session. In that case, we can start communicate with the given Channel object. If we did not received an ResponseCode.OK we did not managed to reconnect to some channel, the reason could either be that some error occurred or simply that we did not have a pairing from before. The next step differs for host and client applications; if the application is a host, we should request a new pairing code via Connect.createNewPair that we can show on the screen; if the application is a client, we should show the user a screen where a pairing code can be entered. The client application can then connect to a channel with the Connect.pair(String) function, where the String is the entered pairing code. After Connect.pair(String) or Connect.createNewPair is called, the library answers on the ConnectCallback.onPair function. This can be seen in Listing 4.3. If we receive a ResponseCode.OK it means that we successfully paired with the given channel and are connected with it. If we got paired as a host application we can fetch the pairing code from the Channel.getPairingCode function.

```java
private Channel channel;
...
@Override
public void onPair(Channel channel, ResponseCode code) {
    if (code.equals(ResponseCode.OK)) {
        this.channel = channel;
        /** We got paired and connected to the channel **/
        if (channel.getPairingCode() != null) {
            /** Received a code, got paired as host. **/
        } else {
            /** Didn’t receive code, got paired as client. **/
        }
    } else {
        /** Error **/
    }
}
```

Listing 4.3: The callback method when trying to pair to a channel

With the given channel, the application can send message with the Channel.sendMessage(Object) function, where the valid arguments are either: Array, Map, or String. All applications within the channel will then receive the message on the ChannelCallback.messageReceived(Object, String) where the Object is the message and the String is the identification of the sender of the message. An application also gets notified when other applications join or leave the
current channel. Such events fire a call to the ChannelCallback.presenceCallback(Object) method where the Object is a JSON message containing information of the event.
Chapter 5

Development

This chapter will describe the development process conducted in the thesis. As stated before, the thesis was conducted in parallel with a thesis investigating the possibilities of integrating support for DIAL in Connect. Therefore some figures might include parts relating to DIAL, but this is of no concern within the scope of my thesis.

5.1 Process

The thesis process was highly influenced by the agile development methodology. Both preferred by myself and used by the developers at Accedo. Consequently, all parts were conducted in an incremental and iterative manner. However, the development process can be divided into the following phases:

- Analyzing the pre-study results.
- Specifying the use cases and requirements.
- Developing the architecture and design.
- Implementation of the solution.
- Testing the solution.

In the first weeks of the development process, the pre-studies were performed. The pre-studies included examination of the technologies of the Chromecast and Accedo Connect. The examination of the Google Cast technology started with review of the preview SDK, which was current at that time. Time were spent on examine demo applications that supported Cast and reading preview documentation. Some weeks into the pre-study process, the official SDK was released; happily a pretty good timing with the process of the thesis. However, since the APIs and documentation had changed, the review process had to be done again. With the official SDK
though, more time could be spent on implementing Cast support in test applications. The pre-study process of the Chromecast was highlighted with a visit at a Chromecast developer day, hosted by Google in London, which was a really good experience and contributed a lot to the data collection of the Cast technology.

The examination of the Accedo Connect technology was conducted by reviewing the documentation and development of test applications. This process was pretty straight forward and the opportunity to have first-hand communication with the actual developers of Connect, were really helpful.

5.2 Pre-study Results

After the examination of the technologies, a process of sorting the findings and comparing the technologies was conducted with the goal to find solutions on how to integrate them with each other. Examining the Connect server cloud diagram seen in Figure 4.1 one could draw the conclusion that the Chromecast should be able to be attached to the Accedo cloud in the same way as the other platforms. However one of the main observations of the Cast technology was that it used the local network as a communication medium. Meaning that adding the Cast platform to the Accedo cloud would be unnatural and against the fundamental workings of the Cast technology (even though it theoretically could be possible). The Cast SDK and the Connect SDK both comes with a communication platform. In that aspect, they both solves the same problem, but with different solutions. Connect use the (Accedo) cloud as a medium, whilst Cast uses the local Wi-Fi network. From a companion device viewpoint, it should be able to communicate with both the Accedo cloud and a Cast device depending on what type of devices the end user has in his living room. An image of the solution idea can be seen in Appendix C.

5.2.1 Conceptual Solution Idea

With the previous findings in mind, the idea of building a framework on the companion device in which these two technologies could co-exist was born. The framework should manage the communication with different devices and provide a simple interface where the developer could send and receive messages from them. Application developers could then implement the framework and use the generic functions of a device objects which would be forwarded to the specific API of that device. The framework should be able to detect devices and communicate with them both via the Connect SDK and the Cast SDK. The framework should also be designed in a way that would let developers of the framework to add new technologies and SDKs in the future. This would make the framework acts as an abstraction layer above the technologies, in other words as a wrapper above the Cast.
The idea was discussed with developers, architects and managers at Accedo with positive feedback. More concrete goals for the framework were formulated as follows:

- The framework should be able to detect devices via the Cast SDK and support manual pairing via the Connect SDK.
- The framework should be able to communicate with devices via the Cast API and the Connect API.
- The framework should be generic enough to enable addition of new technologies in the future.
- The framework should provide a simple interface for application developers to interact with.

The decision was made to develop a generic architecture that should be somehow platform independent, meaning that it could be implemented on different companion device platforms. Within the thesis though, Android was targeted as the platform to implement the architecture on. Also, to test the framework and validate the results, a separate set of demo applications should be developed which implemented the framework and highlights the functionalities of it. To clarify, the conceptual idea of the solutions at this point is shown in Figure 5.1.
5.3 Use Cases & Requirements

To concertize the goals for the solution even further, some basic use cases and requirements was formulated. Since the project was performed in a research and development manner, the explicit functions of the solution were developed as the project proceeded. This process was influenced by a lot of feedback from employees at Accedo, both product developers and application developers, but also senior business developers, to make sure that the solution was going to be applicable in a real production context.

To give a better understanding of the solution, it is important to highlight the flow of requirements in the context. The actual application that end users interact with in their Android device is developed by the application developer. The application developers usually have requirements in terms of features and user experience, but also development deadlines. Therefore pre-built functionality, which can be accessible through an open library, is desirable in a product. The framework should have a well-balanced level of abstraction that provides easy interactions in basic implementations whilst still enable specific customizations when necessary. This could be achieved with a generic interface to a device object, which acts as an abstraction above the actual device implementation. Secondly, the products developers have internal requirements on the architecture and design of the solution. A generic architecture with a high separation of concern enables easy modification and extension of the solution in the future. There should exist a possibility to add new technologies, devices, devices with different features, messages, and so forth. Even though such requirements is very valuable in a long-term perspective it is important to focus on scope of the thesis, that is integrating Chromecast and Connect, and not develop an over-engineered solution. The developed use case diagram can be seen in Figure 5.2.

As stated before, the two main features the application developer should be able to do is: discovery devices, and communicate with applications on the device. Discovery of devices includes find devices that can be connected to, retrieving information about those devices, and also select a specific device. Device communication includes, connecting and disconnecting to a device, launch and stop applications, and send messages between the devices. Messages can be playback commands such as play, pause, and stop, as well as custom commands, or requesting application information.

5.4 Architecture

Since the development was conducted in an exploratory and agile manner, development of the architecture was an iterative process. The first draft of the architecture was developed right after the use cases were finished and is shown in Figure 5.3. The most important generalization made in this version was the separation of functionality within the framework, which resulted in the abstractions discovery of devices and communication with devices. This
might seem a bit over-engineering for the scope of Google Cast and Accedo Connect since both techniques already include functionality for both discovering devices and communicate with devices. But as stated earlier, the thesis was conducted in parallel with a thesis investigating the possibilities of using the DIAL protocol for automating the pairing process of Connect. This abstraction made it possible to add new technologies that just supported one of the abstractions, for example the DIAL protocol only support discovery of devices and could therefore be used to discover both devices using Connect or devices using Cast for communication (as stated earlier, this was possible with the initial versions of the Chromecast). To further understand the abstraction of *discovery* and *device*, it can be compared with communication with the OS and application level of a computer system. The *discovery* abstraction is responsible for finding devices, launching and stopping application etc. which is features that an OS usually handle. The *device* abstraction is responsible for managing a specific application that is running on the OS. Therefore the *discovery* mechanism communicates with the OS level of devices, and the *device* abstraction communicates with applications on it. An explanatory figure can be seen in Appendix D.

### 5.4.1 Refining the Architecture

As the development process proceeded, the architecture was refined and specified more in detail. For the sake of readability, the architecture in this report is divided in two parts. The first part covers the discovery mechanism and is shown in Figure 5.4 and each component will be described below.
Figure 5.3: The first draft of the framework architecture

Figure 5.4: The first part of the refined architecture
Connect Handler

The Connect Handler is the main interaction point of the framework. Mainly, this component should handle functions towards the Device Discovery. The Connect Handler holds an instance of every concrete Device Discovery implementation available within the framework and exposes generic methods that the application developer can call to start and stop a scanning of devices. When such method is called, the Connect Handler simply forwards the request to each respective method for every desired concrete Device Discovery class, following the facade design pattern.

Device Discovery

The Device Discovery component is responsible for discovering devices. The discovery can happen manually, as with Connect, or via the Cast SDK, or any other technique that possibly can be added in the future. The concrete Device Discovery classes know how to find devices with their specific technology. For example, the CastDeviceDiscovery knows which functions to call in the Cast API for finding Chromecast’s on the network.

Device

The Device component in the architecture simply represents a device that the application can connect to. The abstract class Device holds methods that an application developer should be able to call on a device. Such methods are managing the connection and communication with the actual device. The concrete devices, as in the Device Discovery case, know how to handle each function call and forward the request to the corresponding API manager. For example, the concrete Device could be a CastDevice which in practice is a Chromecast device wherefore it performs its requests via the Cast API manager. The application developer receives instances of the Device object and can manage those objects as desired. Which type of subclass the Device really is (and therefore also the implementation of it), is not exposed to the application developer. However, he should be able to check which actual hardware device it is, since this probably is interesting in a user experience perspective.

API Managers

The API managers are responsible for the communication with respective APIs. Often, responses from the API are messages that the application developer would like to receive, wherefore the API manager in those cases also is responsible for forwarding those messages. An important note is that, since the framework is supposed to facilitate the communication with the APIs, it might not be appropriate to forward all received messages directly to the application developer. Given the context, the managers could only
forward the most relevant and important ones. To report messages in a consistent way, a utility component is used.

**Utilities**

The Utility component provides functions that are needed in various places throughout the framework. As seen in Figure 5.4, the main class is the BroadcastHelper. The BroadcastHelper contains help methods for broadcasting messages up to the main application in a consistent way, for example to send instances of found devices to the application.

The second part of the refined architecture is covering the communication with a device, as seen in Figure 5.5. There are two new components that are introduced in this part, namely the Command and the State component.

**Command**

The Command component was developed due to the requirement that different devices should be able to handle different commands. For instance, adding methods like Device.play or Device.volumeUp in the abstract Device object would imply that every possible device should support a play or a volume up command, which not always might be the case (in a application with no media playback). The developed solution was designed with one generic function in the Device object, which should handle all commands (Device.sendCommand(Command)). Each type of device wherefore could imple-
ment support for the commands that it supported, and throw an exception when receiving an unsupported command. Encapsulating the commands in objects also gave the possibility to hold more extensive information about the specific command within the object, for instance a play command would probably need some information about the media to play. The concrete subclasses can be instantiated by the application developer and sent to a Device object that he holds. This solution supports new commands to easily be added to the architecture.

State

The State component was introduced to organize the different states that a device could have, like connected or disconnected, in a consistent manner. The State objects encapsulate the data of a specific state and is sent to the application developer. The states can either be requested or sent when the device has changed its state for some reason. The concrete class DeviceState represent the state that the device has, whilst the MediaState represent the state of a media playback and is therefore supposedly only used in those cases.

5.5 Implementation

This section will more thoroughly describe the classes and how the implementation was performed to achieve the functionalities. For the sake of saving place and to highlight the important parts, the code examples in this section are found in the appendix is compressed, comments are removed and unnecessary functions are not shown.

5.5.1 Framework Classes

The base classes of the architecture are those who are not specifically targeted to a technology, in other words the base of the framework. Technologies that are added to the framework must extend DeviceDiscovery and/or Device depending on the functionality of the technology. To keep a separation of concern within the framework, it is also suggested that an API manager is added. For example, adding support for DIAL in the framework should result a subclass of DeviceDiscovery and a class managing the DIAL protocol.

ConnectHandler

The ConnectHandler class is the main interaction point for an application developer. Its main functionality is act as a abstraction layer above the device discovery classes and forward requests from the developer to them. It also provides some initialization arguments and settings that is needed
within the framework to be set. The class follows an singleton pattern to uphold a single instance to interact with.

**DeviceDiscovery**

The device discovery component has its abstract superclass `DeviceDiscovery` that defines the features of the discovery classes. The `DeviceDiscovery` class contains two methods, namely `startActiveScan` and `stopActiveScan`, as seen in figure Listing 5.1. Every concrete device discovery technique must extend the `DeviceDiscovery` class and therefore also implement the methods. The `startActiveScan` method should also make sure that found devices are broadcasted to the application developer via the `BroadcastHelper` class. Note that neither the `DeviceDiscovery` nor its subclasses is visible for the developer.

```java
public abstract class DeviceDiscovery {
    protected DeviceDiscovery() {

    
    public abstract void startActiveScan();
    
    public abstract void stopActiveScan();
}
```

Listing 5.1: The `DeviceDiscovery` abstract superclass

**Device**

The developer receives a set of `Device` objects when they are found via the concrete device discovery classes. The public methods in the `Device` class are therefore the only methods he could call. Whilst there exist functions to determine which type of device it is, the concrete class that the `Device` object actually is, is abstracted away from the developer. The concrete device subclasses must implement the defined functions and act accordingly. The `Device` class can be seen in Listing 5.2 which shows the most important parts of the class. The object hold values for information about the device such as `friendlyName`, `discoveryType` (how it was discovered), and `uuid` (universally unique identifier), which all have getter functions not shown in Listing 5.2. It also provides functions such as `Device.connect`, `Device.reconnect`, and `Device.disconnect`. When a connection is established, the developer can call the `sendCommand(Command)` function to send instances of the `Command` object to request events at the device. The developer receives answers through the `BroadcastHelper` which broadcasts messages within the application.

```java
public abstract class Device implements Serializable {
    ...
    protected String friendlyName;
    protected DiscoveryType discoveryType;
    protected String uuid;
    ...
    public abstract void connect();
```
5.5. IMPLEMENTATION

The abstract class Command is the superclass of all commands within the framework. Command holds a CommandType value that lets concrete Device objects simply check the type of incoming commands via the Command.getCommandType method. The superclass also provides an implementation of the serializable class. Subclasses can be added to the framework as they are needed, a list of currently available commands in the framework can be seen in Table 5.1. A part of the PlayCommand object can be seen in Listing 5.3.

```java
public abstract class Command {
    public abstract void reconnect();
    public abstract void disconnect();
    public abstract void sendCommand(Command command);
    ...
}
```

Listing 5.2: The most important parts of the Device abstract superclass

```java
public class PlayCommand extends Command {
    // ...
    private String url;
    private int position;
    private boolean autostart;

    public PlayCommand() {
        commandType = CommandType.PLAY;
    }

    public PlayCommand(String url, int position, boolean autostart) {
        this();
        this.url = url;
        this.position = position;
        this.autostart = autostart;
    }

    public PlayCommand(String url) {
        this(url, 0, true);
    }
    ...
}
```

Listing 5.3: A part of a PlayCommand object which is sent to a device

State

The abstract class State is the superclass of the state components. As of now, it only provides the subclasses with a implementation of the serializ-
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomMessageCommand</td>
<td>To send a custom message</td>
</tr>
<tr>
<td>MuteCommand</td>
<td>For muting the playback</td>
</tr>
<tr>
<td>PauseCommand</td>
<td>To pause the playback</td>
</tr>
<tr>
<td>PlayCommand</td>
<td>To play some content</td>
</tr>
<tr>
<td>RequestMediaStateCommand</td>
<td>To request the state of the playback</td>
</tr>
<tr>
<td>ResumeCommand</td>
<td>To resume the playback</td>
</tr>
<tr>
<td>SeekCommand</td>
<td>To seek to a position in the playback</td>
</tr>
<tr>
<td>StopCommand</td>
<td>To stop the playback</td>
</tr>
<tr>
<td>VolumeChangeCommand</td>
<td>To change the volume on the playback</td>
</tr>
</tbody>
</table>

Table 5.1: Currently available commands in the framework

There are two implemented subclasses of the State class, namely DeviceState and MediaState. The DeviceState class, which is shown in Listing 5.4, is responsible for describing the state of a device, including the state of the connection and the state of the application running on the device. The MediaState class is responsible for describing the state of a media playback on the device and can be seen in Appendix E.

```java
public class DeviceState extends State {
    private ConnectionState connectionState;
    private ApplicationState applicationState;

    public enum ConnectionState {
        CONNECTED("connected"),
        NOT_CONNECTED("not_connected"),
        CONNECTING("connecting"),
        CONNECTING_FAILED("connecting_failed"),
        UNKNOWN("unknown");
    }

    public enum ApplicationState {
        STOPPED("stopped"),
        RUNNING("running"),
        INSTALLABLE("installable"),
        NOT_FOUND("not_found"),
        UNKNOWN("unknown");
    }
}
```

Listing 5.4: A fraction of the DeviceState class
5.5. IMPLEMENTATION

Utilities

The Utilities component is for now only populated by the class BroadcastHelper which provides a consistent way for concrete device discovery, device and API manager classes to send messages to the application. Listing 5.5 shows one function of the class who provides a consistent way to send found devices to the application.

```java
public class BroadcastHelper {
    public static void broadcastDiscoveryMessage(Boolean isAdded, Device device) {
        Intent intent = new Intent(ConnectHandler.DISCOVERY_NAMESPACE);
        intent.putExtra("isAdded", isAdded);

        Bundle b = new Bundle();
        b.putSerializable("device", device);
        intent.putExtras(b);
        LocalBroadcastManager.getInstance(ConnectHandler.getContext()).sendBroadcast(intent);
    }
}
```

Listing 5.5: A help function of the BroadcastHelper utility class

5.5.2 Google Cast

Adding support for the Chromecast includes implementing the Cast SDK in the framework. With the base of the framework established, subclasses specific for Cast must be added. The sender Cast API includes functionality for both discovering devices and communicating with application on them, wherefore a subclass of both DeviceDiscovery and Device is added. To keep the separation of concern, a CastManager is created responsible for communication with the API.

CastManager

The CastManager class follows the singleton pattern and handles the communication with the Cast API. Basically, it exposes methods that invoke different steps in the connection phases such as: searching for devices, selecting a device, connecting to a receiver, connecting to a receiver application, establishing a communication channel, and so forth. The CastManager itself is not responsible for how it precede these steps. The most relevant parts of the CastManager can be found in Appendix F.

CastDeviceDiscovery

The concrete class for discovering devices through the Cast SDK is the CastDeviceDiscovery and is shown in Listing 5.6. As one can see, the implemen-
tation knows how to achieve the functions and therefore simply forwards
the request to specific methods in the CastManager.

```java
public class CastDeviceDiscovery extends DeviceDiscovery {
    public CastDeviceDiscovery() {
        
        @Override
        public void startActiveScan() {
            CastManager.getInstance().addCallback();
        }

        @Override
        public void stopActiveScan() {
            CastManager.getInstance().removeCallback();
        }
    }
}
```

Listing 5.6: The CastDeviceDiscovery concrete class

**CastDevice**

The CastDevice is the device object representing a device connected through
the Cast API, presumably a Chromecast. As Listing 5.7 shows, the CastDevice
object handles the connection methods, as well as the sendCommand(Command)
function which checks the type of the command to determine what action
to take.

```java
public class CastDevice extends Device {
    ...
    @Override
    public void connect() {
        CastManager.getInstance().connect(this);
    }

    @Override
    public void disconnect() {
        CastManager.getInstance().disconnect();
    }

    @Override
    public void sendCommand(Command command) {
        switch (command.getCommandType()) {
            case PLAY:
                play((PlayCommand) command);
                break;
            ...
            default:
                /** Command type is not supported **/
                
        }
    }

    private void play(PlayCommand cmd) {
        MediaInfo newMedia = cmd.getMediaInfo();
        ...
        CastManager.getInstance().loadMedia(newMedia, true);
    }
}
```

43
5.5. IMPLEMENTATION

5.5.3 Accedo Connect

The Connect SDK, similar to the Cast SDK, contains functionality to both discover and communicate with devices, even though the discovery is done manually. Therefore the same type of classes is created for Connect.

ConnectManager

The ConnectManager as one might think is interacting with the Connect API. In the same way as the CastManager works, it exposes methods to reach the functions of the actual API. A current limitation of today’s Connect architecture is that it is host dependent. In other words, a channel must contain a host and therefore a host is the application that first initiates a new channel. As stated before, a host logically represents the first screen application, but theoretically is not bound to any specific platform implementation. Usually this host/client division in not a problem for application developers (since applications usually is either a host or a client), which was also the case within the scope of this thesis. But as the parallel performed thesis regarding DIAL integration was implemented on the framework as it was developed, it was necessary for the device to be able to act as both a host and a client. Some of this logic was implemented within the ConnectDevice class and some in the ConnectManager, none of which is in the scope of this thesis.

ManualDeviceDiscovery

The ManualDeviceDiscovery class can be seen in Listing 5.8. The naming of ManualDeviceDiscovery was chosen due to the fact that devices actually not are found; the end user needs to manually connect them by himself (by entering the pairing code that is shown on the first screen). Therefore the application needs a device object to represent that possible connection. Since this device always should be possible to connect to, the startActiveScan function always sends a device directly to the application.

```java
public class ManualDeviceDiscovery extends DeviceDiscovery {
    public ManualDeviceDiscovery () {}

    @Override
    public void startActiveScan () {
        ConnectDevice connectDevice = new ConnectDevice(
            "Manual Device",
            Device.DiscoveryType.MANUAL, "manual" );
    }
}
```

Listing 5.7: The CastDevice concrete class
BroadcastHelper.broadcastDiscoveryMessage(true, connectDevice);

@Override
public void stopActiveScan(){}

Listing 5.8: The concrete class ManualDeviceDiscovery

ConnectDevice

The ConnectDevice represent a device that runs a application powered by Connect. As the Connect API does not include any pre-built channel for media playback messages, these commands must be handled separately, as seen in Listing 5.9. The command objects is converted to a String with a message in the JSON format to; in this case describe a play command. The message must be interpreted by the receiver application itself.

Listing 5.9: The ConnectDevice concrete class

5.5.4 The DIAL Protocol

An example of a technique that not covers both discovery of devices and communication with them is the DIAL protocol. To add DIAL support in the
framework one would only need to implement a subclass of the DeviceDiscovery
class and attach those functions to the necessary actions. This is a good
example of the extendibility of the framework.

5.6 Demo Applications

Since the framework should be used by application developers, it seemed
logical to test the framework by developing a set of demo applications im-
plementing it. It should also function as a guide on how to implement
the framework. Since the companion device application for Android was
in focus, the most resources were given to that implementation, in terms
of development time. Therefore the implementation of first screen applications
using the Cast SDK and Connect SDK was not prioritized. Using a default
media receiver, or a styled media receiver, as the receiver application on the
Chromecast was therefore a good alternative, since they are pre-built. For
Connect, Accedo already had demo applications, so there was no need to
develop a new one.

5.6.1 Android Application

The Android application was developed with a specific use case in mind;
media playback. Even though the framework supports a variety of use cases,
the most implemented one (by Accedo) is just that. The aim was to build
an application that simply showed the strength of the framework.

The following listings will highlight the code relevant to the framework in
the developed demo application. The first thing the developer need to do is
to initialize the ConnectHandler and this is done via calling the ConnectHandler.
initialize(Context, String) function. The Context object is used by the Cast API,
and the String is the application ID retrieved from the Cast SDK Developer
Console, which is needed to communicate with a specific Chromecast on the
network. After that, the developer can start to scan for devices which can
be seen in Listing 5.10.

```java
@override
protected void onResume() {
    super.onResume();
    LocalBroadcastManager.getInstance(applicationContext).
        registerReceiver(mBroadcastReceiver, new IntentFilter(
            mConnectHandler.DISCOVERY_NAMESPACE));
    mConnectHandler.startActiveScan();
}
```

Listing 5.10: Start the scanning for devices and register a receiver for the
callback

To receive devices that the framework finds, the developer registers
a BroadcastReceiver at the LocalBroadcastManager with the given namespace
ConnectHandler.DISCOVERY_NAMESPACE, on which the framework will report
5.6. DEMO APPLICATIONS  CHAPTER 5. DEVELOPMENT

found devices. To actually start the scanning; the method ConnectHandler.
startActiveScan is called. To catch the found devices, an implementation of
the BroadcastReciver is necessary. This can be seen in Listing 5.11.

```
private BroadcastReceiver mBroadcastReceiver = new
    BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            Device device = (Device) intent.getExtras().
                getSerializable(Device.NAME);
            if (intent.getBooleanExtra(Device.ADDED, true)) {
                deviceList.put(device.getFriendlyName(), device);
            } else {
                deviceList.remove(device.getFriendlyName());
            }
        }
    }
```

**Listing 5.11**: The broadcast receiver that retrieves found devices

The Device object can be retrieved from the intent as well as a Boolean
describing if the device was found or if it was lost (the framework lost
its connection to it). The application itself is responsible for holding the
devices in some kind of list and presents them for the user. When the
user has selected a device, the application can connect to it, as shown in
Listing 5.12.

```
public void onDeviceSelected(Device device) {
    if (device.getDiscoveryType().equals(Device.DiscoveryType.
        MANUAL)) {
        showPairingCodeDialog();
    } else {
        device.connect();
    }
}
```

**Listing 5.12**: Connecting to a device

If the selected device was a device that needs to be paired manually, the
user should be prompted to enter a pairing code. Otherwise, the device can
be connected by simply calling the Device.connect method. To receive mes-
sages from a Device, the developer needs to register a BroadcastReceiver with
the ConnectHandler.DEVICE_NAMESPACE. On that namespace, the framework
will report statuses from the device in forms of DeviceState objects and also
MediaState objects describing a media playback. An implementation of this
receiver can be seen in Appendix G. To send messages to the device, the
Device.sendCommand(Command) method can be used. As seen in Table 5.1,
there are a range of available commands that can be sent to the device.

In the developed demo application, the flow of interaction in the appli-
cation is as follows:

- The application presents a list of found devices on the network, as seen
  in Figure 5.6a. The user then selects a device to connect to.
5.6. DEMO APPLICATIONS

(b) Selecting a device

(c) Remote playback

Figure 5.6: Screenshots of the demo application

- When the device is connected to the application it shows a list of videos to the user, as seen in Figure 5.6b. The user selects a video to play.

- The application plays the selected video on the selected device and shows a controller view on the application, as seen in Figure 5.6c. The user can control the playback via the play/pause button and the seek bar.

An observant reader might see a found Roku device in conjunction with a DIAL symbol in Figure 5.6a. This is a implementation from the parallel performed thesis examining the DIAL protocol. Selecting that device would result in a user experience exactly the same as if the user would have chosen a Chromecast (except that the video is played on the Roku device).
Chapter 6

Result & Analysis

The Chromecast is a simple way to enable multi-screen experiences in applications on today’s major platforms. Developing a basic sample application is relatively easy and does not require that many lines of code. Google provides extensive documentation in form of implementation guides, user experience guidelines, API documentation, sample applications, support communities, and so forth, to help developer implement support for Cast. However, the complexity increases when implementing support for the Cast technology in a production context. At first, adding support within an already existing media application presumably requires adding a new state throughout the whole application representing that it is currently managing playback on a Chromecast device. Secondly, support for subtitles, level 1 DRM, different audio tracks in videos, and so forth, is quite limited at the moment. Such functionalities are often desired by CPs.

With that in mind, adding support for the Chromecast in the Accedo Connect solution turned out to not be as straight forward as one might think. Since the communication with a Chromecast device is not performed over the cloud as with Connect, the problem introduced a new dimension of the communication techniques within Connect. Because of that, a new solution was developed, with a much higher complexity than a regular application implementation with the Cast SDK. The result is a proposed framework that acts as a wrapper above the Cast and Connect SDKs and provides generic functions that is independent from the used technology. The result can be analyzed in different viewpoint, mainly from a framework developer and application developer viewpoint, but also from a perspective of an end user.

6.1 Framework Development

From Accedo’s point of view, this framework is a first approach to include and manage communication technologies over the local network within the
Accedo Connect solution. The architecture of the framework provides a solid base and starting point that can be further enhanced to reach a state ready for production. The framework is built with a generic architecture and with a separation of concern in mind, which provides the possibility to add and modify parts of it in the future.

The device and discovery abstractions differentiates the two types of technologies within the framework; technologies for discover devices, and technologies for communicating with devices. Some technologies provides both functionalities (e.g. the Cast SDK), whilst others provides just discovery (e.g. the DIAL protocol) or device communication (e.g. a WebSocket connection). All of which can be supported in the framework. This type of separation is also seen in other solutions in the market, for example in the Cast SDK and in the Samsung MultiScreen SDK (Samsung Electronics Co., Ltd., 2014).

6.2 Application Development

From a viewpoint of an application developer that uses the proposed framework, it provides an easy way for supporting both Accedo Connect and Chromecast in the same application. The framework is generic and can be re-used in multiple projects which help improving the time to market of specific application implementations.

To provide an easy implementation, it is important that the framework have a well-balanced level of abstraction. Abstract ”unnecessary” parts of the APIs that the application developer always implement the same way, is probably desirable; while parts that is customized for each implementation must be sufficient exposed. These kinds of considerations can easily become quite complex problems if the technologies that are being supported by the framework increases. For instance, the framework need to have consistent error handling and expose errors that is made by the application developer whilst hide errors that is not his responsibility. Whether the proposed framework have a good abstraction level is hard to conclude without implementing the framework in at least a couple of applications in a production context.

To be as generic as possible, the proposed framework does not include any front-end specific artifacts. The application’s graphical user interface (GUI) is not affected by the framework and the user experience can be customized as desired for each specific companion application implementation.

Something that has not been considered within the scope of this thesis is the possibility that different technologies within the framework could have different capabilities and limitations. For instance, one technology might support to remotely change the volume, while another one might not. Taking care of this would require further a level of complexity within the framework. Supporting a lot of technologies and capabilities though, might not be desirable since it implies other uncertainties. It would be harder for
an application developer to ensure, for instance, that the performance is sufficient on all the supported hardware’s.

6.3 End User Perspective

In a viewpoint from the end user, the user experience of an application is highly affected by the GUI. What the framework can provide though, is a solution that does not limit any possibilities to build whatever application that the end user wants. As stated in subsection 2.1.1, the four main usage areas for companion device applications are: control, enrich, share, and transfer. So what type of communication is necessary to achieve these use cases? Controlling the first screen application basically requires a one-way communication system, enabling the companion device to send commands like ”channel up” or ”volume up” and so forth. Enriching (showing additional information on the companion device) and sharing usage areas requires the companion device to know what the user is consuming on the first screen. The first screen application therefore needs to be able to broadcast information to companion devices. To achieve a rich cross-device media consumption experience, the transferring use case, the devices probably would need to synchronize the status of the media playback to some account on some cloud service, to be able to freely switch devices. Combining these requirements, a solution that wants to achieve the four main usage areas should support full-duplex communication (two-way communication) between applications and be connected to the Internet. Since the proposed framework does not limit any functionality of neither the Chromecast nor Connect, and the technologies themselves support full-duplex communication, the framework therefore can be used to achieve these use cases.
Chapter 7

Discussion

Companion device interaction has become a hot topic in the market and the interest has grown noticeable even during the time of the thesis. The release of the Chromecast has been a great factor contributing to that interest. The Chromecast provides an easy way (compared to previous solutions) for CPs and developers to extend their already existing applications to include a multi-screen experience. With the all this new attention from the market, one realizes that not everyone have, or might not want, a Chromecast device in their home. There already are a lot of different STBs, smart TVs and game consoles in peoples living rooms which already are connected to the Internet. So, end users already have the hardware and are now also starting to grasp the concept of multi-screen experiences, especially the swipe to play use case. As a CP, enabling these use cases for their content would increase the interest of their services, which is something they are starting to realize as well. But building a good multi-screen experience includes taking a variety of factors in consideration. Both in terms of user experience such as, where should the user interactions take place? And how do we distribute the content over the screens? But also in terms of technical issues like, how should the devices communicate? And which platforms are we supporting?

7.1 End User Interaction Point

As for now, I think the most complex part is to deliver an experience that reach out to the majority of customers while in the meantime provide a consistent user experience throughout the applications. Reaching a lot of customers in the market today implies being available on multiple platforms, which itself is a resource consuming task. Multiple platforms imply different hardware and OSs which potentially could be a big problem when trying to deliver a seamless and consistent experience that the users intuitively know how to interact with.

One of the main issues as I see it, is the issue of where the main in-
The interaction point should be for these kinds of experiences, for instance where the discovery of content and controlling of the playback take place. Some big players in the market want the main interaction point to be on the first screen, typically the TV manufacturers. While other players want the main interaction to be on the second screen, as we have seen in this thesis with Google’s Chromecast ecosystem. Which one that is the most intuitive and best approach is up to research and end users to decide. But as the situation is right now, developers are going to try to combine the approaches, for instance as I did in this thesis. Technically this is totally doable, but there are going to be ambiguities in the user experience which can be confusing for the end user. For example, the developed demo application in this thesis could send a video to both a Chromecast and a Roku device, but the Roku device could also be controlled by a remote control which was not the case with the Chromecast. The issue might not be that apparent in such simple experience, but assume that the Roku device ran a full-fledged first-screen application while the Chromecast still ran a simple video player, and the difference might be more obvious. I personally think that this is something that developers should have in mind when developing experiences that tries to achieve such overlap between different platforms.

7.2 Communication Technique

As described earlier, for a multi-screen solution to be future-proof, I believe it should support a full-duplex communication system between devices. Each device should be able to tell its surrounding which state the device currently has. For instance, if I want to build a set of applications that supports synchronization of media playback between my devices, each device should be able to update the progress of the playback to enable seamless switching between devices. Full-duplex communication also is desirable in a multi-user environment, where users enter and leaves a context frequently. For example, a device enters a room where a video is playing on the TV and gets notified about its playback state.

Using the cloud as a communication medium, as with Accedo Connect, is a solution that fundamentally supports a lot of platforms since most devices are connected to the Internet. However, looking logically at the problem is seems like an ineffective and inefficient way to manage communication between devices that usually are on the same Wi-Fi network. The issue with communication techniques on the Wi-Fi networks is that the interoperability often is inadequate, especially in the TV market where many of the manufacturers promote their own standards. The solution would probably need to be initiated by some third party player. The proposed solution is an initial step for Accedo to take more advantage of the end user’s home network to enhance performance and the user experience, for instance with automatic device discovery.
7.3 Alternative Solutions

The proposed solution in this thesis is of course not the only way that the problem could have been solved. One solution could have been to use the Android’s Media Router framework in a more extensive way. The Media Router APIs is Android’s own framework for sending media to other media playback devices. Figure 7.1 shows a high-level view of the framework and as one can see, the Google Cast itself implements the Media Router APIs. Adding the Accedo Connect Android API to the Media Router framework could end up in a quite seamless way of finding and selecting devices. Examining the way the Cast SDK has been implemented, selecting a Cast device results in using the Cast API, and selecting a Connect device presumably would result in using the Connect API. This would not end up in a consistent interface, as desired in the requirements, for the application developer who still would need to implement both APIs. Also, since this Media Router framework is Android specific it would probably be hard to re-use the architecture in other companion device platforms.

The relevance of the framework’s purpose and architecture is highly validated by a newly released solution by LG (it was released during the concluding phase of the thesis, when the framework in this thesis already was finalized), namely the LG Connect SDK (LG Silicon Valley Lab, 2014). The LG Connect SDK is a framework built on Android and iOS to ease connection to multiple TV platforms, currently the Google Chromecast, Amazon Fire TV, Roku, webOS devices, and the latest versions of LG’s smart TV. The LG Connect SDK aims to provide a simple way of enable the swipe to play use case to those platforms and more to come. Their solution is open source and free to use which lets us examine their architecture, which contains four main components; discovery, device, capabilities, capability listeners, sessions, and information objects (all components consist of a set of classes). Their discovery and device components are fundamentally the same as with my proposed framework, of course there are implementation differences but the purpose is the same. Their information objects component basically corresponds to the state component in my framework. The other components capabilities, capability listeners, and sessions are a result.
of a more extensive support for various technologies. For example, the *capabilities* component is responsible for describing what capabilities devices have, which clearly becomes a necessary component when supporting more devices. My intentions with this comparison is not to be able to draw some conclusions about which framework is the better (obviously there is an enormous difference in resources in place), but rather to highlight the relevance of these frameworks and to validate the result of my thesis.

### 7.4 Future Work

As described before, the interest for companion device interactivity is growing and will probably be keep growing as the use cases becomes more apparent for end users. In such growing areas, as usual, many player aims for developing new cool features just because they can, without any thoughts on what value it will provide for end users. I suggest that more focus should be placed on user experience and user acceptance; what do the users really want? Are they going to use the new features? I also suggest more research that focuses on, and maybe compares, the difference in positioning of the main interaction point for multi-screen experiences; which approach is the most intuitive? What happens if the context changes, for example if the amount of first screen devices increases? How to cope with and control an *Internet of Things* in the living room?

Also, I suggest that the technical aspect of inter-device communication is further explored. How can we integrate more types of devices and technologies, with different capabilities? How can we combine communication technologies that are based on the cloud and the local network (for instance use the cloud as a backup solution)? Evaluating the approach chosen by LG with their Connect SDK in an *Internet of Things* perspective would also be interesting.
Chapter 8

Conclusion

Within this thesis, I have proposed a framework for integrating the Google Chromecast and the cloud-based messaging service that is Accedo Connect, in the same solution. The proposed framework provides a way for application developers to build multi-screen experiences that supports both Chromecast devices and devices using Accedo Connect. Communication with the different devices is done via the technologies own SDKs whilst providing a generic interface against the application developer, who does not need to know which type of device he is actually communicating with. The reason for this abstraction is to ease development of specific application implementations, which enables shorter development time. The framework should not be seen as a production ready solution, rather it should be seen as one of many answers to the question stated in the problem definition of this report (section 1.3), namely: How can the Google Cast technology be integrated with a cloud-based messaging solution?

As stated in section 2.3, today’s companion device applications fails at succeeding since the challenge of controlling all devices in the living room haven’t been addressed enough. The proposed framework for sure does not supports controlling all devices in the living room, but it is a first approach to support multiple technologies, without creating a new one, while in the meantime facilitate the use of them.

A fundamental issue that arises with the integration of these two technologies is the difference of where the main interaction point for the end user is. The Chromecast relies on a companion device on which all interactions, such as browsing content and controlling the playback, is performed. To the contrary, the smart TVs have the main interaction point on the big screen with their own remote controls. Developing multi-screen experiences that overlaps this fundamental fragmentation needs a high consideration of the user experience throughout the set of applications. Having inconsistent interaction points will confuse the end user and result in a bad experience.
Bibliography


Koushik Dutta and Leon Nicholls. Discussion about Chromecast and mDNS. Website, March 2014. https://plus.google.com/110558071969009568835/posts/XFpua1bZTMJ.


Appendix A

The Google Chromecast

Figure A.1: The Google Chromecast (Google Inc., 2014d)
Appendix B

RemoteMediaPlayer Initialization

```java
mRemoteMediaPlayer = new RemoteMediaPlayer();
new RemoteMediaPlayer.OnStatusUpdatedListener() {
    @Override
    public void onStatusUpdated() {
        MediaStatus mediaStatus = mRemoteMediaPlayer.getMediaStatus();
        boolean isPlaying = mediaStatus.getPlayerState() == MediaStatus.PLAYER_STATE_PLAYING;
        ...
    }
};

mRemoteMediaPlayer.setOnMetadataUpdatedListener(
    new RemoteMediaPlayer.OnMetadataUpdatedListener() {
        @Override
        public void onMetadataUpdated() {
            MediaInfo mediaInfo = mRemoteMediaPlayer.getMediaInfo();
            MediaMetadata metadata = mediaInfo.getMetadata();
            ...
        }
    });

try {
    Cast.CastApi.setMessageReceivedCallbacks(mApiClient, mRemoteMediaPlayer.getNamespace(), mRemoteMediaPlayer);
} catch (IOException e) {
    /** Exception while creating media channel **/
}
```

Listing B.1: Initialization of the RemoteMediaPlayer channel
Appendix C

Solution Idea

Figure C.1: Idea of what the framework should achieve
Appendix D

Discovery and Device Abstraction

Figure D.1: The discovery and device abstraction compared to the communication within a computer system
Appendix E

The MediaState Object

```java
public class MediaState extends State {
    ... 
    private MediaInfo mediaInfo;
    private long playbackPosition;
    private long mediaDuration;
    private PlaybackState playbackState;
    private PlaybackStateIdleReason playbackStateIdleReason;

    public enum PlaybackState{
        PLAYING("playing"),
        PAUSED("paused"),
        BUFFERING("buffering"),
        IDLE("idle"),
        UNKNOWN("unknown");

        private String state;

        private PlaybackState(String state){
            this.state = state;
        }

        @Override
        public String toString(){
            return state;
        }
    }

    public enum PlaybackStateIdleReason{
        NONE("none"),
        FINISHED("finished"),
        CANCELED("canceled"),
        INTERRUPTED("interrupted"),
        ERROR("error");

        private String reason;

        private PlaybackStateIdleReason(String reason){
            this.reason = reason;
        }
    }
}
```
@Override
public String toString()
{
    return reason;
    ...}

public MediaState(MediaInfo mediaInfo, long playbackPosition,
        long mediaDuration,
        PlaybackState playbackState,
        PlaybackStateIdleReason playbackStateIdleReason)
{
    this.mediaInfo = mediaInfo;
    this.playbackPosition = playbackPosition;
    this.mediaDuration = mediaDuration;
    this.playbackState = playbackState;
    this.playbackStateIdleReason = playbackStateIdleReason;
}

public MediaState(String title, String subTitle, String studio,
        String url, String imgUrl, String bigImageUrl,
        long playbackPosition, long mediaDuration,
        PlaybackState playbackState)
{
    this(buildMediaInfo(title, subTitle, studio, url, imgUrl, bigImageUrl), playbackPosition,
            mediaDuration, playbackState,
            PlaybackStateIdleReason.NONE);
    ...}

Listing E.1: The MediaState object
Appendix F

The CastManager Class

```java
public class CastManager {

    ...

    /** Step 1; Initialization of the Cast Manager **/

    /**
     * Creates a new CastManager instance and adds the Cast
     * category to the MediaRouteSelector
     */
    private CastManager () {
        mMediaRouter = MediaRouter.getInstance(mContext);
        mMediaRouteSelector = new MediaRouteSelector.Builder()
            .addControlCategory(CastMediaControlIntent.
                categoryForCast(mApplicationId))
            .build();
        mMediaRouterCallback = new MyMediaRouterCallback();
    }

    /**
     * Initializes the CastManager
     * @param context Should be the application context of the
     * application
     * @param applicationId
     */
    public static void initialize(Context context, String
        applicationId){
        mContext = context;
        mApplicationId = applicationId;
        if (mCastManager == null){
            mCastManager = new CastManager();
        }
    }

    /**
     * Gets the instance of the CastManager singleton object
     * @return null if the CastManager is not initialized
     */
    public static CastManager getInstance(){
        return mCastManager;
    }
}
```
/** Step 2: Discovering and selection of routes **/

/**
 * Starts searching for devices, should be called in the activity’s onResume
 */
public void addCallback()
    {  
        mMediaRouter.addCallback(mMediaRouteSelector, 
                                mMediaRouterCallback, 
                                MediaRouter.CALLBACK_FLAG_PERFORM_ACTIVE_SCAN);
    }

/**
 * Stops the searching for devices, should be called in the activity’s onPause
 */
public void removeCallback()
    {
        mMediaRouter.removeCallback(mMediaRouterCallback);
    }

/**
 * Callback class for the MediaRouter
 */
private class MyMediaRouterCallback extends MediaRouter.Callback {
    @Override
    public void onRouteSelected(MediaRouter router, MediaRouter.RouteInfo info) {
        mGoogleCastDevice = CastDevice.getFromBundle(info.getExtras());
        connectToReceiver();
    }

    @Override
    public void onRouteUnselected(MediaRouter router, MediaRouter.RouteInfo info) {
        teardown();
        mGoogleCastDevice = null;
        mAccedoCastDevice = null;
    }

    @Override
    public void onRouteAdded(MediaRouter router, MediaRouter.RouteInfo info) {
        tv.accedo.chromecast.connectlibrary.device.cast.CastDevice cd = 
                                deviceFactory(info);
        BroadcastHelper.broadcastDiscoveryMessage(true, cd);
    }

    public void onRouteRemoved(MediaRouter router, MediaRouter.RouteInfo info) {
        tv.accedo.chromecast.connectlibrary.device.cast.CastDevice cd = 
                                deviceFactory(info);
BroadcastHelper.broadcastDiscoveryMessage(false, cd);

/**
 * Connect to a Google Cast device. Reconnects if possible.
 * @param accedoCastDevice
 */
public void connect(tv.accedo.chromecast.connectlibrary.
device.cast.CastDevice accedoCastDevice){
    if (mAccedoCastDevice != null || mGoogleCastDevice != null){
        /** Not first device selected **/
        if (mAccedoCastDevice.equals(accedoCastDevice)){
            /** Same device selected **/
            DeviceState state = new DeviceState(DeviceState.
            ConnectionState.CONNECTING);
            state.getConnectionState().setMessage("Reconnecting");
            BroadcastHelper.broadcastDeviceStateMessage(state);

            if (isConnectedToReceiver()){  
                /** We are connected **/
                if (isChannelConnected()){  
                    /** Sane device is already connected **/
                    BroadcastHelper.
broadcastDeviceStateMessage(new
                    DeviceState(
                    DeviceState.ConnectionState.
                    CONNECTED, DeviceState.
                    ApplicationState.RUNNING));
                } else{
                    launchReceiverApplication();
                }
            } else{  
                connectToReceiver();
            }
        } else{  
            teardown();
        }
        mAccedoCastDevice = accedoCastDevice;
        mMediaRouter.selectRoute(mAccedoCastDevice.
        getMediaRouteInfo());
    }
}

/** Step 3; Connection/Reconnection to the receiver **/
/**
 * Connects to the receiver. Can be called to reconnect
 * {@link com.google.android.gms.common.api.GoogleApiClient}
 * if it has lost its connection.

private void connectToReceiver() {
    try {
        /** The Cast.Listener callbacks are used to inform the sender application about receiver application events */
        mCastListener = new Cast.Listener() {
            @Override
            public void onApplicationDisconnected(int errorCode) {
                BroadcastHelper.broadcastDeviceStateMessage(
                        new DeviceState(DeviceState.ConnectionState.NOT_CONNECTED));
                teardown();
            }
            @Override
            public void onApplicationStatusChanged() {
                /** onApplicationStatusChanged **/}
            }
            @Override
            public void onVolumeChanged() {
                /** onVolumeChanged **/
            }
        };
        /* Initializing connections callbacks */
        mConnectionCallbacks = new ConnectionCallbacks();
        mConnectionFailedListener = new ConnectionFailedListener();

        /** The Cast SDK APIs are invoked using GoogleApiClient */
                                    .CastOptions
                                    .builder(mGoogleCastDevice, mCastListener);

        mApiClient = new GoogleApiClient.Builder(mContext)
                        .addApi(Cast.API, apiOptionsBuilder.build())
                        .addConnectionCallbacks(mConnectionCallbacks)
                        .addOnConnectionFailedListener(
                                mConnectionFailedListener)
                        .build();

        /** Establishing a connection using the GoogleApiClient instance */
        mApiClient.connect();
    } catch (Exception e) {
        BroadcastHelper.broadcastDeviceStateMessage(

new DeviceState(DeviceState.ConnectionState.CONNECTING_FAILED));
}

/**
 * Class for handling callbacks regarding the connection to the cast device
 */
private class ConnectionCallbacks implements GoogleApiClient.ConnectionCallbacks {
    @Override
    public void onConnected(Bundle connectionHint) {
        if (mApiClient == null) {
            BroadcastHelper.broadcastDeviceStateMessage(
                new DeviceState(DeviceState.ConnectionState.CONNECTING_FAILED));
            return;
        }

        try {
            /** Check if onConnectionSuspended has been called **/
            if (mWaitingForReconnect) {
                mWaitingForReconnect = false;
                /** Check if the receiver app is still running **/
                if ((connectionHint != null)
                        && connectionHint.getBoolean(Cast.EXTRA_APP_NO_LONGER_RUNNING)) {
                    teardown();
                    BroadcastHelper.broadcastDeviceStateMessage(
                        new DeviceState(DeviceState.ConnectionState.CONNECTING_FAILED));
                } else {
                    Cast.CastApi.setMessageReceivedCallbacks(
                        mApiClient, mRemoteMediaPlayer.getNamespace(), mRemoteMediaPlayer);
                }
            } else {
                launchReceiverApplication();
            }
        } catch (Exception e) {
            BroadcastHelper.broadcastDeviceStateMessage(
                new DeviceState(DeviceState.ConnectionState.CONNECTING_FAILED));
        }
    }
}

//
@Override
public void onConnectionSuspended(int cause) {
    mWaitingForReconnect = true;
    if (cause == GoogleApiClient.ConnectionCallbacks.CAUSE_SERVICE_DISCONNECTED) {
        /* Failed to connect to receiver, service disconnected */
    } else if (cause == GoogleApiClient.ConnectionCallbacks.CAUSE_NETWORK_LOST) {
        /* Failed to connect to receiver, network lost */
    } else {
        /* Failed to connect to receiver, unknown cause */
    }
}
/**
 * Class for handling failed connection callbacks from the cast device
 */
private class ConnectionFailedListener implements GoogleApiClient.OnConnectionFailedListener {
    @Override
    public void onConnectionFailed(ConnectionResult result) {
        teardown();
        BroadcastHelper.broadcastDeviceStateMessage(
            new DeviceState(DeviceState.ConnectionState.CONNECTING,FAILED));
    }
    ...
}
/** Step 4; Launching the application on the receiver **/
/**
 * Launches the receiver application. This method will always attempts a join first (do not
 * relaunches the application if it already is running).
 */
private void launchReceiverApplication() {
    if (mLaunchApplicationCallback == null) {
        mLaunchApplicationCallback = new LaunchApplicationCallback();
    }
    Cast.CastApi.launchApplication(mApiClient, mApplicationId, false)
        .setResultCallback(mLaunchApplicationCallback);
/** 
 * Callback class for the launch application request 
 */
private class LaunchApplicationCallback implements 
ResultCallback<Cast.ApplicationConnectionResult> {

    @Override
    public void onResult(Cast.ApplicationConnectionResult result) {
        Status status = result.getStatus();
        if (status.isSuccess()) {
            /** Launching the communication channel to the 
             * application **/
            launchChannel();
        } else {
            /** Teardown the connection **/
            teardown();
        }
    }

    /** Step 5; Launching a communication channel to the 
     * application **/
    ...

    /** 
     * Start the receiver application channel 
     */
    private void launchChannel() {
        /** Using a RemoteMediaPlayer, a channel pre-built for 
         * media messages **/
        mRemoteMediaPlayer = new RemoteMediaPlayer();
        mRemoteMediaPlayer.setOnStatusUpdatedListener(
            new RemoteMediaPlayer.OnStatusUpdatedListener() {
                @Override
                public void onStatusUpdated() {
                    if (!isChannelConnected()) return;
                    MediaStatus mediaStatus = mRemoteMediaPlayer.getStatus();
                    if (mediaStatus != null) {
                        broadcastMediaStatus(mediaStatus);
                    }
                }
            });
        mRemoteMediaPlayer.setOnMetadataUpdatedListener(
            new RemoteMediaPlayer.OnMetadataUpdatedListener() {
                @Override
                public void onMetadataUpdated() {
                    if (!isChannelConnected()) return;
                    MediaStatus mediaStatus = mRemoteMediaPlayer.getStatus();
                }
            });
    }
}
APPENDIX F. THE CASTMANAGER CLASS

```java
MediaInfo mediaInfo = mRemoteMediaPlayer.getMediaInfo();
if (mediaInfo != null && mediaStatus != null) {
    broadcastMediaStatus(mediaStatus);
}
```

```java
try {
    /* Initializing the RemoteMediaPlayer channel */
    Cast.CastApi.setMessageReceivedCallbacks(mApiClient,
        mRemoteMediaPlayer.getNamespace(),
        mRemoteMediaPlayer);

    /* Telling the android app that it is connected to the receiver app */
    BroadcastHelper.broadcastDeviceStateMessage(
        new DeviceState(
            DeviceState.ConnectionState.CONNECTED,
            DeviceState.ApplicationState.RUNNING));
} catch (IOException e) {
    /* Exception while creating media channel */
}
```

```java
/* Step 6; Sending messages to the application */

/* Loads a media to the cast device
 * @param mediaInfo
 * @param autoPlay
 */
public void loadMedia(MediaInfo mediaInfo, boolean autoPlay)
{
    mContext = ConnectHandler.getContext();
    if (!checkIfConnected()) return;
    try {
        mRemoteMediaPlayer.load(mApiClient, mediaInfo,
            autoPlay)
            .setResultCallback(new ResultCallback<
                RemoteMediaPlayer.MediaChannelResult>() {
                @Override
                public void onResult(RemoteMediaPlayer.
                    MediaChannelResult result) {
                    if (result.getStatus().isSuccess()) {
                        if (!checkIfConnected()) return;
                        MediaStatus mediaStatus =
                            mRemoteMediaPlayer.
                            getMediaStatus();
                        if (mediaStatus != null) {
```
/**
 * Plays the current media, if any
 */

public void playMedia()
{
    if (!checkIfConnected()) return;
    try {
        mRemoteMediaPlayer.play(mApiClient);
    } catch (IOException e) {
        /** Failed to send play command **/
    }
}

/**
 * Pauses the current media, if any
 */

public void pauseMedia()
{
    if (!checkIfConnected()) return;
    try {
        mRemoteMediaPlayer.pause(mApiClient);
    } catch (IOException e) {
        /** Failed to send play command **/
    }
}

/** Step 7; Disconnecting the receiver **/

/**
 * Disconnects the cast device
 */

public void disconnect()
{
    teardown();
}

Listing F.1: A fraction of the CastManager class
Appendix G

A BroadcastReceiver Implementation

private BroadcastReceiver mDeviceReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.hasExtra(DeviceState.NAME)) {
            DeviceState deviceState = (DeviceState) intent.getSerializableExtra(DeviceState.NAME);
            if (deviceState.getConnectionState() == DeviceState.ConnectionState.CONNECTED) {
                mSelectedDevice.sendCommand(new RequestMediaStateCommand());
            }
        }
        if (intent.hasExtra(MediaState.NAME)) {
            MediaState mediaState = (MediaState) intent.getSerializableExtra(MediaState.NAME);
            mMediaPosition = mediaState.getPlaybackPosition();
            mMediaDuration = mediaState.getMediaDuration();
            mMediaState = mediaState.getPlaybackState();
            mMediaStateIdleReason = mediaState.getPlaybackStateIdleReason();
            updateMediaState();
        }
    }
};

Listing G.1: An implementation of the BroadcastReceiver registered with the ConnectHandler.DEVICE_NAMESPACE namespace
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera åktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsmann i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet – or its possible replacement – for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

©Niklas Lavrell