Using the DIAL Protocol for Zero Configuration Connectivity in Cross-Platform Messaging

by

Emil Bergwik

LIU-IDA/LITH-EX-A–14/030–SE

2014-06-18
Final thesis

Using the DIAL Protocol for Zero Configuration Connectivity in Cross-Platform Messaging

by

Emil Bergwik

LIU-IDA/LITH-EX-A–14/030–SE
2014-06-18

Supervisor: Anders Fröberg, Linköping University
Shu Liu, Harbin Institute of Technology
Peter Steiner, Accedo Broadband AB

Examiner: Erik Berglund, Linköping University
Abstract

Today’s living room context offers more and more possibilities when it comes to when and how to interact with the television and media content offerings. Buzzwords such as ”TV Everywhere” is something that both hardware manufacturers, content providers and television networks are pursuing to great lengths. At the core of such marketing schemes is the availability of platform-independent content consumption. In a Utopian setting, the end-user should never have to worry if he or she is currently using a smart TV, tablet, phone or computer to view a video or photos, play music or play games. Taking the concept even further, the devices should also be able to connect and communicate with each other seamlessly. Having for example a television set (first screen) controlled by a mobile phone (second screen) is commonly referred to as companion device interaction and is what this thesis has investigated. More specifically, a way of discovering and launching a first screen application from a second screen application using the zero configuration discovery protocol named DIAL has been implemented into a cross-platform messaging solution. A case study was conducted to gather data about the system and its context as well as what was needed of the framework in terms of architecture design, use cases and implementation details. A proof of concept application was developed for Android that used the proposed framework, showcasing the ease of use and functionality presented in integrating DIAL into such a solution. Since DIAL is so well-documented, easy to understand and is becoming one of the industry standards among consumer electronic manufacturers in terms of device discovery, I believe it should become a standard for so called zero configuration companion device interactivity.
Addendum

This thesis has been performed in parallel and collaboration with another thesis work done by Niklas Lavrell. This has naturally resulted in the two thesis reports sharing some textual- and artifact similarities.
Acknowledgement

First of all I would like to thank all of my coworkers at Accedo Broadband AB, my supervisor Peter Steiner and Fredrik Sandberg and everyone else at the company for all of the feedback, help and opportunities to grow that I have been given during the thesis period. I would also like to extend my thanks to my supervisors and examiners at both Linköping University and Harbin Institute of Technology: Anders Fröberg, Erik Berglund and Shu Liu. Last but not least, I would like to thank my friends and family for all of the support I have gotten during these five years studying, without whom none of this would be possible.

Emil Bergwik
Stockholm, June 2014
# Contents

1 Introduction ........................................ 2
   1.1 Background of Study ............................. 3
   1.2 Purpose & Aim .................................. 3
   1.3 Problem Definition ............................... 4
   1.4 Scope & Limitations .............................. 4
   1.5 Report Disposition ............................... 5

2 Research Method ................................... 6
   2.1 Case Study ...................................... 6
   2.2 The Case Process ................................ 7
       2.2.1 Phase One: Plan & Design .................. 7
       2.2.2 Phase Two: Prepare, Collect & Analyse .... 8
       2.2.3 Phase Three: Report & Conclude .......... 9
       2.2.4 Case Study Validity ....................... 9

3 Literature Review ................................ 10
   3.1 Companion Device Interactivity ................ 10
       3.1.1 Background ................................. 10
       3.1.2 Current Consumer Trends ................... 12
       3.1.3 Controlling, Enriching, Sharing & Transferring ... 12
   3.2 Convergence Technologies ........................ 13
       3.2.1 Digital Living Network Alliance ........... 14
       3.2.2 Apple AirPlay ................................ 15
       3.2.3 Google Chromecast .......................... 16
   3.3 The DIAL Protocol ............................... 16
       3.3.1 Background ................................ 16
       3.3.2 The DIAL Use Case ......................... 17
       3.3.3 DIAL Service Discovery ..................... 18
       3.3.4 DIAL REST Service ........................ 20

4 Case Study ....................................... 24
   4.1 Chosen Integration Model ........................ 24
   4.2 Case Context .................................... 25
       4.2.1 The Connect Technology .................... 25
       4.2.2 The Current Connect Process ............... 27
J  Broadcast Helper Snippet  
K  Roku DemoCast Application  
L  Using the Samsung Multiscreen SDK  
M  Using the Proposed Framework
Acronyms

CE  Consumer Electronics
DIAL  Discovery And Launch
EPG  Electronic Programme Guide
ITV  Interactive Television
PDA  Personal Digital Assistant
QR  Quick Response
SDK  Software Development Kit
STB  Set Top Box
UDA  UPnP Device Architecture
UDP  User Datagram Protocol
URL  Uniform Resource Locator
Chapter 1

Introduction

The definition of the Smart TV ecosystem of the future is, now more than ever, receiving massive attention from major technology players. Not only are these players the TV Consumer Electronic (henceforth CE) manufacturers themselves, but also content providers, smartphone- and other device manufacturers which previously have not been concerned with any kind of living room experience. Often times, a CE manufacturer provide a wide array of products that span across multiple market segments and even product categories. In these cases, the manufacturer will make moves towards building an ecosystem that encapsulates and integrates their own products and making them work together. Technologies such as Apple AirPlay and Samsung Multiscreen are such examples. What is a possible scenario in the long run though, and what is actually happening, is a market fragmentation, where no device of one brand is compatible with a device of a different brand. From a sales and marketing perspective, this could perhaps promote consumer brand loyalty, but the question is if it hinders or promotes technological advancement.

The upside of these technologies though, are that they enable an abundance of new use case, which were impossible when the user was restricted to using the one-way communication of the remote control. Such use cases might be selecting content (such as a movie) on a phone to be played on the TV or showing social media feeds or content meta data on the companion device during playback on the TV.

In this thesis report, the DIAL protocol has been explored as a way of easily enabling such use cases by providing the user with a seamless way of launching and connecting to a Smart TV application via a handheld device. The focus of the thesis has been to prove that it can be integrated as a part of an existing communication protocol but also to facilitate the process of developing new applications using this technology by providing a reusable and generic framework.
1.1 Background of Study

Accedo Broadband AB is a world market leader when it comes to providing media solutions for CE manufacturers, content providers and multimedia networks. They are headquartered in Stockholm, Sweden but have offices in New York, Silicon Valley, Hong Kong, London and many more. Their product portfolio consists of application store solutions, platform-independent messaging solutions, multiscreen multimedia applications and cross-platform development kits. This thesis will focus on the messaging solution, named Accedo Connect, which is a cornerstone in enabling communication between devices, such as the previously mentioned companion device interactivity. The problem with companion device interactivity as it is presented to the user today is that in order to enable this kind of functionality, the involved devices has to be "paired" in one way or another. This process is often cumbersome and can involve navigating both the remote control and the device as well as displaying and entering a PIN code on any device wishing to connect. This naturally lowers the user experience and hinders the acceptance from the general public that any new technology requires in order to become a de facto standard. If the device pairing could happen behind the scenes, and if the Smart TV application could be launched without the use of a remote control, it would pave the way toward a better user experience and wider user adoption of the mentioned functionality.

1.2 Purpose & Aim

The Accedo Connect solution today requires that any set of devices wishing to communicate with each other are connect to- and present within the same communication channel. Joining said channel requires manually inputting a pairing code on the handheld device presented on the Smart TV. The purpose of this thesis is to investigate if the DIAL protocol can be used to eliminate this process as well as eliminating the need for having to use a remote control to start the Smart TV application. The aim of this thesis is to further increase the usability of companion device technologies by streamlining the pairing of a companion device to a Smart TV. In doing this, the aim is also to provide a solution that makes it easy to integrate other technologies similar to DIAL further down the road.
1.3 Problem Definition

Given the purpose and aim defined above, the problem definition for this thesis is as follows.

- *How can the DIAL protocol be used in order to eliminate the need for manual pairing between a companion device and a first screen device?*
- *How can the DIAL protocol be implemented in a way that enables easy integration of arbitrary discovery protocols in the future?*

1.4 Scope & Limitations

The following scope and limitations has been set up for this thesis.

- The proposed framework will use the DIAL protocol version 1.6.5 for discovering and launching a proof-of-concept application from a smartphone to a first screen device. Therefore, any limitations implied by protocol restrictions will have to be considered.
- The proposed framework will use Accedo Connect as a means of communication. Therefore, any limitations implied by the architecture of Accedo Connect will have to be considered.
- The mobile framework and proof-of-concept application will be developed purely for Android and Java.
- The time to develop a proof-of-concept first screen (Smart TV or other receiver device) application will be limited and will therefore be limited to one platform only. The chosen platform will be decided based upon which manufacturer that currently has best support and documentation for DIAL protocol implementation.
- The goal of the implementation is not to get a production-ready framework or application, wherefore it might or not be technically feasible to implement in a live environment. However, an effort should be made to make the framework as scalable and maintainable as possible.
- There is no monetary budget for this thesis, therefore the project will be restricted to free-of-charge tools, frameworks and licenses. Furthermore, the thesis time limit is 20 weeks, of which approximately 10 weeks should be spent developing the framework and proof-of-concept applications for mobile and first screen device.
- The thesis work will be conducted in parallel with another thesis work on integrating the Google Chromecast technology into the Accedo Connect solution. This means that a tight collaboration between these two projects should be maintained and that design considerations and solutions might have to overlap in order for a successful project completion.
1.5 Report Disposition

The thesis report is divided into six chapters: Introduction, Research Method, Theoretical Background, Case Study, Result & Analysis and Discussion.

- **Chapter 1 - Introduction** presents the background of this study, its purpose and aim, the problem definition as well as the scope and limitation and the report disposition.

- **Chapter 2 - Research Method** presents the case study research method I have chosen for this project, including its design, structure and presentation.

- **Chapter 3 - Literature Review** presents the theoretical framework upon which I have based my studies, including any relevant research about companion device interactivity as well as technical specifications needed to complete the project.

- **Chapter 4 - Case study** presents the case study performed as a part of this thesis.

- **Chapter 5 - Result & Analysis** presents the result of the case study and analyses them from two viewpoints; the developer and the end-user.

- **Chapter 6 - Discussion** presents the discussion following the results of this thesis, including what the findings of this thesis implies for the Accedo Connect solution as well as present some outlooks for the future of companion device interactivity that the thesis findings may suggest.
Chapter 2

Research Method

This chapter describes the research method that I have used in my studies.

2.1 Case Study

Using case studies as a tool for exploring a phenomenon within a context has been around in social science and information system studies for many years, while not as wide-spread in software engineering (Runeson and Höst, 2008, p. 2). Runeson and Höst (2008) states that case studies are a good way of studying contemporary phenomenons in which contextual factors are hard to overlook without running a risk of modifying the experiment outcome. Runeson and Höst make a point of the fact that this holds especially true when studying a software engineering activity, the outcome of which very much depends on surrounding factors. Taylor (2013, p. 1) states that, if executed properly, a case study is situated in a real-life context; enables exploration of complex situations and information by relying on multiple data sources and provides enough context description to allow the reader to make judgments about the relevance to its own situation. Robson (2011) categorized case studies into four types, listed below.

- **Exploratory**: Seeks to investigate an area of interest that may or may not be previously known to the researchers or its surrounding.

- **Descriptive**: Describes a situation or context to gain knowledge and materialize a phenomenon that may not yet have been accounted for.

- **Explanatory**: Explains a situation, often drawn from causality, i.e., what happened and why did it happen?

- **Emancipatory**: Seeks to improve the situation as a whole or some part of it.

---

1 Runeson and Höst (2008) label this category as "Improving".
2.2 The Case Process

As with any other formal research project, there is a need for a formal process to support the research within the case study. However, many researchers suggest that the case study method is a flexible way of performing research and that the process is not entirely set in stone, should circumstances change. (Runeson and Höst, 2008; Yin, 2009) As such, steps may be reiterated if its outcome would prove inadequate. Caution should be taken however, to prevent striving too far away from the original research context and objectives (which should result in a new, separate case study). Sangster-Gormley (2013, p. 8) applied the case study process proposed by Yin (2009) and divided their case study into three stages\(^2\), as seen in Figure 2.1. The steps included are described next.

![Figure 2.1: The case study process by Sangster-Gormley (2013).](image)

2.2.1 Phase One: Plan & Design

The case study research begins with planning, in which decisions on what to study, how to study it and what the intent of the study is. Sangster-Gormley (2013, p. 7) suggests that the researcher asks itself: "What is this a case of?" in order to understand what situational context and phenomenon it is really dealing with. If the case or area of study is previously unknown to the researcher, Yin (2009) suggests that a literature review is performed to gain

\(^2\)The process has been slightly modified to include only a single case within the study.
basic knowledge about recent studies. The researcher must then design the actual research execution. This involves deciding whether or not to study multiple cases, and if the study should be holistic or explicitly specify which areas to study within the context (Sangster-Gormley, 2013, p. 8). Lastly, the researcher has to identify what sources of data could be important to studying the case and develop strategies on how to collect that data. This could be in the form of interviews, document reviews and/or observations. It is noted by Sangster-Gormley (2013, p. 9), that it is important to also work out how to gain access to the data, and to gain approval of the data collection plan from any concerned authority (e.g. steering committees, supervisors, etc.) within the case study context. A data collection pilot test should also be performed, to verify that the plan holds in the real-life setting and corrections to it should be made if the pilot test fails in some way.

2.2.2 Phase Two: Prepare, Collect & Analyse

When the design of the case study seems to be properly done and presents a viable plan for data collection, the researcher may proceed to the second phase, which is collecting data. As in many other research projects, it is important that the data collected can be validated. The easiest way for this is by so called triangulation, in which findings from one data source is confirmed by evidence from another data source. (Runeson and Höst, 2008, p. 8). Lethbridge et al. (as cited in Runeson and Höst, 2008, p. 14) categorize data collection activities into three categories, namely first degree, second degree and third degree techniques. First degree data collection would be when the researchers comes into direct contact with the actual sources of the data, such as interview subjects or focus groups. Data collection of the second degree is when the data comes directly from its source but the researcher does not come into contact with its research subjects. Lastly, third degree collection activities are those that collect data from already created data artifacts such as documentation and/or databases. These categories are comprised of different, sometimes overlapping collection methods. As previously mentioned, interviews are always a form of first degree collection, as document inspection is always a third degree activity, whereas observations can be conducted either in a first degree manner, or as a second degree collection, through recorded video- or audio data observation. A second dimension of the data collection activity is the nature of its metrics, be it qualitative or quantitative data collection. Runeson and Höst (2008, p. 19) states that the decision on what data to collect could be based on the findings of the Goal Question Metric method, in which goals are decided and research questions derived and refined based on these goals, and lastly the metrics for answering the questions are derived from the questions.
2.2.3 Phase Three: Report & Conclude

The last, but perhaps most important part of a case study is to analyse and report the data collected throughout the research project. Based on what nature of metrics that has been used (quantitative or qualitative) in the collection process, the analysis will naturally differ on certain points. Quantitative analysis can materialize as diagrams, plots, histograms and/or predictive models whereas qualitative analysis more so relies on making conclusions about the case findings, maintaining a line of argument and evidence. Further, the data analysis and conclusions can set out either to be hypothesis generating, confirming or negating. The first-mentioned sets out to find new explanations to a phenomenon, the second seeks to confirm an already formulated hypothesis whereas the third, negative case analysis aims at explaining a phenomenon in an alternative way, perhaps contradicting an existing hypothesis. (Runeson and Höst, 2008, pp. 20-21)

According to Yin (2009), case study reports can be structured in a number of ways, such as linear-analytic reports, in which the case is described in a linear fashion from problem to conclusion(s); as a comparative study in which comparisons are made between cases; or unsequenced to describe a phenomenon extending across studies.

2.2.4 Case Study Validity

The case study and its findings, as any other research finding have a level of validity in terms of result bias, subjectivity and truthfulness. Yin (as cited in Runeson and Höst, 2008, p. 23) presented four aspects of this validity, as seen below.

- **Construct validity**: To what level the researcher has influenced the measures and findings based on subjective opinions and or research intentions.

- **Internal validity**: To what level unspecified or out-of-scope variables affect the phenomenon under investigation.

- **External validity**: To what level the case findings or conclusions can be applied to other contexts.

- **Reliability**: To what level the case study can be duplicated with the same or similar results and how much the researcher has affected the data or conclusions.

There are a number of ways to improve or control the above-mentioned aspects of validity, such as data triangulation, maintaining a structured case study protocol or having research colleagues perform peer reviews. (Runeson and Höst, 2008, p. 24)
Chapter 3

Literature Review

In order to better understand the field of research on companion device interactivity and its current technologies, a literature review was conducted. This chapter introduces the theoretical background that I have based my studies on.

3.1 Companion Device Interactivity

3.1.1 Background

Companion device interactivity (as seen in Figure 3.1) and computer-to-TV interactions in general, although very much a technological buzz word of today, has been researched ever since the 1990’s (Coffey and Stipp, 1997, p. 61). Researchers back then acknowledged that even though TV usage might decline as the Internet and PC usage gained ground with young adults and children, it would never be entirely replaced. Instead, it was suggested that the two mediums would co-exist and the experience they provided could even be used as cross-promotion for the other channel. (Coffey and Stipp, 1997, pp. 64-66) More recent research expands on this and suggest that as the technological infrastructure surrounding us develops, the TV becomes more and more of a communication hub through which an abundance of media experiences are made available (Hess et al., 2011, p. 11). However paradoxical, research also found that as technological breakthroughs was being made in the field, the TV viewing experience was beginning to be less and less device dependent (Bernhaupt et al., 2012, p. 144).

The TV found its way onto the Internet with the Interactive Television (ITV) and Smart TV concepts, and with that, the step toward enabling TV content on other devices was not far away. Users were no longer restricted to watching their favorite TV exactly when it aired, nor did they even need to watch it on the television. Furthermore, second screen devices were not only limited to presenting the media itself, but could also serve to enrich the TV
viewing by connecting people through social networks and showing content meta data related to the media being displayed on the TV (Cesar et al., 2008, pp. 172-174; Bernhardt et al., 2012, p. 144). Other research found that as more and more advanced technology found its way into our homes and living room in particular, the rate at which we used other devices while simultaneously watching television increased dramatically (Bernhardt et al., 2012, p. 144). More and more advanced Electronic Programme Guides (EPGs) were developed to tailor the TV experience to each individual viewer, acknowledging the differences in viewing habits that parameters such as viewer age, gender and personal interests imposed (Bernhardt and Pirker, 2013, p. 2).

As more and more features and technology was integrated into both the TV and the second screen devices, the amount of use cases needed to be supported by the input devices involved in the process grew. For the television, this meant increasingly advanced user interfaces and remote controls were needed. One such interface presented by Tsekleves et al. (2007, p. 203), although in no way the first of its kind, supported simple use cases such as access to ITV services and EPGs through a Personal Digital Assistant (PDA), but offered no way of switching channel or volume levels. It received positive feedback from its test subjects, much due to the fact that it was perceived as easy to use and because it had a responsive and efficient interface. One of the most promising attributes of such a device though, would be its scalability (supporting new features) and adaptability (changing contexts), a feature that is highly limited in standard remote controls given their restricted modularity. Furthermore, having a single point of interaction to the TV is something that been expressed as long-since sought after. Having to learn and control multiple input devices in order to operate a TV, DVD player, Set Top Box (STB) and game console is not user-friendly and does not support the "eyes free" concept. (Bernhardt et al., 2012, p. 145) People, although generally positive toward the notion of using such a device as a remote control, expressed their concern for the costs it incurred on the household both from purchasing it (instead of buying a standard re-
mote control) and from daily operation (such as data traffic and application usage).

### 3.1.2 Current Consumer Trends

In a report by Google Mobile Ads (2012), it was found that as smartphones are being more and more integrated into our daily lives, they effectively become backbone of our daily interaction with media and the consumption thereof. It was also found that over 77 percent of TV viewers used some sort of companion device at the same time as they watched TV on a normal day, with 49 percent using a smartphone specifically as their companion device. One survey participant was quoted saying

> I do find myself being distracted from what I’m watching a lot more, now that I have these devices. I’ll find myself, just out of habit, picking up the touch pad or the phone and deciding to search on the Internet for a little bit.

Another study found that as much as 86 percent of TV viewers perform another media-related activity in parallel to watching TV (Tsekleves et al., 2009, p. 206). The study also found that viewers generally regard mobile and PC usage as something anti-social, while TV viewing is something highly social. Users however, were generally open to the notion of using the smartphone for such things as sharing videos or photos on a TV monitor (Tsekleves et al., 2009, p. 205). Evelien and Paulussen (2012, pp. 197) found that most of its interviewees had its respective companion device(s) close at hand both when consuming media alone as well as when in the company of others, for such occasions when more information was requested about a certain show or when the TV was showing something that was not of interest to the interviewee. In research published by Red Bee Media (2012, p. 1), some reasons for this new user adoption is the increasingly tech-savvy TV content consumers; the improved home infrastructure, in terms of wireless networks, device availability; as well as the fact that the home as a whole is getting more and more technological, with integrated services finding its way onto devices previously perceived as 'dumb'. Other research, suggest that the users adoption within some age groups are also influenced by such factors as increased perceived ease-of-use and social influence from other people (Taylor, 2013, p. 8). Meanwhile, market analysts project as many as 1.8 billion connected TV devices to be distributed globally by 2016, with 570 million homes owning or having access to one (Gallagher et al., 2012, p. 3).

### 3.1.3 Controlling, Enriching, Sharing & Transferring

Central to any companion device interaction pattern is the concept of controlling, enriching, sharing or transferring, either individually or in combination. These concepts were first coined by Cesar et al. (2008, p. 172)
and describe the essence of what basic use cases a companion device implementation most often aims at fulfilling. Controlling refers to the user being able to control media playback, much like a standard remote control. Enriching and sharing, in synergy with another works to customize the media consumption based on who you are, and what preferences you have. This is done by overlaying content or otherwise modifying the media playback with such things as social media augmentation or content meta data such as reviews, trailers or related content. Lastly, transferring media refers to the possibility of viewing media content on any device as contexts and conditions change. In terms of popularity and user acceptance, this is probably the part of today's companion device technology that has come the furthest with buzzwords such as video on demand and TV anywhere. Although the technology trends start to support these new use cases, some research has also found that interactivity is not unanimously accepted and always something positive. Vorderer et al. (2009, p. 361) showed that interactive media consumption in a way that controls the actual content (such as the concept of "viewer as a director" (Chorianopoulos, 2008, p. 560)) is not always something entirely positive, since some viewers might be more inclined to view the TV viewing activity as something naturally passive. Furthermore, the concept of the "hundred-button" remote control, where new interactivity patterns are supported merely by adding more buttons to an existing remote control, provides a cautionary suggestion that increasing the amount of interaction possibilities that a device enables might not always be something positive. Meanwhile, mobile phones and tablets are still perceived as simple and easy to use, even though they can possibly contain hundreds of applications and enabled use cases. (Gritton, 2013, pp. 41-43) In essence however, the user experience involved in these types of interactions are affected by three aspects, namely; the users internal state, such as needs and previous experiences; the system design and characteristics, such as ease of use, interaction flows, etc; as well as the context and surroundings, i.e. if it is happening in a for example a living room or formal business meeting. This affects how the user perceives the interactivity and use cases presented by the system and therefore also how the user perceives the product it is using. (Hassenzahl and Tractinsky, 2006, p. 95).

3.2 Convergence Technologies

In order to support these new use cases, many industry giants has put in a considerable effort to align themselves with these types of technologies as well as gain as large of a customer base compared to their competitors as possible. Some reasons for this new effort are listed as the affordability, availability and ubiquity of devices that support these new technologies Doughty et al. (2012, pp. 80-81). Building an entirely new ecosystem like this, making mobile devices work together with television sets, STBs and game consoles, has often required the manufacturers to limit the support to merely extend
to their own brand. The reasons for this might stem from marketing schemes
designed to cross-promote the company’s products while at the same time
strengthening the company’s own technological ecosystem. However, most
often the reasons come from technological fragmentation such as the wide
variety of hardware and software embedded in the involved devices used
for graphics rendering, connectivity capabilities and other factors (Dawson,
2013). These factors makes it hard for the manufacturers of such devices to
decide on what type of cross-brand features to support. In this way, these
convergence technologies result in a market fragmentation, where devices
from one manufacturer may not work with devices from another manufac-
turer. The fact remains though, that the technologies themselves provide
great opportunities for interesting use cases and that they in essence con-
tribute to the general public usage of companion devices in a TV context
(Google Mobile Ads, 2012, p. 25). Jolly and Evans (2013, pp. 1-2) name
some of the functionality that the underlying technologies should support
are personalised remote controls and multiscreen experiences such as screen
sharing and social media aggregations. The following section describe some
of the most common standards for device interoperability, with a brief ex-
planation of their underlying protocols and of what practical implications
they have in terms of using them.

3.2.1 Digital Living Network Alliance

The Digital Living Network Alliance, or DLNA, is a trade organization con-
sisting of over 250 member companies, such as Microsoft, Samsung and
Intel. They share a common vision of a connected, digital home with de-
vice interoperability to be achieved through setting cross-industry design
guidelines and protocol usage. The standards covers a multitude of infras-
tructure and product levels, such as physical media, transportation, digital
rights management and protocols for streaming. (Digital Living Network
Alliance, 2014a)

3.2.1.1 Underlying protocols

DLNA device certification requires that a number of protocols concerned
with different layers of the IP stack are supported. The ones that are of
interest in this thesis are the ones responsible for device discovery and me-
dia management. DLNA uses the UPnP Device Control Protocol (DCP)
Framework for discovering DLNA certified devices on a wireless network.
Devices are discovered through the use of the UPnP Simple Service Discov-
ery Protocol (SSDP) and UPnP Audio/Video (AV) specification for media
control such as playback control and volume commands. (Digital Living
Network Alliance, 2014b)
3.2.1.2 Practical Implications

The DLNA specification makes it possible to access local media stored on for example a network-accessible hard drive or computer from a companion device and control if the playback of chosen media should happen on for example a TV screen or if it should be played on the local machine. Since media storage, control and transfer is local, it is generally safe to say that playback and control is instantaneous and that it will remain under user control during each stage of user manipulation, e.g. not being transferred from one device to another via the Internet. The specification however, has its limitations. (Hess et al., 2012, p. 45) Hess et al. mentions some of them as not being able to access content from the Internet and not being able to share user and content access configurations across devices.

3.2.2 Apple AirPlay

Apple AirPlay is a protocol stack developed by Apple Inc. as a way to stream content such as audio, video or photos from one device to another. Licensed by Apple Inc. as a third-party technology for other consumer electronic manufacturers to implement in other product lines, it is now implemented in a number of devices and software suites, such as third-party streaming services, speakers and docking stations (Grobart, 2010).

3.2.2.1 Underlying protocols

Very limited documentation exists on what protocols are actually used behind the scenes when it comes to AirPlay technology, and none that is officially condoned by Apple Inc. Reverse engineering has however revealed that it uses the Apple zero-configuration protocol suite named Bonjour. Bonjour uses Multicast DNS (mDNS) as defined in RFC 6762 (Cheshire and Krochmal, 2013; Apple, Inc., 2014b) for performing name resolutions and discovering services and devices on a wireless network. With later firmware and certain hardware configuration, device discovery is also enabled via Bluetooth technology (Lee, 2014). Subsequent media control is done via the Remote Audio Output Protocol (RAOP) and AirPlay service, encrypted via an AES encrypted TCP connection. (Aruba AirGroup, 2014; Apple, Inc., 2014a; Cheshire and Krochmal, 2013)

3.2.2.2 Practical Implications

Apple AirPlay is capable of both screen mirroring and video and audio playback, from local as well as remote sources such as YouTube or Netflix. It is however, limited to Apples iDevices, such as iPhones, iPads or Mac. (Wikimedia Foundation, Inc., 2014)
3.2.3 Google Chromecast

The Google Chromecast is in essence not a convergence technology, but more of a wireless display technology. It does however, work with Android, Windows, Chrome OS, iOS and Mac so it indeed supports cross-platform usage. It requires a cast-enabled device such as a Google Chromecast, a local wireless network as well as an Internet connection. The Chromecast differs from its competitors in the way it handles streaming. Instead of streaming content from the device it is being controlled by, it fetches the content from the original source, such as a hard drive, home theater PC or streaming service such as YouTube or Netflix. Thus it eliminates the need for the companion device to process and stream the media to the receiving device. (Rowlands, 2013)

3.2.3.1 Underlying protocols

The Google Chromecast uses the Cast SDK to enable inter-device communication, wherefore not much information has been released on what protocols are used for communication. What is known though, is that in early development it used the DIAL Protocol (later described in Section 3.3) with the UPnP Simple Service Discovery Protocol (SSDP) but that it is now using multicast DNS (mDNS) to do device discovery (Dutta and Nicholls, 2014). An early attempt at reverse engineering the technology was made by Nicholls (2013) where it was discovered that websockets and a proprietary protocol called Remote Access Media Protocol (RAMP) was being used by applications, but that technology has since been replaced with other, yet unspecified technology (Nicholls, 2014).

3.2.3.2 Practical Implications

The Google Chromecast is currently fairly limited in terms of how much processing power it has. Therefore, screen mirroring capabilities are limited, while video and audio playback is the primary use case. This means that possibility of having full-blown first screen applications with highly dynamic user interfaces is limited, yet theoretically possible. It provides no security, and therefore relies fully on the local wireless network it is connected to, to have security features enabled. (Ochs, 2013)

3.3 The DIAL Protocol

3.3.1 Background

The DIAL (Discovery And Launch) protocol was launched in early 2013 in a joint attempt by YouTube and Netflix, supported by Sony and Samsung, to ease the process of discovering devices and launching an application once a found device has been selected (Roettgers, 2013). As the name suggest, the
3.3. THE DIAL PROTOCOL  

The DIAL protocol is used only for discovering devices and launching applications on a discovered device remotely. Leaving the actual in-application communication out of the protocol specification was a conscious choice in the effort to create the protocol, Scott Mirer, Director of Product Management at Netflix said in an interview (Roettgers, 2013).

Once apps from the same provider are running on both screens, there are several feasible methods for implementing control protocols either through the cloud or on the local network. And not every service or application is focused on the same kinds of use cases. Rather than try to get universal agreement on these protocols and use cases, it seemed best to leave room for innovation.

With this vision in mind, the protocol was constructed to consist of two basic building blocks, DIAL Service Discovery and DIAL REST Service, described in Section 3.3.3 and Section 3.3.4 respectively.

3.3.2 The DIAL Use Case

The DIAL protocol, as mentioned before, is used as a way for companion devices, such as a smartphone or tablet to discover first screen devices, such as a television, STB or Blu-ray player on a wireless network and subsequently launch an application on a selected device. In Table 3.1, a comparison is presented, showcasing the minimal amount of steps required to control a first screen device using a companion device using DIAL compared to if DIAL was not used. As seen, the amount of steps involved in the process of connecting a companion device to a first screen device is essentially halved, with steps 1, 3, 4 and 5 eliminated if using DIAL. This is what is sometimes referred to as ”zero configuration”. In practice, this means that a user can, for example, start its YouTube application on a smartphone or tablet and ”DIAL” a video clip onto the big screen. Since communication subsequent to the actual discovery and launch can be performed by any proprietary protocol, through any medium such as web sockets or a cloud-based solution, there really is no limit as to what the use cases can be once an application has been launched using DIAL. (Netflix, 2012)
3.3. THE DIAL PROTOCOL  

3.3.3 DIAL Service Discovery

The first component of the Discovery And Launch (DIAL) protocol is the DIAL Service Discovery (SD). DIAL SD adheres to the UPnP Simple Service Discovery Protocol (SSDP) and involves two actors, the DIAL Client (which is a companion device) and the DIAL server (which is the UPnP server running on the first screen device). It involves two possible request/response pairs, as described below and seen in Figure 3.2 on page 19.

1. Searching for Devices (Client) A client discovers DIAL enabled devices within a wireless network by sending a User Datagram Protocol (UDP) multicast packet to multicast address 239.255.255.250 port 1900, defined as an M-SEARCH request. According to Section 1.2.2 of the UPnP Device Architecture (UDA), the UDP packet is required to have the fields specified in Listing 3.1 (UPnP Forum, 2008). A client, or control point, should send this packet more than once, preferably periodically to safeguard against the unreliability of the UDP protocol (UPnP Forum, 2008, p. 19).

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: ssdp:discover
ST: urn:dial-multiscreen-org:service:dial:1
MX: <Number of seconds to randomly wait between 0 and MX>
```

Listing 3.1: M-SEARCH UDP Packet.

<table>
<thead>
<tr>
<th>With DIAL</th>
<th>Without DIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Launch mobile application.</td>
<td>1. Launch TV application using remote control.</td>
</tr>
<tr>
<td>2. Select media to be played.</td>
<td>2. Launch mobile application.</td>
</tr>
<tr>
<td>3. Select a device to play media on.</td>
<td>3. Go to pairing screen on TV</td>
</tr>
<tr>
<td>4. Press play.</td>
<td>4. Go to pairing screen on mobile.</td>
</tr>
<tr>
<td></td>
<td>5. Input the pairing mechanism code (QR, PIN, etc) shown on TV on mobile.</td>
</tr>
<tr>
<td></td>
<td>6. Select media to be played.</td>
</tr>
<tr>
<td></td>
<td>7. Press play.</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of launch process with and without using DIAL.
The important part here is the ST (Search Target) header, specified as `domain-name:service:service type:version` which indicates that the client is requesting responses from DIAL enabled devices. (Netflix, 2012, p. 4)

2. Responding as a Device (Server) A DIAL enabled device that receives a UDP packet as defined above will respond to the requesting IP address. The response will follow the specification defined in Section 1.2.3 of the UDA (UPnP Forum, 2008), seen in Listing 3.2. All header fields are required except for DATE, which is recommended. The most important header in the response is the LOCATION header, which is the client-accessible IP address pointing to the so called device description file. This file will be the target to the subsequent HTTP GET request from the client.

```
HTTP/1.1 200 OK
CACHE-CONTROL: max-age = <Number of seconds until advertisement expires>
DATE: <Date when response was generated>
EXT: <Empty>
LOCATION: <URL for UPnP description for root device>
SERVER: <OS and OS version, UDA specification supported & UPnP product and version>
ST: urn:dial-multiscreen-org:service:dial:1
USN: <Advertisement UUID>
```

Listing 3.2: M-SEARCH Response.
3. Getting Application Information (Client) In order for the client to get more information about a DIAL device it is interested in, it has to retrieve the device description of the so-called root device of the UPnP server running on the DIAL device. This is done by issuing an HTTP GET request for the device description XML file from the LOCATION header Uniform Resource Locator (URL) received in the prior response, for example as seen in Listing 3.3.

```
GET /<Application URL>/dd.xml HTTP/1.1
HOST: <IP address of REST server>:<PORT of REST server>
CACHE-CONTROL: no-cache
```

Listing 3.3: HTTP GET for device description XML file.

4. Responding with Device Description (Server) When a DIAL server receives an HTTP GET request for a valid device description location, it will respond with said information, plus an additional header field specifying the Application-URL, which is the absolute path from which applications that are DIAL enabled can be fetched from. The device description content as defined in Section 2.1 of the UDA can be seen in Appendix A. The part essential to a DIAL client is however, for most use cases, the above mentioned Application-URL header. This header, which when appended with the application name, such as ”YouTube” (e.g. http://192.168.1.137/dial/YouTube) will be the resource representation (referred to as the Application Resource URL) in the DIAL REST service, described in Section 3.3.4.

3.3.4 DIAL REST Service

The second component of the DIAL protocol is the DIAL REST Service (RS) which is responsible for managing application launch, stop and querying. In this interaction, the UPnP server from the DIAL SD (described in Section 3.3.3) is no longer active, but instead a RESTful service running on the server handles incoming requests. It involves three possible request/response pairs, which are described below and can be seen in Figure 3.3 on page 21.

1. Requesting Application Information (Client) A client wishing to know more about a specific application can issue an HTTP GET to the Application Resource URL, created from step 4 of the DIAL SD interaction described above. This HTTP GET can be seen in Listing 3.4.

```
GET /<Application URL>/<Application Name> HTTP/1.1
HOST: <IP address of REST server>:<PORT of REST server>
CACHE-CONTROL: no-cache
```

Listing 3.4: HTTP GET for application information.
Figure 3.3: DIAL REST Service interactions.

2. Responding with Application Information (Server)  Given that the HTTP GET request received was valid and the application exists on the platform (or is installable), the DIAL RS will respond with an XML file containing application information, as seen in Listing 3.5.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<service xmlns="urn:dial-multiscreen-org:schemas:dial">
  <name>Application Name</name>
  <options allowStop="true || false"/>
  <state>running || stopped || installable="Installation URL"
  </state>
  <link rel="run" href="Application Instance resource name"/>
</service>
```

Listing 3.5: Application information XML file.

The options element shown indicates whether or not the application allows users to issue stop requests to running instances of the application, essentially controlled by the DIAL RS. The state element indicates if the application is starting/running, if it is stopped or optionally if it is available for installation. In that case, the string value directly subsequent to install-
3.3. THE DIAL PROTOCOL  CHAPTER 3. LITERATURE REVIEW

`label=` will resolve to a direct link to the installable file of the application (for example an application store). The link element is optional and its `href` value will be the last part of the Application Instance URL of a running application. This is needed if the application supports client requests for stopping a running instance of the application.

3. Requesting Application Launch (Client)  Launching an application is performed by issuing an HTTP POST to the Application Resource URL, as seen in Listing 3.6. If the client wishes to submit arguments to the application to parse on launch, the Content-Type header has to be text/plain and its encoding has to be UTF-8. Parameters may be passed in any format, such as JSON, XML or simple key value pairs as seen in the example below.

```
POST /< Application URL>/*< Application Name>/ HTTP/1.1
Host: <IP address of REST server>:<PORT of REST server>
Cache-Control: no-cache
Content-Type: text/plain; charset="utf-8"
parameterKey=parameterValue
```

Listing 3.6: HTTP POST for application launch.

4. Responding to Launch Request (Server)  The RESTful service running on the DIAL server will respond based on whether or not the application name resolves to a valid application, if the length of the HTTP POST message body is accepted by the device and what state the requested application is currently in. If the application is successfully launched by the DIAL RS, the server will respond with response code 201 CREATED with a LOCATION header containing the absolute path of the running application, the Application Instance URL mentioned earlier. The application should never assume that any security checks has been performed on the launch parameters, wherefore such actions needs to be handled by the receiving application. However, the DIAL protocol strictly specifies that the DIAL parameters must be ensured not to circumvent the fundamental platform OS (Operative System) security checks, such as controlling process ownership, priority or cross-application communication.

5. Requesting Application Stop (Client)  Using the Application Instance URL received in the HTTP POST response from step 4 described above, the client may issue an HTTP DELETE request, as seen in Listing 3.7.

```
DELETE /<Application URL>/Application Name>/run HTTP/1.1
Host: 192.168.1.137:8060
Cache-Control: no-cache
```

Listing 3.7: HTTP DELETE for application stop.
In this example, the suffix "run" after Application Name is the href value of the link element received from the Application information XML file in Listing 3.5.

6. **Responding to Stop Request (Server)** Upon receiving an HTTP DELETE request for a valid and running Application Instance URL, and given that the DIAL RS supports the DELETE operation, the DIAL server will try to stop the application execution and respond with response code 200 OK. If the operation is not supported however, 501 NOT IMPLEMENTED will be sent, and if the application was not running or is not found it will respond with 404 NOT FOUND.
Chapter 4

Case Study

The following chapter describes the exploratory case study performed in this thesis work, including an introduction about the case context and background as well as a description of the framework development process and DIAL integration.

4.1 Chosen Integration Model

In order to successfully develop the framework and integrate the DIAL technology into said framework, an iterative interpretation of the waterfall model was used, as presented in Figure 4.1 on page 25. The process was started by gathering contextual knowledge in the form of a literature review presented in Chapter 3 above, including information about companion device interactivity in general and the DIAL protocol. The second part of this stage was gathering knowledge about the specific case of Accedo and the Accedo Connect technology. This was iterated and validated in meetings with developers working with Accedo Connect in particular to make sure that a correct situational interpretation had been made. The next stage of the development process was to gather requirements both from developers potentially using the framework in the future as well as the end-users using an application integrating the framework and DIAL use case in particular. Based on these requirements, a conceptual- and refined architecture was developed. Both the formulation of requirements and architecture proposals was also iterated in meetings mainly with software architects from Accedo, but also with their Scrum Master. After the artifacts from these stages was approved, the coding of the actual framework and protocol integration began. Finally, demonstration applications was developed to validate the result and showcase its functionality and potential flaws. The following sections describes these stages more in detail.
4.2 Case Context

The product under investigation in this case study was Accedo Connect, a product of Accedo Broadband AB, of which an general overview can be seen in Figure 4.2. The product powers a wide variety of customer solutions around the world and enables inter-device communication through a white label cloud-based messaging system. Through the use of Accedo Connect, content providers and Consumer Electronics (CE) manufacturers alike can enable new, innovative use cases for their customers using different devices in their media consumptions, such as companion device video playback control, content meta-data displays and social TV connections.

4.2.1 The Connect Technology

As previously stated, Connect is a cloud-based messaging system, through which devices running different operative systems can communicate with each other without the developer having to concern itself with what sort of devices will send and listen for messages within an interaction session.
4.2. CASE CONTEXT

This is enabled by maintaining a distributed messaging service, which can provide so called channels to which a device can publish- (send), and subscribe (listen) to messages and events. These channels are securely accessed via pairing codes, either in the form of Quick Response (QR) - or PIN (numeric) codes. The high-level architecture of Accedo Connect is presented in Figure 4.3 and as seen it consists of three components, namely the device application, the messaging server and the Accedo Connect service.

Firstly, the device application is the application running on a television set, a STB, or a companion device that is powered by Accedo Connect, which is referred to as a Connect client. Secondly, the messaging server, is a server cluster that the Connect solution uses to provide the inter-device and -service communication. The cluster is distributed worldwide for stability, scalability and performance. Lastly, the Connect service handles peripheral functionality, such as session persistent pairings, library injection and a host/client structure described below.
4.2.1.1 Persistent Pairing

By utilizing the Accedo Connect service, a communication channel can persist between communication sessions. This means that any set of devices that has previously paired and communicated with each other automatically gets connected to each other once going online. Thus, the persistent pairing technology eliminates the need for using pairing codes each time that the user starts the application on the first- and second screen devices. Joining another channel once paired requires the device to unpair from the previous channel. All of the necessary device information is stored securely within the Accedo Connect service.

4.2.1.2 Library Injection

Once connected and authenticated against the Connect service, a device gets the most recent client library (i.e. code) injected into its application. This library injection enables the latest changes in the messaging system to propagate throughout the network of customers on demand, meaning that code changes only need to be deployed in one place for devices to retrieve when they need it. This library injection is only applicable to JavaScript libraries and is therefore something that mostly concerns first screen (namely TV) applications running web applications.

4.2.1.3 Host/Client Channels

The Connect architecture is founded on the fact that a Connect channel has two different actors, hosts and clients, in a one-to-many relationship. The host, most often being a television set or STB, can control what devices may access the channel by being the only one capable of requesting new pairing codes and kicking other devices. The host device is also the only device able to create a new channel within the cloud platform. The client on the other hand, most often being a touch pad or smart phone, has little to no control over the channel itself, but is mostly concerned with listening to what is going on in the channel as well as sending messages to it.

4.2.2 The Current Connect Process

In order to enable communication between a set of devices through Accedo Connect (see Figure 4.4), a so called host device has to create a channel, if there is none from a previous persistent pairing (described in Section 4.2). The host device is, in most cases a television set, STB or game console\(^1\), which creates a channel once the user navigates to the pairing screen on the host device with the respective remote control. This could potentially require handling multiple remote controls (such as switching input source

\(^1\)Other devices, such as mobile phones may also act as host devices.
4.3 Development Process

This section describes the development process from use case design, requirement gathering, to software design, implementation and validation.

4.3.1 The Proposed Framework

As described in previous sections of this report, an idea came to mind about making the pairing process happen "behind the scenes" and using DIAL to eliminate the need for remote control interactions in starting the first screen application. Looking at how the Google Chromecast worked with users simply "casting" media from the second screen to the first screen, it became...
apparent that this could prove a huge leap in minimizing the amount of steps required to enable the companion device use cases described in Section 4.2. The vision was to enable the companion device application to connect to the Connect servers, request pairing parameters, find DIAL enabled devices on a local, wireless network and once the user chooses a device to connect to, pass the pairing parameters to the selected device in order for it to connect to the same communication channel. Once the two devices are paired, subsequent communication goes through the messaging servers of Connect using some proprietary protocol. Making this scenario a reality could then potentially enable the three scenarios depicted in Appendix B.

4.3.1.1 Designing the Use Cases

First, a set of use cases for the proposed framework was created as seen in the use case diagram in Figure 4.5. The application developer using the framework to develop an application should be able to discover devices on the current wireless network; find more information about a found device; select a device; and start communicating with it. Once a device had been selected, the developer should be able to call methods for querying application status; launch the application on the selected device; send commands to the device; disconnect from it; and stop a running instance of the application. Looking back at what the DIAL protocol enables in terms of use cases, it can be seen that the integration thereof would be concerned with device discovery as well as application management such as launching, querying and stopping the application. The device communication itself, not being a part of the DIAL specification, would have to be handled by Accedo Connect.

4.3.1.2 Requirements Gathering

In order for the proposed framework to work properly within the context of Accedo Connect, a number of requirements needed to be considered, presented in Table 4.1. For the sake of saving space, not all requirements are atomic, meaning that they might cover more than one functionality. Firstly, the framework must use the service provided by Accedo Connect for pairing and communication purposes (Req. No. 1, 2). This means it must be able to initiate against the Connect service; connect to an already existing channel or create a new one; as well as request pairing parameters to pass to a first screen device. Secondly, the framework must use the DIAL protocol as it was described in Section 3.3 to find DIAL servers on a network and launch an application on a device (Req. No. 3, 4). The framework should also be able to query application information and stop a running instance of an application on a DIAL server (Req. No 5, 6). Since the framework will be used in a proof-of-concept application, it must provide an API to execute the pairing process (Req. No. 1.1); find DIAL servers on a network (Req. No. 3.1); launch an application on a first screen device (Req. No. 4.1); and communicate with that device (Req. No 2.1). The
<table>
<thead>
<tr>
<th>Req. No.</th>
<th>Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The framework must use the Connect SDK for pairing.</td>
</tr>
<tr>
<td>1.1</td>
<td>The framework must provide an API for pairing against the Connect service.</td>
</tr>
<tr>
<td>2</td>
<td>The framework must use the Connect SDK for communication.</td>
</tr>
<tr>
<td>2.1</td>
<td>The framework must provide an API for communication with the messaging service.</td>
</tr>
<tr>
<td>3</td>
<td>The framework must use the DIAL protocol to search for DIAL devices.</td>
</tr>
<tr>
<td>3.1</td>
<td>The framework must provide an API for searching for DIAL devices.</td>
</tr>
<tr>
<td>4</td>
<td>The framework must use the DIAL protocol to launch an application on a DIAL device.</td>
</tr>
<tr>
<td>4.1</td>
<td>The framework must provide an API for launching an application on a DIAL device.</td>
</tr>
<tr>
<td>5</td>
<td>The framework should use the DIAL protocol to query a DIAL device for application information.</td>
</tr>
<tr>
<td>5.1</td>
<td>The framework should provide an API for querying a DIAL device for application information.</td>
</tr>
<tr>
<td>6</td>
<td>The framework should use the DIAL protocol to stop an application on a DIAL device.</td>
</tr>
<tr>
<td>6.1</td>
<td>The framework should provide an API for stopping an application on a DIAL device.</td>
</tr>
<tr>
<td>7</td>
<td>The framework should be designed in a way that makes it possible to integrate Google Cast technology into the framework without any major modifications to existing framework code or architecture.</td>
</tr>
<tr>
<td>8</td>
<td>The framework should be designed in a way that abstracts what underlying discovery and communication technology is used.</td>
</tr>
<tr>
<td>9</td>
<td>The companion device application must use the API provided in Req. No. 1.1, 2.1, 3.1 &amp; 4.1 to pair, communicate, find DIAL devices and launch an application respectively.</td>
</tr>
<tr>
<td>10</td>
<td>The companion device application must use the API provided in Req. No. 4.1 to pass parameters to the first screen application.</td>
</tr>
<tr>
<td>11</td>
<td>The companion device application should use the API provided in Req No. 5.1 and 6.1 to query for application information and stop an application respectively.</td>
</tr>
<tr>
<td>12</td>
<td>The first screen application should use the pairing parameters passed from companion device application in connecting to the Connect service.</td>
</tr>
<tr>
<td>13</td>
<td>The first screen application should use the Connect SDK for pairing with pairing parameters from Req. No. 11.</td>
</tr>
<tr>
<td>14</td>
<td>The first screen application should use the Connect SDK for communication with the messaging service.</td>
</tr>
</tbody>
</table>

Table 4.1: Framework and application requirements.
framework should also be built in such a way that the Connect technology can co-exist with the Google Cast technology (Req. No. 7). It should be built in such a way that the underlying technology used is abstracted from a developer using the framework, meaning that the process of discovering-, selecting- and communicating with a Google Cast device should be identical to the same process with a DIAL- & Connect-powered first screen device (Req. No. 8). The functionality provided by the proposed framework will be showcased with a companion device proof-of-concept application. This application must use the API provided by the framework for pairing to a channel; communicating with the first screen device; find DIAL servers on a network; launch an application on a selected device; and pass pairing parameters to a selected device (Req. No. 9, 10). The application should also be able to use the framework API to query for application information from a DIAL device as well as stopping a running instance of the application on the selected device (Req. No. 11). This implies that no technology-specific code should have to be implemented by the developer itself. The project will also require a first screen proof-of-concept application to be
4.3. DEVELOPMENT PROCESS

4.3.1.3 Creating a Conceptual Architecture

Based on the requirements shown in Table 4.1, a high-level conceptual architecture was created through a number of iterations, with the final version seen in Figure 4.6 on page 32. This conceptual architecture shows that two abstractions were made, namely device discovery and device communication. This choice was made both for the sake of separating concerns but also for enabling easier integration of new technologies in the future that may or may not be concerned with both components. As seen, the DIAL and manual pairing component is only concerned with discovering devices and subsequent communication is handled through calls to the Accedo Connect SDK. Meanwhile, the Google Cast SDK component spans both components, as the technology integrates both components.

4.3.1.4 Refining the Architecture

Building onto the conceptual architecture, and specifically the two abstractions (device & discovery), the refined architecture seen in Figure 4.7 on page 33 was created. Four major components were defined, namely handler,
manager-, discovery- and device component(s). The handler is responsible for handling the communication between the framework and developer. Meanwhile, the Cast- and Connect managers is responsible for communicating with the Cast- and Connect SDK respectively. The DIAL manager is responsible for communicating with a selected DIAL device (such as launching, stopping and querying application information). The discovery component was comprised of four sub-components, an interface or abstract device discovery and one discovery component for each mechanism planned to be supported by the framework (Cast, DIAL and manual). The device component, much like the discovery component, would contain an abstract device sub-component and one inheriting (or implementing) device sub-component for Cast and Connect respectively. Since the proposed framework would be generic in the way it handled message- and command passing between different device types, two sub-components was created representing commands that devices can send as well as states that the device can be in. Lastly, since the framework would be dealing a lot with the HTTP protocol as well as local application communication via local broadcasts, two utility (helper) components was proposed to alleviate the handling of these activities.
4.3. DEVELOPMENT PROCESS

4.3.1.5 Target Platforms

As specified in Section 1.4, the framework and proof-of-concept companion device application was developed purely in Java. Because of technical limitations in terms of package functionality, the minimum Android SDK version was set to 11 (Android 3.0 Honeycomb), while the target SDK was set to 19 (Android 4.4 KitKat). The devices used for application testing was an LG Nexus 4 and LG Nexus 5. A market survey of first screen device manufacturers and their level of DIAL support was conducted to find out what platform to target for the development of a proof-of-concept first screen device application. Because of thesis report scope limitations, the survey was limited to four CE manufacturers: Samsung, LG, Panasonic and Roku. The response from Samsung was negative, as seen in the response from a Samsung certification support representative (priv. comm., 2014).

I am sorry to inform this, but I got confirmation from HQ that DIAL documentation cannot be opened to public, so we cannot provide you with the information you need.

The investigation into Panasonic and LG support ended similarly, as seen in the two quotes below from LG and Panasonic contact persons at Accedo (priv. comm., 2014).

This is not supported by Panasonic then if the information is not on the developer portal. Even if it works, from an earlier experience, they will reject the app[lication] even if it works but uses unsupported protocols, frameworks etc. Panasonic is exceptionally strict about this.

[...] officially LG has not included information for DIAL in their SDK (or upcoming SDK) and is giving similar answer as Samsung.

Lastly, DIAL support for Roku hardware was investigated. Fortunately, DIAL support was official and readily available within the Roku External Control Guide within its SDK (Roku, Inc., 2014). The results of the survey made the choice easy and the Roku Streaming Stick was chosen as the target platform for first screen device application development. It should be noted however, that all of the above-mentioned manufacturers actually has the DIAL protocol integrated into many of their newest product lines. Their responses seen above could just suggest that they do not have official support for 3rd party development at the time of writing this thesis report.

4.3.2 Developing the Framework

Developing the proposed framework started with defining the most high level classes and interfaces of the refined architecture in Figure 4.7 in order to find general attributes and shared functionality across sub-components.
This section will describe the Connect Handler, SDK manager, the DIAL device discovery as well as the Connect device implementation. The implementation of Cast technology support is outside of the scope of this report and therefore neither the cast discovery-, device-, nor -manager components will be explained in the following section.

4.3.2.1 Handler Component

The most high level part of the proposed framework, the Connect Handler provides a point of communication with the underlying components and holds references to the different discovery mechanisms, as well as their respective managers and devices. It handles manager instantiation as well as the starting and stopping of discovery mechanisms. Since this object will be the coordinator of events and communication, there can only be one instance of it, wherefore it is a singleton class.

4.3.2.2 Manager Components

**Connect Manager**  In order to communicate with the Connect API, the proposed framework was required to communicate with-, and handle two classes and implement two interfaces by the already defined Connect architecture, as described below (see Figure 4.8 on page 36).

- **Connect (Class):** One of two main interaction points for the Connect SDK, handling state management, channel objects and pairing commands within a given channel, such as kicking, connecting and listing currently paired devices.
- **ConnectCallback (Interface):** The interface which to implement for receiving responses from API calls from the Connect object.
- **Channel (Class):** The second main interaction point for the Connect SDK, which handles the sending of messages within a channel.
- **ChannelCallback (Interface):** The interface which to implement for handling incoming messages, events (such as join and leave) as well as errors occurring within a given channel.

The Connect Manager therefore holds instances of both the Connect class and the Channel class and makes method calls on them, which in turns communicates with the Connect back-end (API). The back-end knows about the interface methods of their respective callbacks and responds through them once command processing is completed. The implementations of each callback interface then broadcasts these responses through the local broadcast helper class to the application implementing the framework as seen in Listing J.1 in Appendix J. This component was therefore partly (together with the Connect Device component described below) responsible for Req. No. 1, 1.1, 2 and 2.1 as defined in Table 4.1.
**DIAL Manager**  Communicating via the DIAL protocol with a selected device’s RESTful service (as described in Section 3.3.4) is handled by the DIAL Manager. This class contains methods for starting and stopping an application as well as querying a selected device for its application state. It uses the HTTP Helper utility class described later to issue its HTTP GET, POST and DELETE operations in a thread separate from main program execution. Furthermore, it uses the `XmlPullParser` from the Android SDK to parse the XML formatted application information file seen in Listing 3.5 on page 21 in order to determine application state. Listing C.1 in Appendix C shows how the DIAL Manager class issues HTTP requests for starting, stopping and querying information for a chosen Connect Device. As seen, it first determines the application state by issuing the first HTTP GET request defined in Section 3.3.4. Based on this result it either launches the application, if the application is not running; prompts the user for installation, if it is marked as installable; or does nothing if the application is already running. Stopping an application is identical, differing only on what HTTP request operation it executes. This component was therefore responsible for handling Req. No. 4, 4.1, 5, 5.1, 6 and 6.1 seen in Table 4.1.
4.3.2.3 Discovery Components

The discovery component contains an abstract super class Device Discovery as well as two sub classes, DIAL Device Discovery and Cast Device Discovery.

**Abstract Device Discovery** The device discovery abstract class merely contains two abstract methods for starting and stopping scanning for devices (see Listing 4.1). Both the class and methods are abstract because the superclass itself should not be possible to instantiate and to require any sub-class to implement the methods it defines. This component covered Req. No. 7 and 8 in Table 4.1.

```java
public abstract class DeviceDiscovery {
    protected DeviceDiscovery() {}  
    public abstract void startActiveScan();
    public abstract void stopActiveScan();
}
```

Listing 4.1: Device discovery abstract class.

**DIAL Device Discovery** The DIAL Device Discovery handles discovering of DIAL enabled devices on a network, namely the Service Discovery part of the DIAL protocol (as described in Section 3.3.3). This component covered Req. No. 3 and 3.1 defined in Table 4.1. Since the DIAL device discovery would need to send and receive HTTP traffic, it needed to be offloaded from the main program execution into its own threads. Furthermore, following the UPnP discovery architecture specified in UPnP Forum (2008, p. 19), broadcast packets needed to be sent multiple times while simultaneously being able to receive incoming network broadcast responses. Therefore, the dial device discovery component was implemented as seen in Listing D.1 in Appendix D. The UML diagram of this component can be seen in Figure E.1 in Appendix E. As seen in this UML diagram, the component contains a Loader class, which handles the multi-threaded sending and receiving of packets. This Loader thread is the work horse of the DIAL process handling and the code for this can be seen in Listing F.1 in Appendix F\(^2\). Once the dial device discovery loader thread was started, it would automatically start sending broadcast packets over UDP as defined in Listing 3.1 on page 18. This message was constructed by sending the

\(^2\)Inspiration for this implementation is attributed to Leon Nicholls, whose GitHub repository can be found at https://github.com/entertailion/DIAL.
M-SEARCH query string over a UDP socket to the broadcast address 239.255.255.0:1900 periodically. Incoming responses were handled in a separate thread where important parts were extracted from the response by parsing the string content from the response packets, saving the location IP address and determining if it was in fact a DIAL server that was found by looking at the SEARCH TARGET header.

**DIAL Server & DIAL Server Broadcasts** If a response packet from a DIAL enabled device has been received, the loader thread will create a so-called DIAL Server Broadcast object instance to represent the device. This device will be compared to a map of already known DIAL devices (since the device may have already been found in a previous broadcast response). Such devices, along with Cast devices, are filtered out. Cast devices are filtered out since they respond to DIAL broadcasts but is already being handled by the Cast Device Discovery mechanism (not covered by this thesis). Devices that pass the filtering are first queried for additional information by issuing an HTTP GET request to their device description URL, then polymorphed into a DIAL Server, a sub class of Dial Server Broadcast. The newly created object is then passed to the loader thread’s callback interface, triggering the DIAL Device Discovery class’ implementation on the IDialCallback interface’s `onResult()` methods as seen in Listing D.1. The `onResult(DialServer ds)` method will then convert the Dial Server to a Connect Device (described in Section 4.3.2.4 below) and broadcast it within the application scope using the broadcast helper (described in Section 4.3.2.6). Once the Loader thread has reached its maximum broadcast interval, it will call the second `onResult(int status)` method of the IDialCallback interface, to let the Dial Device Discovery class know that it should halt the thread it is running the Loader instance in, thus stopping the DIAL broadcasts.

**DIAL Constants** Also included in the UML diagram in Figure E.1 in Appendix E is a DialConstants class. This class contains helpful string- and integer values that are used by the DialDeviceDiscoveryLoader class. This includes strings such as the header fields, M-SEARCH query, the address and port on which to broadcast UDP packets, broadcast intervals and flags (integers) for the loader class to use to let the callback class know it is finished loading.

**4.3.2.4 Device Components**

The device component contains a Device super class as well as two subclasses: Connect Device and Cast Device.

**Abstract Device** The device communication is generalized into an abstract super class which contained shared device information and methods
such as status information, discovery type, connection methods and messaging commands (see Listing G.1 in Appendix G). Since the technology actually used within each of these methods differs between Cast and Connect platforms, all methods were made abstract. The abstract device, together with the device discovery component described above, covers Req. No. 7 and 8.

**Connect Device** Communicating as a Connect device meant extending the abstract Device class as seen in Listing G.1 and implementing its abstract methods. Communicating with the ConnectManager component shown in Figure 4.7 on page 33 could then be done for example by `ConnectManager.getInstance().init(this)` within the overridden `connect()` method of the abstract Device class. This will trigger the connection process described in Figure 4.4 on page 28. Implementing the `sendCommand(Command)` was done by switching over the command types supported and having private methods handle each command type. A snippet of code from this part can be seen in Listing H.1 in Appendix H.

**4.3.2.5 Command Objects**

The refined framework seen in Figure 4.7 also included a Command object for easy handling of device communication, which was implemented as an abstract class, parts of which can be seen in Listing 4.2 on page 39. This component, together with the abstract classes described above covered Req. No. 7 and 8.

```java
public abstract class Command {
    public enum CommandType{
        PLAY ("play"),
        PAUSE ("pause"),
        STOP ...
        /** OMITTED CODE **/
    }
    CommandType commandType;

    public CommandType getCommandType(){
        return this.commandType;
    }
}
```

Listing 4.2: Abstract Command class.

This abstract class is then extended in for example a `PlayCommand` sub class with attributes that are needed in order for a receiving first screen device application to start media playback, such as where to find the media (url), position to start playback from and whether or not to automatically start playback. Any command to be sent to a device is then passed into the
calling of the `device.sendCommand(Command command)` method in Listing G.1 on page 80. The `device` is the selected device instance, such as a DIAL enabled TV or a Google Chromecast.

### 4.3.2.6 Utility Components

Lastly, in order to further the separation of concern, two helper classes was implemented, namely the HTTP Helper and the Broadcast Helper.

**HTTP Helper** The HTTP Helper class contained everything needed to setup and execute HTTP requests. This included everything from specifying protocol parameters such as HTTP version and character set, to connection parameters such as connection- and socket timeout variables. It was custom-designed to work against DIAL enabled devices, meaning it supported only the GET, POST and DELETE requests described in Listing 3.4 (page 20), Listing 3.6 (page 22) and Listing 3.7 (page 22). Listing I.1 in Appendix I shows a snippet of code from this class, specifically the HTTP POST method for launching a DIAL device with a pairing code for Accedo Connect. As seen, the URL prepended to the application name forms the so called Application Resource URL described in Section 3.3.4. By writing POST parameters in a data stream for the open connection, the parameters are passed to the receiving DIAL device. Methods involving the GET and DELETE operations, which did not require parameters to be passed, used the standard Apache HTTP library to execute its requests.

**Broadcast Helper** The last component of the framework was the Broadcast Helper class. This class existed to help other classes bundle messages for cross-component, application-scoped communication. It was mainly intended for the application layer to listen in on any local broadcasts from this class, but in theory any interested component within the framework could listen in on important events from other parts of the framework given that it knew the so called namespace that the messages of interest was broadcast over. Listing J.1 in Appendix J shows the method used by discovery mechanisms to broadcast that a new device has been found on the network. Listening for these discovery messages was then as simple as registering a broadcast receiver on the `LocalBroadcastManager` from the Android SDK, as seen in Listing J.2 in Appendix J.

### 4.4 Framework Validation

In order to validate that the framework was usable in an application context, a set of demonstration applications was developed for Android and Roku. The Android application used the proposed framework to support the use cases presented in Figure 4.5 on page 31 while the Roku application was a
custom, standalone application developed to showcase the DIAL use cases enabled on the Android devices.

### 4.4.1 The Android Proof of Concept

The Android application that was developed, named DemoCast, was a GUI-based video player application consisting of one device list fragment, for viewing available devices on the network; one video list fragment, for browsing videos from a remote video content server; one video player activity, for local video playback; as well as one remote control activity for controlling playback on a selected first screen device application. The architecture for the application can be seen in Figure 4.9 on page 41. The application was developed as it was intended to be used in the future by an application developer, meaning that the only communication that it had with the framework was with its ConnectHandler instance, as seen in Figure 4.7 on page 33 as well as with the device selected by the user. This covered Req. No. 9, 10 and 11 in Table 4.1. The first screen that the user saw was the device list (Figure 4.10 on page 42), from which it could choose a device to start communicating with. Upon successfully connecting to a chosen device, the application would automatically move to the video list, from which the user could choose a video to start playing remotely as seen in Figure 4.11. Once the user had selected a video, the application would switch to the remote control activity (Figure 4.12), or local player activity if no device had been selected or if the connection failed. Navigation between these fragments and activities could also be controlled by the user via a so called Navigation Drawer fragment that slides in from the left (Figure 4.13).

![Figure 4.9: The architecture of the DemoCast Android application.](image)
4.4. FRAMEWORK VALIDATION

CHAPTER 4. CASE STUDY

4.4.2 The Roku Proof of Concept

As described in Chapter 4, the target platform for first screen application development was the Roku Streaming Stick. Therefore, a so called Roku “channel” (application) was developed using the BrightScript language and the Roku External Control Guide as defined by Roku, Inc. (2014). Also an early beta of the Accedo Connect library for Roku was used for communicating with second screen devices. The application architecture was composed of three parts, the graphical interface, the Connect interface as well as the event handlers (see Figure 4.14 on page 43). This architecture was chosen based on the fact that Roku applications are restricted to one so called “event loop” which handles all incoming events such as network traffic and user interactions. This means that the Connect object was solely responsible for dispatching events to the application. This included both Connect events (Presence-, Success-, Error-, State Change- and Message Callbacks in Figure 4.14) but also other events, such as video playback events and remote control key presses (Event Callback in Figure 4.14). The code for this application can be seen in Listing K.1 in Appendix K. Enabling the DIAL protocol in the first place was done by setting the flag `dial=true` in the general configuration file of the application. Once a user had selected the Roku device from the device list seen in Figure 4.10, the device would start the DemoCast application on the Roku unit, authenticate and connect to the Connect messaging service using the pairing parameters it received and await user instructions. This covered Req. No. 12 and 13 in Table 4.1. Given that the connection to the server was successful, the screen seen in Figure 4.15 was displayed. Once a user chooses a video to start playing, the `PlayCommand` was sent from the companion device, through the Connect messaging service and was received in the Message Callback component (seen in Figure 4.14) of the Roku application. This meant that the last requirement,
Req. No. 14 in Table 4.1 was fulfilled. The application was also capable of resuming a connection in order to resume video playback after a potential connection failure. Furthermore, if a second companion device launches the application and it was already running and paired with another device, it would be stopped and relaunched with the new launch parameters. An experimental function was also partly implemented which enables other device(s) to manually pair to a DIAL launched session, enabling many-to-one playback control. This was however something not shown in the GUI, but controlled from the debug console of the Android and Roku units respectively.
4.4.3 Field Studies

As a part of the case study, two shorter field studies was conducted to investigate the Google Chromecast- and the Samsung Multiscreen SDK technology and their development processes.

4.4.3.1 Google Code Lab

As a part of the EMEA (Europe, Middle East & Africa) Chromecast Code Lab arranged by Google Inc. in London (England) an opportunity was given to sit down with Google employees and talk about the proposed framework, DIAL, Cast technology and companion device interactivity in general. It was clear that Google’s vision for companion devices interacting with television sets primarily should be centered around the companion device and not so much the remote control. The Google developers that attended the venue was interested, but a bit cautious about Accedo Connect coexisting with Chromecast technology. However, they made it clear that the use case that DIAL supports in “beaming” content to a first screen remotely from a companion device is the future of this type of interactivity. One developer expressed a vision that the DIAL icon should become the de facto standard for users wanting to control media playback on a first screen.

4.4.3.2 Samsung Developer Day

During a one day developer event at the Samsung Nordic head office in Kista (Sweden) a group of developers from Samsung presented their new Samsung Multiscreen SDK. Samsung has in the past made efforts similar to the Multiscreen SDK, in the form of their N-Service API (previously named Convergence API) and DIAL API. Since Samsung has decided to support the DIAL protocol in their smart TV offering from 2014 and onwards, they have launched a Multiscreen SDK for developers wishing to easily enable companion device interactivity within their applications. This SDK wraps DIAL support in an API for discovery and uses WebSockets for device communication. The high-level architecture of the SDK can be seen in Figure 4.17.

As seen, the SDK supports both web applications, native as well as hybrid applications (native applications with web components). The SDK is comprised of an SSDP server handling the device discovery via DIAL, an HTTP server handling any outgoing network traffic as well as a WebSocket server which handles device communication. The device class present in both the first- and second screen component enables discovery of Samsung Smart TVs supporting the SDK within a network or via a cloud solution. Lastly, the channel class is what enables the communication over WebSockets between a set of devices. The previous N-Service API limited a communication channel to four clients maximum and messages to a maximum.

\[^{3}\]Although DIAL support can be found in some premium models from earlier years.
size of 4 megabytes. The Samsung Multiscreen SDK however has no such restrictions. The Samsung representatives present during the day made it clear that they would support applications using the DIAL protocol as a standalone protocol in their own frameworks, not forcing developers to using the Samsung Multiscreen SDK. An extract of an application using the Samsung Multiscreen SDK can be seen in Listing L.1 in Appendix L. The code presented showcases how an application goes from searching for devices using the `Device.search(..)` method, to getting the correct application and launching via a call to `device.getApplication(..)` and `application.launch(..)`, respectively. Then, the application gets the communication channel for the application and lastly it communicates with it by calling `device.getChannel(..)` and `channel.send(..)`.

Figure 4.17: The architecture of the Samsung Multiscreen SDK. ©Samsung Electronics Inc.
Chapter 5

Result & Analysis

This chapter presents the result of the case study, in which the case implementation is analysed and related to the theoretical framework. The results are analysed from the viewpoint of the developer, in terms of framework design, features and limitations and ease-of-use in implementation; as well as the end-user, in terms of application usage.

5.1 Developer

This section presents an analysis of the results from a potential developer’s viewpoint, looking into the logical structure of the framework and it’s ease-of-use.

5.1.1 Framework

The high-level framework structure, separating device discovery from communication is a logical choice that has since also been adopted in other companion device technologies, such as the LG Connect SDK (LG Silicon Valley Lab, 2014) and Samsung Multiscreen SDK (Samsung Electronics Co., Ltd., 2014). This supports the notion that the most fundamental abstracting design choice was reasonable. Having separate discovery components for each mechanism that Connect should support makes it easy to integrate new mechanisms in the future and also facilitates disabling any mechanism that a potential customer may not want or need in their product. Similarly, having device classes represent each communication mechanism supported makes it easy to integrate new communication technologies, should one want to. The abstraction also makes the code more maintainable, since the technologies used are separated from one another. Creating a Command object structure also makes it easy for device classes to control what kind of communication is currently supported and which is not. Furthermore, it simplifies things for a developer wanting to send commands, comparing it to requiring the
developer to package a message in a specific way such as JSON- or XML-formatting, which is prone to errors. By using this architecture, the project covers requirements 1-8 defined in Table 4.1 on page 30 meaning that all framework-related requirements was fulfilled.

5.1.2 DIAL Integration

The integration of DIAL into the Connect solution with the proposed architecture in mind is logical and abstracts the technical details from a developer implementing the framework into an application. Every step of the DIAL process described in Section 3.3 from Service Discovery to RESTful communication is handled by framework components, namely the DIAL Device Discovery and DIAL Manager respectively. Both components are fully compliant with the DIAL protocol specification as defined by Netflix (2012) as well as any applicable guidelines defined by the UPnP Device Architecture (UPnP Forum, 2008). The DIAL Manager and -Device Discovery classes together handles everything in terms of parsing the device description- and application information XML files. Furthermore, sufficient error and exception handling has been implemented to gracefully handle network errors and any badly formatted XML files. By only exposing methods for starting and stopping device scanning within the Device Discovery component, and methods for connecting, sending commands and disconnecting within the Device component, the developer does not need to know any of the DIAL protocol specifics. By making the DIAL integration handle most of the necessary threaded execution, developers only needs to put minimal effort into this matter. This could however prove to be a negative aspect of the implementation, if for example an application developer wants to handle the threading in implementing the functionality on devices with hardware- or software limitations. Talking about the DIAL protocol in general, it can be noted that it is very easy to make an application DIAL-enabled. This is so because of the amount of effort that YouTube and Netflix together with Sony and Samsung put in to make it as light-weight as possible. Enabling DIAL discovery and launch on the receiving platforms that were investigated was most often merely a matter of writing a line in the application manifest or configuration file, such as dial=true in the case of Roku. Minimizing the implementing developer’s need of extra configuration enables easy integration and could be a catalyst for the protocol becoming more wide-spread. There also exist a few restrictions stemming from how Connect was originally intended to be used with television sets being hosts and companion devices being clients. Since Connect as of the writing of this report is built on this concept, the DIAL integration presents a few issues in how it puts the companion- and first screen application into either host- or client mode. As with most other first screen devices, it is possible to use a remote control to interact with the Roku unit. This makes it possible to start the application on the unit manually via the remote, which in turn potentially puts it in host
device mode in the Accedo Connect service. This presents some problems with how to handle companion devices wishing to DIAL the same application, themselves also being host devices since devices cannot easily switch from one mode to another, let alone command other units to switch from one mode to another. This problem may be alleviated by modifying how the Accedo Connect back-end works, making channels host-independent, i.e. everyone within the channel being clients and having the same access and administration rights within it. Another problem with integrating the DIAL protocol into Accedo Connect is the fact that the DIAL responses for HTTP requests described above cannot be modified by the application. This means that it is truly a one-way communication, making it impossible for the developer to reply with application-specific information. An example of when this would be good would be when responding to the HTTP POST request for launching the application, when you would want to respond with whether or not the connection to the Accedo Connect service was successful (as well as if the application was successfully launched). This was fixed by making the Android framework listen for so called presence events in the channel from the remote device it tried to launch. If a presence event was received from the same device as it launched, the framework knew that the device successfully connected and could proceed to the next state. This worked in the case of Roku devices, since it advertised the same ID in the serialNumber tag of the device description (seen in Appendix A) as the ID it used in communicating with Accedo Connect. This might not be the case for Samsung or LG devices for example, requiring another technique for solving the problem of knowing when two devices are successfully paired. A suggested solution for this problem could be to have the launched device broadcast the pairing code it received upon launch (or another device specific code) when joining the channel, and in that way letting the companion device know when it successfully connected to the same channel.

### 5.1.3 Context

Using the proposed framework in developing a companion device application is easy and as described above, requires almost no knowledge about what underlying discovery and communication technology is being used. Counting the minimum amount of framework-related lines of code required in integrating the framework into the Android DemoCast application amounts to approximately 15, as seen in Listing M.1 in Appendix M. This showcases the simplicity of using the framework and increases the likelihood that developers will, in the end, use the framework in developing their applications. The proof of concept applications developed for Android and Roku together covered requirements 9-14 defined in Table 4.1, which together with requirement 1-8 covered by the framework and DIAL integration meant that every project requirement is fulfilled.
5.1.4 Multiscreen SDK Comparison
The proposed framework share many similarities with the Samsung Multiscreen SDK (described in Section 4.4.3). As seen in this SDK architecture Figure 4.17 on page 45, the two solutions share the abstraction and separation of device discovery and communication mechanisms. They both support automatic device discovery via DIAL (UPnP SSDP), as well as manual pairing through pairing codes. Therefore, both frameworks are network-independent in the sense that they do not necessarily require a wireless network to enable device discovery and pairing. Another feature shared between the two solutions is the host dependency, meaning that there is a need for a host device in the channel which controls channel accessibility. This also means that only a host device is able to open a new communication channel in both Connect and Samsung Multiscreen SDK. Contrary to the Samsung Multiscreen SDK however, the proposed framework is not limited to only Samsung Smart TVs being the target for first screen applications since both DIAL and Connect are platform independent technologies. Comparing the development aspects of the proposed framework to the Samsung Multiscreen SDK (as seen in Listing M.1 in Appendix M compared to Listing L.1 in Appendix L), it can be seen that the latter frequently requires the developer to implement callback interfaces for the method calls it provides. It also requires the developer to handle the threading issues involved in the asynchronous discovery and communication mechanisms. My personal opinion is that, although giving the developer a higher sense of control, this can also result in code having to be duplicated across each anonymous interface implementation or possibly result in empty callback methods. Furthermore, it will most likely result in highly nested code, possibly making such an implementation of the Samsung Multiscreen SDK hard to understand and maintain. In saying this, the framework proposed in this thesis gives less control to the developer but also potentially requires less code in an implementation.

5.2 End-User
This section presents the implications of the case study from the perspective of a future end-user of an application implementing the framework.

5.2.1 Control, Enrich, Share & Transfer
Referring back to the findings of Cesar et al. (2008) regarding companion device interaction patterns, the implementation of the proposed framework into a second screen application could have impact on two of the mentioned aspects, namely control and transfer. As mentioned in Section 1.1, the problem today with companion device control and -interactivity is that it in many cases requires many tedious steps in order to be enabled. This
refers to the need of using multiple remote controls to navigate through the setup process, as well as entering pairing information on the involved devices. This process hinders the rate of adoption for ordinary users wishing to use these interactivity patterns. Implementing the proposed framework for seamlessly initiating the control of a first screen device through a second screen device could prove to alleviate this issue. Furthermore, since the DIAL protocol supports launch parameters to be passed along with application launch, it fully supports the concept of transferring content from one device to another. This could for example be used to transfer playback from a tablet to a television set once a user enters its living room. Both the control and transfer usage scenarios are therefore something that the proposed framework supports and is showcased by the DemoCast applications. The enrich- and share use cases are not applicable to the use cases presented in Figure 4.5 on page 31 and is therefore not covered in this analysis.

5.2.2 Convergence Technology Comparison

Comparing the DIAL-Connect integration with the other technologies for remotely controlling a first screen application described in Section 3.2, a number of advantages and disadvantages in using the proposed framework are uncovered. As with the Google Chromecast, device discovery through DIAL is still restricted to local wireless network because of the UPnP SSDP protocol specification, but since the framework uses Accedo Connect, a network independent technology, subsequent device communication can use a mobile network as well as a medium. This could be a selling point if the end-user’s wireless network is unreliable or restricted in some way. Since the proposed framework also supports manual pairing through pairing codes, it also is possible to use it entirely independent of network capabilities, something that is not possible with the Chromecast. Looking at the hardware limitations of the Google Chromecast, restricting first screen applications to simpler video players, integrating DIAL with Accedo Connect presents another selling point. Because DIAL is available on a range of devices, many of which have better hardware than the Chromecast, and since Connect is platform independent, there is more room for innovation in developing the first screen applications. This means that there may be interactivity use cases enabled by using Connect with DIAL that is not possible if using Chromecast. With both the DIAL protocol and Accedo Connect being platform independent, the proposed framework also has an advantage over the Apple AirPlay technology, which is restricted to the Apple ecosystem (Wikimedia Foundation, Inc., 2014). However, such a degree of freedom does have its negative sides, such as the fact that new applications has to be developed for every new device type to support while the case of AirPlay and Chromecast only requires applications to be developed for one platform. Lastly, comparing it to DLNA, the DIAL integration in Connect presents some advantages, such as DLNA’s lack of support for remote application
launch that DIAL enables. Furthermore, Connect is not limited to fetching content from the local network but may retrieve the content from anywhere, as showcased in the DemoCast applications where content was fetched from servers hosted by Google. As with the Chromecast, DLNA is built for local networks such as WiFi or LAN and therefore requires the user to have the appropriate network infrastructure at home in order to work.
Chapter 6

Discussion

This section presents a discussion about the results, the validity of the case, as well as some conclusions on how the outcome may or may not affect the future of the Connect solution and similar solutions.

6.1 Framework

It is my belief that the proposed framework presents a viable solution on how the support for device discovery mechanisms such as DIAL can be integrated into the current Connect solution. It presents a potential developer with the minimum amount of required interaction points and abstracts the underlying technology without limiting the functionality it enables. The framework architecture is extensible in the way that it divides device discovery from communication, meaning that future technologies should be easy to integrate. Furthermore, the framework support for Google Chromecast, co-developed with Niklas Lavrell, showcases that the architecture supports the integration of technologies that cover both discovery and communication mechanisms. Implementing the framework into the DemoCast application shows that using DIAL in conjunction with Accedo Connect could prove to be a big leap for companion device interactivity adoption, however the actual impact of it needs further investigation. In general, the framework and its underlying technologies, Accedo Connect, DIAL and Chromecast supports the concept of controlling, enriching, sharing and transferring that was brought up in Chapter 3 as defined by Cesar et al. (2008). Mostly however, this thesis has been concerned with the concept of transfer in particular, a usage area that received an increased interest in recent reports (Google Mobile Ads, 2012).
6.2 DIAL Integration

The DIAL protocol is readily available on a number of platforms and it is an open protocol integrating well-documented and standardized procedures from well-renowned actors such as UPnP, Google and Netflix. Therefore, the framework supports the convergence of a fragmented TV technology market. However, since no implementation of for example the Samsung Multiscreen SDK was tested, nothing can be said about how easy the framework is to use in relation to other available solutions. Furthermore, looking at the response received from Samsung, LG and Philips in the search for a suitable target platform for DIAL integration, some caution is advised in adopting the DIAL protocol, as the official support for the protocol in practice might be limited. This potential problem might however be of less concern in the future, as the protocol is still fairly new and has not yet been widely adopted by the developer community. In fact, at the time of finishing this thesis, one of the manufacturers that was contacted prior to starting development approached me ”behind the scenes” and asked me what content provider I was developing a DIAL enabled application for, suggesting that they are more and more opening their support. At about the same time, a new version of the DIAL protocol was released, bringing new features that directly addressed many of the issues brought up in this report. The major update to the protocol was that two way communication was enabled, giving the application developer the possibility to pass information from a running application instance to devices wishing to DIAL to the same application. With protocol support starting to opening up, and with such a fundamental flaw in the protocol removed, the future for the protocol in this context is looking brighter than ever.

6.3 Context

In developing the framework, some points to ponder was uncovered in terms of where Accedo Connect should position itself in the future. First of all, I believe that the device discovery and -communication abstraction should be integrated into the Accedo Connect architecture. In doing this, I also believe that the DIAL protocol should be implemented as a part of the discovery component. One of the biggest bumps in the road while developing the framework was the way that Connect handled devices being clients and hosts. This presented problems unique to this case, since Connect had previously only been used with the television being host and companion device being client. As mentioned earlier I therefore propose that Connect should be made host/client-independent where all devices are made ”equal” within channels, giving them the same rights to issue new pairing codes, kick devices and such. This could lower the amount of bugs arising from one device being in one state while requiring the rights of another state but could also present other unknown issues in application development.
6.4 Case Validity

The case study performed to understand and develop a proposed framework for integrating DIAL into Accedo Connect has been of an exploratory nature. The research area of remote application launch was uncharted water for both me and Accedo as a company. Therefore, not much could be said about what the outcome would be at the beginning of the thesis work. Firstly, something should be said about the data collection activities involved in this case study. Most of the data collection has come from first degree data collection activities, wherefore the scope and intentions of the data collection has been known to me. This would be contrary to third degree activities, in which the researcher may have a harder time knowing if the data collection is biased by out-of-scope variables or limited in some way. Some data collection coming from artifacts such as SDK documentation and articles were of course of the third degree. Furthermore, this has been a qualitative case study, meaning that no specific attention has been given to quantitative measurements and analysis. In terms of construct validity, it is hard tell whether or not I have influenced the findings, since I have been both the architect, designer and developer of the integration. To minimize the effect on the construct validity, every part of the framework design process was iterated a number of times and approval of all artifacts was requested (and given) from many parts of the company. This process involved approval from developers, architects, product owners and even top management. Since the DIAL integration was for a very specific product developed and maintained by Accedo for which all aspects are known and documented, the internal validity can be considered high. This implicates that the external validity, the level of applicability of case findings in other contexts could be considered moderate, since the integration was developed to work with the Accedo Connect infrastructure with all of its related features and limitations. Also, as specified in Section 1.4, the integration was limited to the Android and Roku platforms, meaning that the conclusions may not apply to other platforms. However, since the DIAL protocol is an open and platform-independent protocol, the case findings and conclusions should still hold some relevance to other case contexts. Lastly, the reliability of the case study should be considered moderate. This is so since there might be other ways to design and implement the protocol into the solution, based on architect- and developer experience as well as amount of time and resources available in the different project stages. However, since the integration strictly follows the DIAL protocol specification, the project should share some characteristics with potential future implementations. This assumption is also supported by the fact that major CE manufacturers such as Samsung and LG has chosen to implement their discovery mechanisms in ways similar to the proposed framework.
6.5 Conclusion

Based on the case findings and discussion presented above, some conclusions can be drawn on what this type of implementation does for companion device interactivity. Integrating discovery protocols such as the DIAL protocol into a cross-platform messaging solution is highly recommended. This recommendation is based on the findings presented by my literature review, showing that more and more users are moving between devices with different capabilities as they consume media and that they want to bring content with them as they do this. Using DIAL for transferring the media in content consumption as proposed in this thesis supports this type of behavior and should become a catalyst for companion device interactivity use cases. The choice of DIAL as a means for device discovery is of course not mandatory, but since it has received wide support in consumer electronics products since its release in 2012, it is highly recommended. Furthermore, seeing how easy it is to enable DIAL on receiving devices such as Roku or Samsung TV’s further strengthens the case for this particular protocol. Other protocols such as Multicast DNS, not covered by this case study, might however also prove to be viable options for this type of zero configuration connectivity.

In general, it can be noted that these types of protocols are probably gaining much of their support thanks to the high popularity for new streaming technologies such as the Google Chromecast. It is highly likely that these types of products will force television-, STB- and game console manufacturers to adopt protocols such as DIAL in order to maintain their market positions, further strengthening the case for the type of implementation proposed in this thesis.

It should be noted however, that the case study presented in this report has only been concerned with exploring the possibility of integrating these new discovery mechanisms from a technical point of view. This being said, nothing can be said with scientific proof about how much these new technologies will actually impact the market and change user behavior and consumption patterns. Looking back at the question formulations:

- How can the DIAL protocol be used in order to eliminate the need for manual pairing between a companion device and a first screen device?
- How can the DIAL protocol be implemented in a way that enables easy integration of arbitrary discovery protocols in the future?

and summarizing the findings of this thesis, a set of recommendations is proposed.

- Integration of discovery mechanisms such as the DIAL protocol for zero configuration connectivity in platform-independent messaging solutions is entirely possible and I personally recommend the option presented in this thesis to be further investigated.
6.6 Future Work

The work presented in this thesis gives rise to a number of potential research topics in the field of companion device interactivity. First of all, I would suggest a comparative study to investigate the level of user acceptance of a companion device application actually incorporating the proposed framework compared to an application implementing standard pairing mechanisms. Also, an investigation could be performed on what other discovery technologies, such as Multicast DNS, may be integrated into Accedo Connect and how well the proposed framework architecture handles these. Lastly, a larger cross-technology investigation should be conducted, comparing current discovery mechanisms such as DIAL, Multicast DNS and UPnP Simple Service Discovery to find out which technologies to incorporate. This study should both be concerned with details on what technical implications the different technologies bring to the table, but also what they in the long run mean for the market fragmentation. The driving question for this study should be whether or not it is the right way to go to make a solution ”horizontal” by supporting as many technologies as possible, or if a focus on specialization on a limited set of technologies is better.
Bibliography


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


Appendix A

Device Description XML File

```xml
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>base URL for all relative URLs</URLBase>
  <device>
    <deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
    <friendlyName>short user-friendly title</friendlyName>
    <manufacturer>manufacturer name</manufacturer>
    <manufacturerURL>URL to manufacturer site</manufacturerURL>
    <modelDescription>long user-friendly title</modelDescription>
    <modelName>model name</modelName>
    <modelNumber>model number</modelNumber>
    <modelURL>URL to model site</modelURL>
    <serialNumber>manufacturer’s serial number</serialNumber>
    <UDN>uuid:UUID</UDN>
    <UPC>Universal Product Code</UPC>
  </device>
  <serviceList>
    <service>
      <serviceType>urn:schemas-upnp-org:service:
        serviceType:v</serviceType>
      <serviceId>urn:upnp-org:serviceId:serviceID</serviceId>
      <SCPDURL>URL to service description</SCPDURL>
    </service>
  </serviceList>
</root>
```
APPENDIX A. DEVICE DESCRIPTION XML FILE

Listing A.1: Device description XML file

```xml
<service>
  <controlURL>URL for control</controlURL>
  <eventSubURL>URL for eventing</eventSubURL>
</service>

Declarations for other services defined by a UPnP Forum working committee (if any) go here
Declarations for other services added by UPnP vendor (if any) go here
</serviceList>
<deviceList>
  Description of embedded devices defined by a UPnP Forum working committee (if any) go here
  Description of embedded devices added by UPnP vendor (if any) go here
</deviceList>
<presentationURL>URL for presentation</presentationURL>
</device>
</root>
```
Appendix B

Proposed Usage Scenarios

Figure B.1: Potentially enabled scenarios with proposed framework.
Appendix C

DIAL Manager Code Snippet

```java
public class DialManager {

    private static final String TAG = DialManager.class.getSimpleName();
    private static DialManager instance = null;
    private static Context mContext;
    private ApplicationState applicationState;

    public static synchronized DialManager initialize(Context context){
        if(instance == null){
            instance = new DialManager(context);
        }
        return instance;
    }

    public static DialManager getInstance(){
        if(instance == null){
            Log.e(TAG, "DialManager instance has not yet been built. Please build an instance");
            throw new InstantiationException();
        }
        return instance;
    }

    private DialManager(Context context){
        mContext = context;
    }

    /**
     * Launches the application specified in the device with the specified pairing code
     * @param device
     * @param pairingCode
     */
    protected void launchApplication(final ConnectDevice device,
            final String pairingCode,
            final boolean isAlreadyPaired){
        Thread launchThread = new Thread(new Runnable() {
            // Thread code
        });
    }
}
```
```java
@Override
public void run() {
    HttpHelper httpHelper = new HttpHelper();

    /* Get application status (running/stopped/etc) */
    HttpResponse response = httpHelper.sendHttpGet(device.getApplicationResourceUrl(), "DemoCast");
    String state = null; // save the application state

    try {
        XmlPullParserFactory xmlPullParserFactory = XmlPullParserFactory.newInstance();
        xmlPullParserFactory.setNamespaceAware(true);
        XmlPullParser xpp = xmlPullParserFactory.newPullParser();
        /* Sets the XML parsers content */
        InputStreamReader isr = new InputStreamReader(response.getEntity().getContent());
        xpp.setInput(new BufferedReader(isr));
        /** OMITTED CODE **/
        isr.close(); // close the input stream we used for reading the dd.xml
    } catch (Exception e) {
        Log.e(TAG, "Something went wrong while parsing the XML");
    }

    /* Depending on application state, we should either launch (app not running), install (app is not installed but client allowed to install), ask client to manually pair (if application already running), or do nothing (if state is something undefined as far as DIAL is concerned */
    if (state == null) {
        Log.e(TAG, "Application not found! Launch will not be performed.");
        return;
    } else {
        Log.d(TAG, "Application state " + state);
        if (state.equals("running")) {
            /* Application is already running, should perhaps relaunch or order the client to do manual pairing or open websocket and query for pairing code used. */
```

65
BroadcastHelper.broadcastDeviceMessage("onConnected", "Launching with DIAL not performed, app already running");

} else if (state.equals("stopped")) {
    // Application is installed but not running, we can safely launch using DIAL
    StringBuffer responseMessage = httpHelper.sendHttpPostAsStream("DemoCast", pairingCode, isAlreadyPaired);
    Log.d(TAG, "HTTP POST Response code was: " + response.getStatusLine());
    Log.d(TAG, "HTTP POST Response code was: " + responseMessage.toString());
} else if (state.startsWith("installable=")) {
    /* Application is not installed, but is available for installation
    User should be sent to TV App Store and begin installation */
    String installationUrl = state.split("installable=")[1]; // get the address
    response = httpHelper.sendHttpGet(installationUrl);
    Log.d(TAG, "HTTP POST Response code was: " + response.getStatusLine());
} else {
    // Any other state than the above is not supported by the DIAL protocol
}
}

launchThread.start();

/**
 * Stops the application on the given device, if the application is running. If the application was already stopped (or in a state other than running), it does nothing.
 */
protected void stopApplication(final ConnectDevice device, final DialResultCallback callback) {
    Thread stopThread = new Thread(new Runnable() {
        @Override
        public void run() {
            HttpHelper httpHelper = new HttpHelper();
            etc...
        }
    });
    stopThread.start();
}
/* Get application status (running/stopped/etc) */
HttpResponse response = httpHelper
    .sendHttpGet(device.
        getApplicationResourceUrl(), "DemoCast");
String state = null; // save the application state
try {
    XmlPullParserFactory xmlPullParserFactory =
        XmlPullParserFactory.newInstance();
    xmlPullParserFactory.setNamespaceAware(true);
    XmlPullParser xpp = xmlPullParserFactory.
        newPullParser();
    /* Sets the XML parsers content */
    InputStreamReader isr = new
        InputStreamReader(response.getEntity().
            getContent());
    xpp.setInput(new BufferedReader(isr));
    /** OMITTED CODE **/
    isr.close(); // close the input stream we
        used for reading the dd.xml
} catch (Exception e) {
    Log.e(TAG, "Something went wrong while
        parsing the XML");
    DialManager.this.applicationState =
        ApplicationState.UNKNOWN;
    callback.onApplicationState(DialManager.this.
        applicationState);
    return;
}

/* Depending on application state, we should
   either launch (app not running),
   install (app is not installed but client
   allowed to install,
   ask client to manually pair (if application
   already running), or
   do nothing (if state is something undefined as
   far as DIAL is concerned */
if (state == null) {
    Log.e(TAG, "Application not found! Stop will
        not be performed.");
    DialManager.this.applicationState =
        ApplicationState.NOT_FOUND;
} else {
    Log.d(TAG, "Application state "+ state);
    if (state.equals("running")) {
        response = httpHelper
            .sendHttpDelete(device.
                getApplicationResourceUrl(),
                "DemoCast");
        if (response.getStatusLine().
            getStatusCode() == 200) {
            // 200 OK */
    /* 200 OK */
}
DialManager.this.applicationState = ApplicationState.STOPPED;
} else if(response.getStatusLine().getStatusCode() == 404){
    /* 404 NOT FOUND */
    DialManager.this.applicationState = ApplicationState.NOT_FOUND;
} else if(response.getStatusLine().getStatusCode() == 501){
    /* 501 NOT IMPLEMENTED */
    DialManager.this.applicationState = ApplicationState.NOT_FOUND;
}
} else if(state.equals("stopped")){
    DialManager.this.applicationState = ApplicationState.STOPPED;
} else if(state.startsWith("installable=")){
    DialManager.this.applicationState = ApplicationState.INSTALLABLE;
} else{
    DialManager.this.applicationState = ApplicationState.UNKNOWN;
}
}
callback.onApplicationState(DialManager.this.applicationState);
});
}
stopThread.start();
}

public void getApplicationState(final ConnectDevice device,
final DialResultCallback callback){
Thread queryThread = new Thread(new Runnable() {
    @Override
    public void run () {
        HttpHelper httpHelper = new HttpHelper ();
        /* Get application status (running/stopped/etc) */
        HttpResponse response = httpHelper
            .sendHttpGet (device.getApplicationResourceUrl(), "DemoCast");
        String state = null; // save the application state
        try{
            XmlPullParserFactory xmlPullParserFactory =
                XmlPullParserFactory.newInstance();
            xmlPullParserFactory.setNamespaceAware(true);
            XmlPullParser xpp = xmlPullParserFactory.
                newPullParser();
            /* Sets the XML parsers content */
            InputStreamReader isr = new InputStreamReader(
                response.getEntity().getContent());
            xpp.setInput(new BufferedReader(isr));
            /** OMITTED CODE **/
ISR.close(); // close the input stream we used for reading the dd.xml
}
} catch (Exception e) {
Log.e(TAG, "Something went wrong while parsing the XML");
DialManager.this.applicationState = ApplicationState.UNKNOWN;
callback.onApplicationState(DialManager.this.applicationState);
return;
}
if(state == null) {
DialManager.this.applicationState = ApplicationState.NOT_FOUND;
} else {
Log.d(TAG, "Application state " + state);
if(state.equals("running")) {
DialManager.this.applicationState = ApplicationState.RUNNING;
} else if(state.equals("stopped")) {
DialManager.this.applicationState = ApplicationState.STOPPED;
} else if(state.startsWith("installable=")) {
DialManager.this.applicationState = ApplicationState.INSTALLABLE;
} else {
// Any other state than the above is not supported by the DIAL protocol
DialManager.this.applicationState = ApplicationState.UNKNOWN;
}
} callback.onApplicationState(DialManager.this.applicationState);
}
queryThread.start();

Listing C.1: DIAL Manager Class.
Appendix D

DIAL Device Discovery Class

```java
/** OMMITTED IMPORTS **/
/**
* Handles discovery of dial devices and sends them back to the
* callback interface
*
public class DialDeviceDiscovery extends DeviceDiscovery
    implements IDialResultCallback {
    private final String TAG = DialDeviceDiscovery.class.getSimpleName();
    private Thread mDialDeviceDiscoveryLoaderThread;
    private DialDeviceDiscoveryLoader mDialDeviceDiscoveryLoader;

    public DialDeviceDiscovery(){
        super();
    }

    @Override
    public void startActiveScan() {
        mDialDeviceDiscoveryLoader = new DialDeviceDiscoveryLoader(this);
        mDialDeviceDiscoveryLoaderThread = new Thread(
            mDialDeviceDiscoveryLoader);
        mDialDeviceDiscoveryLoaderThread.start();
    }

    @Override
    public void stopActiveScan(){
        mDialDeviceDiscoveryLoader.close();
        mDialDeviceDiscoveryLoaderThread.interrupt();
    }

    @Override
    public void onResult(DialServer dialServer) {
        assert dialServer != null;
        ConnectDevice newDevice = dialServer.toConnectDevice();
        BroadcastHelper.broadcastDiscoveryMessage(true,
            newDevice);
    }
```

70
```java
@Override
public void onResult(int status) {
    if (status == DialConstants.STOP_RECEIVING) {
        mDialDeviceDiscoveryLoader.close(); // close socket
        mDialDeviceDiscoveryLoaderThread.interrupt();
    }
}
```

Listing D.1: DIAL device discovery implementation.
Appendix E

DIAL Device Discovery
Component UML Diagram

Figure E.1: The DIAL Device Discovery Component UML diagram.
/** OMITTED IMPORTS **/  

/**  
 * Runnable for broadcasting for DIAL servers and receiving  
 * their responses  
 */  
public class DialDeviceDiscoveryLoader implements Runnable {  
    /* LOG TAG */  
    private static final String TAG = DialDeviceDiscoveryLoader.class.getSimpleName();  

    /* DatagramSocket to send and receive UDP packages to/from */  
    private DatagramSocket mSocket;  

    /* Broadcast interval */  
    private int mBroadcastInterval;  

    /* Map for storing the advertising DIAL servers responding  
     to multicast request */  
    private Map<String, DialServerBroadcast> mDialServerBroadcastsFound;  

    /* Callback handler for results */  
    private IDialResultCallback mCallback;  

    public DialDeviceDiscoveryLoader(IDialResultCallback callback) {  
        mCallback = callback;  
        try{  
            mSocket = new DatagramSocket();  
            mSocket.setBroadcast(true);  
        }catch(SocketException se){  
            Log.e(TAG, "SocketException while creating new DatagramSocket", se);  
        }  
        mBroadcastInterval = DialConstants.BROADCAST_INTERVAL_MIN;  
    }
@Override
public void run() {
  if (mDialServerBroadcastsFound == null) {
      mDialServerBroadcastsFound = new HashMap<String, DialServerBroadcast>();
  }

  byte[] incomingBuffer = new byte[4096]; // Construct the package that will contain response

  Thread sendBroadcastThread = new Thread(new Runnable() {
      @Override
      public void run() {
          Log.d(TAG, "Starting broadcast thread");
          while (true) {
              /* Send the broadcast request */
              try {
                  byte[] outgoingBuffer = DialConstants.M_SEARCH.getBytes();
                  DatagramPacket packet = null;
                  try {
                      packet = new DatagramPacket(
                          outgoingBuffer,
                          outgoingBuffer.length,
                          InetAddress.getByName(DialConstants.MULTICAST_IP),
                          DialConstants.MULTICAST_PORT);
                  } catch (UnknownHostException e) {
                      Log.e(TAG, "Address lookup failed", e);
                  }
                  mSocket.send(packet);
                  Log.d(TAG, "Sent broadcast packet");
              } catch (IOException ioe) {
                  Log.d(TAG, "DatagramSocket closed by broadcast thread.");
                  return;
              }
              /* Sleep the thread */
              try {
                  Thread.sleep(mBroadcastInterval);
              } catch (InterruptedException ie) {
              }
              /* Double the wait interval */
              mBroadcastInterval = mBroadcastInterval * 2;
          }
      }
  });
  sendBroadcastThread.start();
}
while (mBroadcastInterval < DialConstants.BROADCAST_INTERVAL_MAX)
{
    Log.d(TAG, "Starting receiving thread");
    try{
        /* Try to open a new socket and handle what has come in */
        DatagramPacket packet = new DatagramPacket(
            incomingBuffer, incomingBuffer.length);
        mSocket.receive(packet);
        parseResponse(packet);
    } catch (InterruptedIOException ioe) {
        Log.d(TAG, "InterruptedIO", ioe);
    } catch (Exception e){
        Log.d(TAG, "Other exception", e);
        break;
    }
}
/* Interrupt broadcast thread and tell activity to stop loader */
sendBroadcastThread.interrupt();
mCallback.onResult(DialConstants.STOP_RECEIVING);

/**
 * Handles a response {DatagramPacket}, parses its contents
 * and if the packet contained DIAL payload and was not malformed, adds
 * @param packet the packet to handle and parse
 */
private void parseResponse(DatagramPacket packet){
    String packetContent = new String(packet.getData(), 0, packet.getLength());
    String[] packetContentArray = packetContent.trim().split("\r");
    for(String content: packetContentArray){
        if(content.startsWith(DialConstants.HEADER_DD_ADDRESS)){
            deviceDescriptionAddress = content.substring(10);
        } else if(content.startsWith(DialConstants.HEADER_SEARCH_TARGET)){
            String searchTarget = content.substring(4);
            if(!searchTarget.equals(DialConstants.SEARCH_TARGET_DIAL)){
                Log.w(TAG, "Broadcast received from non-DIAL device");
                return;
            }
        }
    }
    if(deviceDescriptionAddress.equals(""){
        return;
    }
}
Uri uri = Uri.parse(deviceDescriptionAddress); // Extract the IP (Host) and Port of the LOCATION
DialServerBroadcast dialServerBroadcast;
try{
    /* Try to create a new instance of DialServerBroadcast */
    dialServerBroadcast = new DialServerBroadcast(deviceDescriptionAddress);
} catch (UnknownHostException e){
    Log.e(TAG, "Could not resolve IP address from LOCATION header");
    return;
}
/* Add the found advertisement to the list of known advertisers */
if (mDialServerBroadcastsFound != null &&
    !mDialServerBroadcastsFound.containsKey(dialServerBroadcast.getLocation())){
    mDialServerBroadcastsFound.put(dialServerBroadcast.getLocation(), dialServerBroadcast);
    // Create a new instance of a DIAL server
    DialServer dialServerFound = getDialServer(dialServerBroadcast);
    if (null == null) {
        Log.d(TAG,"Either no HttpResponse was received from GET, or Cast device found");
        return;
    }
    // send the dial servers back to the DeviceDiscovery class
    mCallback.onResult(dialServerFound);
}

/**
 * Creates a new DialServer instance from a DialServerBroadcast
 * @param dsb the DialServerBroadcast to construct the DialServer from
 * @return the DialServer instance created or null if the DialServerBroadcast did not
 * respond to the HTTP GET sent to it OR if the DialServerBroadcast is a Google Chromecast
 */
public DialServer getDialServer(DialServerBroadcast dsb){
    HttpResponse response = new HttpHelper().sendHttpGet(dsb.getLocation());
    DialServer dialServer = null;
    // parse the response and get the headers and content to construct DIAL server instance
    if(response != null){
        String applicationsUrl = null;
        Header header = response.getLastHeader(DialConstants.HEADER_APPLICATION_URL);
        // ...
if(header != null){
    applicationsUrl = header.getValue();
}
try{
    XmlPullParserFactory xmlPullParserFactory =
            XmlPullParserFactory.newInstance();
    xmlPullParserFactory.setNamespaceAware(true);
    XmlPullParser parser = xmlPullParserFactory.
            newPullParser();
    /* Sets the XML parsers content */
    InputStreamReader isr = new InputStreamReader(
            response.getEntity().getContent());
    parser.setInput(new BufferedReader(isr));
    parser.require(XmlPullParser.START_TAG, null, "root");
    parser.nextTag();
    while(parser.next() != XmlPullParser.END_TAG){
        if (parser.getEventType() != XmlPullParser.
                START_TAG) {
            // Not at start , skip loop
            continue;
        }
        String tagName = parser.getName();
        Log.i(TAG, " Tag Name : " + tagName);
        if(tagName.equals("device")){
            dialServer = readDeviceDescription(
                    parser, dsb, applicationsUrl);
        }else{
            skipTag(parser);
        }
    }
    isr.close(); // close the input stream we used
    for reading the dd. xml
}catch(Exception e){
    Log.e(TAG, "Something went wrong while parsing
        the XML");
}
return dialServer;
}

private DialServer readDeviceDescription(XmlPullParser
parser,
        DialServerBroadcast dsb,
        String applicationsUrl){
    DialServer dialServer = null;
    String friendlyName = null;
    String uuid = null;
    String manufacturer = null;
    String modelName = null;
    try{
        while(parser.next() != XmlPullParser.END_TAG) { 
            if (parser.getEventType() != XmlPullParser.
                    START_TAG) {
                // Not at start, skip loop
                continue;
            }
APPENDIX F. DIAL DEVICE DISCOVERY LOADER CLASS

```java
String tagName = parser.getName();
if (tagName.equals("friendlyName")){
    friendlyName = readTagText(parser, tagName);
} else if(tagName.equals("serialNumber")){
    uuid = readTagText(parser, tagName);
} else if(tagName.equals("manufacturer")){
    manufacturer = readTagText(parser, tagName);
    if(manufacturer.equals("Google Inc.")){
        // Discarding Cast devices responding to broadcasts
        return null;
    }
}
else if(tagName.equals("modelName")){
    modelName = readTagText(parser, tagName);
}
else{
    skipTag(parser);
}
dialServer = new DialServer(dsb.getLocation(), applicationsUrl, friendlyName, uuid, manufacturer, modelName);
}
```
/**
 * Skips the parsers current tag and all of its child tags
 * @param parser
 */
private void skipTag(XmlPullParser parser) {
  try {
    if (parser.getEventType() != XmlPullParser.START_TAG) {
      return;
    }
    int depth = 1;
    while (depth != 0) {
      int tagType = parser.next();
      if (tagType == XmlPullParser.END_TAG) {
        depth--;
      } else if (tagType == XmlPullParser.START_TAG) {
        depth++;
      }
    }
  } catch (XmlPullParserException xpe) {
    Log.e(TAG, "XmlPullParserException in skipTag()").
  } catch (IOException ioe) {
    Log.e(TAG, "IOException in skipTag()".
  }
}

public void close() {
  if (mSocket != null) {
    mSocket.close();
  }
}
Appendix G

Abstract Device Class

```java
public abstract class Device implements Serializable {

    public enum DeviceStatus {
        CONNECTING (0, "Connecting"),
        CONNECTED (1, "Connected"),
        BUFFERING ...;

    }  

    public enum DiscoveryType {
        DIAL,
        CAST,
        MANUAL;

    }

    public Device(String friendlyName, DiscoveryType discoveryType, String uuid) {
        this.deviceStatus = DeviceStatus.DISCONNECTED;
    }

    public abstract void connect();
    public abstract void reconnect();
    public abstract void disconnect();
    public abstract void sendCommand(Command command);
}
```

Listing G.1: Abstract device class.
Appendix H

Connect Device Code Snippet

```java
/** OMITTED IMPORTS **/
public class ConnectDevice extends Device {
    /* OMITTED CODE */
    @Override
    public void sendCommand(Command command) {
        if (command == null) return;
        switch (command.getCommandType()) {
            case PLAY:
                play(command);
                break;
            case PAUSE:
                /* OMITTED CODE */
            default:
                /* Will happen if neither of the cases above are satisfied */
                Log.d(TAG, String.format("Command type: %s not supported by Device: %s", command.getCommandType(), this.getDiscoveryType()));
                break;
        }
    }

    private void play(Command command) {
        PlayCommand playCommand = (PlayCommand) command;
        MediaInfo selectedMedia = playCommand.getMediaInfo();
        try {
            JSONObject messageObject = new JSONObject();
            messageObject.put("command", playCommand.getCommandType());
            if (selectedMedia != null) {
                MediaMetadata mm = selectedMedia.getMetadata();
                // We have received a play command for new media to be played
                JSONObject payload = new JSONObject();
                payload.put("url", selectedMedia.getContentId());
                payload.put("streamformat", "mp4");
            }
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
```
```java
payload.put("position", 0);
payload.put("autostart", true);
payload.put("title", mm.getString(MediaMetadata.KEY_TITLE));
payload.put("subtitle", mm.getString(MediaMetadata.KEY_SUBTITLE));
payload.put("studio", mm.getString(MediaMetadata.KEY_STUDIO));
payload.put("imageurl", Utils.getImageUrl(selectedMedia, 0));
payload.put("bigimageurl", Utils.getImageUrl(selectedMedia, 1));
messageObject.put("payload", payload);
}
ConnectManager.getInstance().sendMessage(messageObject.toString());
} catch (JSONException jse) {
    Log.e(TAG, "Play: Something went wrong while parsing JSON", jse);
} catch (ConnectException ce) {
    Log.e(TAG, "Could not play: ", ce);
}
```

Listing H.1: Connect Device sendCommand implementation snippet.
Appendix I

HTTP Helper Snippet

```java
public StringBuffer sendHttpPostAsStream(String url, String application, String pairingCode, boolean isAlreadyPaired)
{
    try {
        URL obj;
        if (url.endsWith("/") ^ application.startsWith("/")){
            obj = new URL(url + application);
        } else {
            obj = new URL(url +"/" + application);
        }
        HttpURLConnection con = (HttpURLConnection) obj.
            openConnection();

        // Add request header
        con.setRequestMethod("POST");
        con.setRequestProperty("Content-Type", "text/plain;
            charset=UTF-8");

        // Send post request
        con.setDoOutput(true);
        DataOutputStream wr = new DataOutputStream(con.
            getOutputStream());
        String urlParameters = "pairingCode=" + pairingCode +
            
            
            "&isAlreadyPaired=" + isAlreadyPaired;
        wr.writeBytes(urlParameters);
        wr.flush(); wr.close();

        BufferedReader in = new BufferedReader(new
            InputStreamReader(con.getInputStream()));
        String inputLine;
        StringBuffer response = new StringBuffer();
        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        in.close();
        return response;
    }
    /** OMITTED CODE **/
}
```

Listing I.1: HTTP Helper POST request method.
Appendix J

Broadcast Helper Snippet

```java
private static void broadcastDiscoveryMessage(Device device){
    intent.setAction(ConnectHandler.DISCOVERY_NAMESPACE);
    Bundle bundle = new Bundle();
    bundle.putSerializable("device", device);
    intent.putExtras(bundle);
    LocalBroadcastManager.getInstance(ConnectHandler.getContext()).sendBroadcast(intent);
}
```

Listing J.1: Broadcast helper class device discovery broadcast method.

```java
@Override
protected void onResume() {
    LocalBroadcastManager.getInstance(getApplicationContext()).registerReceiver(mDiscoveryReceiver,
    new IntentFilter(DemoCastApplication.getConnectHandler().DISCOVERY_NAMESPACE));
}
```

Listing J.2: Registering a receiver for device discovery.
Appendix K

Roku DemoCast Application

```vbs
' ********************************************************
* Main function
* arg will be be either:
* 1 (Started from remote):
*  splashTime: X
*  source: homescreen
* OR
* 2 (Started from DIAL):
*  splashTime: X
*  source: dial
*  pairingCode: XXXX
* isAlreadyPaired: true/false

*********************************************************
Sub Main(arg as Dynamic)
  if arg <> invalid then
    print " -------- Argument -------- 
    print arg
    print " -------- Argument type -------- 
    print type(arg)
  endif

  parameters = CreateObject("roAssociativeArray")
  if arg <> invalid and arg.pairingCode <> invalid and arg.
    isAlreadyPaired <> invalid then
    parameters.pairingCode = arg.pairingCode
    if parameters.pairingCode = "true" then
      parameters.pairingCode = true
    else
      parameters.isAlreadyPaired = false
    endif
  else
    parameters.pairingCode = invalid
    parameters.isAlreadyPaired = invalid
  endif

  connectObj = Setup(parameters)
  connectObj.this.setupcanvas(connectObj)
  connectObj.this.eventloop(connectObj)
End Sub
```
Function Setup(parameters as Dynamic) as Object
print "Initializing"
this = {
  'Screen':
    canvas: CreateObject("roImageCanvas")
    player: CreateObject("roVideoPlayer")
  occupancyRect: {x: 50, y: 50, w: 200, h: 100}
  stateTextRect: {x: 300, y: 50, w: 650, h: 100}
  pairingCodeRect: {x: 1000, y: 50, w: 200, h: 100}
  loadingTextRect: {x: 300, y: 520, w: 650, h: 200}
  progressBarRect: {x: 350, y: 500, w: 598, h: 37}
  thumbImageRect: {x: 50, y: 400, w: 100, h: 200}
  homeImageRect: {x: 400, y: 220, w: 460, h: 206}
  'Connect':
    isAlreadyPaired: false
  'Playback':
    progress: 0
    playbackPosition: 0
    playbackState: "idle"
    idleReason: ""
    paused: invalid
    mediainfo: invalid
  urlTransferId: invalid 'Used to identify url event response to thumb image dl request'
  'Functions':
    eventloop: EventLoop
    setupcanvas: SetupCanvas
    paint: PaintCanvas
    playvideo: PlayVideo
}

connectObj = acConstructConnectObject(this.canvas)
this.port = connectObj.port
connectObj.this = this

'Check launch arguments'
if parameters.pairingCode <> invalid then
    connectObj.pairingCode = parameters.pairingCode
endif
connectObj.this.isAlreadyPaired = parameters.isAlreadyPaired

'Setup canvas'
this.canvas.setMessagePort(this.port)
this.canvas.SetLayer(0, { Color: "#000000" })
this.canvas.show()
this.targetRect = this.canvas.getCanvasRect()
Setup video player
this.player.setMessagePort(this.port)
this.player.SetPositionNotificationPeriod(1)
this.player.SetDestinationRect(this.targetRect)

Setup connect object callback methods
connectObj.callbacks.onSuccess = onSuccessCallback
connectObj.callbacks.onMessage = onMessageCallback
connectObj.callbacks.onPresence = onPresenceCallback
connectObj.callbacks.onFail = onFailCallback
connectObj.callbacks.onEvent = onEventCallback
connectObj.callbacks.onStateChange = onStateChangeCallback

return connectObj

End Function

Function EventLoop(connectObj as Object)
' Initiate connect object and start its event loop '
deviceId = CreateObject("roDeviceInfo").GetDeviceUniqueId()
appKey = "3b7afb40fe7ee1a3d5e2ab7ccdb32672"
connectObj.init(appKey, deviceId)
while true
    wait (100, connectObj.port)
end while
End Function

Function SetupCanvas(connectObj as Object)
print "Setting up canvas"
m.canvas.AllowUpdates(false)
m.paint(connectObj)
m.canvas.AllowUpdates(true)
End Function

Function PaintCanvas(connectObj as Object)
print "Painting canvas"
splash = []
list = []

' Paint progress bar '
if m.progress < 100 then
    print "progress ": m.progress
    progress_bar = {TargetRect: m.progressBarRect, url: "pkg:/images/progress_bar.png"}
    color = "#303030"
    splash.Push({
        url: "pkg:/images/usecase1.png"
        TargetRect: m.homeImageRect
    })
if m.progress = 0 then
    ' We havent even started loading '
    progress_bar.url = ""
    list.Push({
        Text: "Waiting for input..."
        TextAttrs: { font: "small", color: "#FFFFFF" }
        TargetRect: m.loadingTextRect
    })

End Function
APPENDIX K. ROKU DEMOCAST APPLICATION

})

print progress_bar.url
else

fileSystem = CreateObject("roFileSystem")
    ' Check if thumb image has been downloaded, if so then display it
    if fileSystem.Exists("tmp:/thumb_image.jpg") then
        thumb_image = {TargetRect: m.thumbImageRect, url: "file://tmp/thumb_image.jpg"}
        list.Push(thumb_image)
    endif
endif

if connectObj.this.paused = invalid then
    list.Push({
        Text: "Loading..." + chr(10) +
                connectObj.this.medainfo.title + chr(10) + connectObj.this.medainfo.studio
        TextAttrs: { font: "small", color: "#FFFFFF" }
        TargetRect: m.loadingTextRect
    })
endif

if m.progress > 0 AND m.progress < 20 then
    progress_bar.url = "pkg:/images/progress_bar_1.png"
    print progress_bar.url
else if m.progress >= 20 AND m.progress < 40 then
    progress_bar.url = "pkg:/images/progress_bar_2.png"
    print progress_bar.url
else if m.progress >= 40 AND m.progress < 75 then
    progress_bar.url = "pkg:/images/progress_bar_3.png"
    print progress_bar.url
else
    progress_bar.url = "pkg:/images/progress_bar_4.png"
    print progress_bar.url
endif

list.Push(progress_bar)
endif

print "paused "; connectObj.this.paused
if connectObj.this.paused = true then
    fileSystem = CreateObject("roFileSystem")
        ' Check if thumb image has been downloaded, if so then display it
    if fileSystem.Exists("tmp:/thumb_image.jpg") then
        thumb_image = {TargetRect: m.thumbImageRect, url: "file://tmp/thumb_image.jpg"}
        list.Push(thumb_image)
    endif
list.Push({

88
Text: `connectObj.this.medainfo.title + chr(10) + connectObj.this.medainfo.studio`
TextAttrs: `{ font: "small", color: "#FFFFFF" }`
TargetRect: `m.loadingTextRect`
```
endif

' Paint pairing code '
if connectObj.pairingCode <> invalid
    list.push({
        Text: "Code: " + connectObj.pairingCode
        TextAttrs: `{ font: "small", color: "#FFFFFF" }`
        TargetRect: `m.pairingCodeRect`
    })
endif

'Clear previous content '
m.canvas.ClearLayer(0)
m.canvas.ClearLayer(1)
m.canvas.ClearLayer(2)
if m.playbackState = "idle" then
    m.canvas.SetLayer(0, { Color: "#303030" })
else
    m.canvas.SetLayer(0, { Color: "#00000000", CompositionMode: "Source" })
endif

' Set new content '
m.canvas.SetLayer(1, splash)
m.canvas.SetLayer(2, list)
' Clear lists for next use '
list.clear()
splash.clear()
```

End Function

Function PlayVideo(connectObj as Object, mediainfo as Object)
    print "Loading Video..."
    ' Reset progress and playback position for new playback
    fileSystem = CreateObject("roFileSystem")
    fileSystem.Delete("tmp/thumb_image.jpg") ' Delete previous thumb image (if any)'
    connectObj.this.progress = 0
    connectObj.this.playbackPosition = 0
    contentList = []
    connectObj.this.medainfo = mediainfo
    content = {
        Stream: { url: mediainfo.url }
        StreamFormat: mediainfo.streamFormat
        Title: mediainfo.title
        Description: mediainfo.subtitle
        SDPosterUrl: mediainfo.imageUrl
        HDPosterUrl: mediainfo.bigImageUrl
    }
    contentList.push(content)
    connectObj.this.player.setContentList(contentList)
    connectObj.this.player.seek(mediainfo.position)
if mediainfo.autostart
    connectObj[this].player.play()
else
    connectObj[this].player.pause()
endif

' Get the thumb image asynchronously '
request = CreateObject("roUrlTransfer")
connectObj[this].urlTransferId = request.getIdentity()
port = connectObj.port
request.SetUrl(mediainfo.imageurl)
print " request url : "; request.getUrl()
request.SetMessagePort(port)
async_req_sent = request.AsyncGetToFile("tmp:/thumb_image.jpg")
print " Request sent : "; async_req_sent
connectObj[this].paint(connectObj)
End Function

Function onSuccessCallback (connectObj)
    print "- - - - - - - - -"
    print " START onSuccessCallback "
    print " State : "; connectObj.currentState
    print " Occupancy : "; connectObj.occupancy
    occupancyText = []
    occupancyText.Push({
        Text: " Occupancy : " + str(connectObj.occupancy)
        TextAttrs: { font: " small ", color: " # FFFFFF " } 
        TargetRect: connectObj[this]. occupancyRect
    })
    connectObj[this].canvas.ClearLayer(4)
    connectObj[this].canvas.SetLayer(4, occupancyText)
    print " END onSuccessCallback "
    print "- - - - - - - - -"
End Function

Function onPresenceCallback (connectObj, msg)
    print "- - - - - - - - -"
    print " START onPresenceCallback "
    print " Message type : "; type(msg)
    print " Message: "; msg
    print " Message content type : "; type(msg.message)
    print " Message content : "; msg.Message
    occupancyText = []
    occupancyText.Push({
        Text: " Occupancy : " + str(connectObj.occupancy)
        TextAttrs: { font: " small ", color: " # FFFFFF " } 
        TargetRect: connectObj[this]. occupancyRect
    })
    connectObj[this].canvas.ClearLayer(4)
    connectObj[this].canvas.SetLayer(4, occupancyText)
    print " END onPresenceCallback "
    print "- - - - - - - - -"
End Function

Function onFailCallback (connectObj, message)
    print "- - - - - - - - -"
    print " START onFailCallback "

Function onMessageCallback (connectObj, msg)
print "START onMessageCallback"
print "Message type: "; type(msg)
print "Message: "; msg
print "Message content type: "; type(msg.message)
print "Message content "; msg.message
if type(msg.message) = "String"
    json = ParseJSON(msg.message)
    print "JSON "; json
    print "JSON payload "; json.payload
else if json.command = "play" and json.payload <> invalid
    connectObj.this.playvideo(connectObj, json.payload)
    connectObj.this.paused = false
else if json.command = "play" and json.payload = invalid and connectObj.this.paused = true
    connectObj.this.player.resume()
    connectObj.this.paused = false
else if json.command = "pause" and connectObj.
    this.paused = false
    connectObj.this.player.pause()
    connectObj.this.paused = true
else if json.command = "seek"
    connectObj.this.playbackPosition = json.
    payload.position
    connectObj.this.player.seek(connectObj.this.
    playbackPosition)
if connectObj.this.paused = true then
    connectObj.this.player.resume()
else if json.command = "request_media_state"
    print "received request for media state
    update"
    message = constructMediaStateMessage( connectObj)
    connectObj.sendMessage(message)
endif
print "END onMessageCallback"
End Function

Function onEventCallback (connectObj, event)
print "START onEventCallback"
print "Event type: "; type(event)
if type(event) = "roVideoPlayerEvent" then
    if event.isStatusMessage() then
        connectObj.this.paused = true
        print "Raw progress: " + stri(event.GetIndex())
        progress% = event.GetIndex() / 10
        if connectObj.this.progress <> progress%
            connectObj.this.progress = progress%
        connectObj.this.paint(connectObj)
    end if
    else if event.GetMessage() = "start of play" then
        ' once the video starts, clear out the canvas so it doesn't cover the video '
        connectObj.this.canvas.SetLayer(0, { color: "#00000000", CompositionMode: "Source" })
        connectObj.this.canvas.show()
        connectObj.this.paused = false
        connectObj.this.playbackState = "playing"
        connectObj.this.idleReason = 
        message = constructMediaStateMessage(connectObj)
        connectObj.sendMessage(message)
    endif
else if event.isPlaybackPosition() then
    ' Update the playback position '
    print "Playback position "; event.getIndex(); " seconds"
    connectObj.this.playbackPosition = event.GetIndex() *1000
else if event.isPaused() then
    connectObj.this.paused = true
    connectObj.this.playbackState = "paused"
    connectObj.this.idleReason = 
    connectObj.this.paint(connectObj)
    message = constructMediaStateMessage(connectObj)
    connectObj.sendMessage(message)
else if event.isResumed() then
    connectObj.this.paused = false
    connectObj.this.playbackState = "playing"
    connectObj.this.idleReason = 
    connectObj.this.paint(connectObj)
    message = constructMediaStateMessage(connectObj)
    connectObj.sendMessage(message)
else if event.isFullResult() then
    connectObj.this.progress = 0
    connectObj.this.mediainfo = invalid ' Reset media info before sending '
    connectObj.this.paused = invalid
    connectObj.this.playbackState = "idle"
    connectObj.this.idleReason = 
    connectObj.this.paint(connectObj)
    message = constructMediaStateMessage(connectObj)
    connectObj.sendMessage(message)
else if event.isRequestFailed() then

print "Video playback failed"
print "Reason: "; event.GetMessage()
    connectObj.this.mediaInfo = invalid ' Reset media info before sending'
    connectObj.this.paused = invalid
    connectObj.this.playbackState = "idle"
    connectObj.this.idleReason = "error"
message = constructMediaStateMessage(connectObj)
    connectObj.sendMessage(message)
endif
else if type(event) = "roImageCanvasEvent" then
    if event.isRemoteKeyPressed() then
        handleKeyPress(connectObj, event.getIndex())
    endif
endif
else if type(event) = "roUrlEvent" then
print "Received url event"
if connectObj.this.urlTransferId <> invalid then
    ' if the urlevent was transfer complete and is the same as we requested'
    if event.getInt() = 1 and event.getSourceIdentity() = connectObj.this.urlTransferId then
        print "Image transfer complete"
        ' We got the response from thumb image request, repaint canvas'
        fs = CreateObject("roFileSystem")
        folders = fs.getDirectoryListing("tmp:\")
        for each folder in folders
            print "Folder path "; folder
        end for
        connectObj.this.paint(connectObj)
    else
        print "Image transfer started"
    endif
endif
endif
print "END onEventCallback"
print "- - - - - - - - -"

End Function

Function onStateChangeCallback (connectObj, state, oldState)
print "- - - - - - - - -" print "START onStateChangeCallback: " print "New state: "; state print "Old state: "; oldState
' Always draw the state on the canvas'
stateText = []
stateText.Push({
    Text: state
    TextAttrs: { font: "small", color: "#FFFFFF" }
    TargetRect: connectObj.this.stateTextRect
})
occupancyText = []
occupancyText.Push({
    Text: "Occupancy: " + str(connectObj.occupancy)
    TextAttrs: { font: "small", color: "#FFFFFF" }
    TargetRect: connectObj.this.occupancyRect
})
connectObj.this.canvas.ClearLayer(3)
correctObj.this.canvas.ClearLayer(4)
correctObj.this.canvas.SetLayer(3, stateText)
correctObj.this.canvas.SetLayer(4, occupancyText)
stateText.clear()
if(state = correctObj.states.INITIALIZED)
  if connectObj.this.isAlreadyPaired = invalid
    ' We started app with remote control '
correctObj.hasPair(hostConnectCallback)
  else if connectObj.this.isAlreadyPaired then
    ' We are already paired to the correct 
    channel, just reconnect '
correctObj.hasPair(
      companionReconnectCallback)
  else
    ' We are not paired with correct channel 
    , so check if paired and if so, 
    disconnect from that '
    ' and connect to correct one with 
    pairing code '
correctObj.hasPair(
      companionConnectCallback)
  endif
endif
print "END onStateChangeCallback"
print "- - - - - - - - -"
End Function

Function companionReconnectCallback (correctObj as Object, 
hasPair as boolean)
  print "companionReconnectCallback"
  if hasPair then
    ' device auth successful, we are connected, do 
    nothing '
    print "Succes: Reconnected"
  else
    ' Something wrong has happened '
    print "Error: Could not reconnect"
  endif
End Function

Function companionConnectCallback (correctObj as Object, hasPair 
as boolean)
  print "companionConnectCallback"
  if hasPair then
    ' We are in companion mode but was connected to 
    wrong channel, first we need to disconnect 
    from old '
    correctObj.leave() 
    correctObj.unpair(unpairCompanionConnectCallback
    )
  else
    ' either we are host in one channel or we are 
    not paired at all '
    correctObj.pairWithCode(correctObj.pairingCode)
  endif
Function hostConnectCallback (connectObj as Object, hasPair as boolean)
    print "hostConnectCallback"
    if hasPair then
        ' We are reconnected to a channel, and dont need
        to do anything ' 
    else
        connectObj.pair()
    endif
End Function

' ********************************************************
* Callback function for companion unpair wishing to pair with 
* new channel.
* Will try to pair with pairing code if successfully unpaired 
* from other channel 
* connectObj: the application connect object 
* isUnpaired: true if successfully unpaired, else false
******************************************************** '
Function unpairCompanionConnectCallback (connectObj as Object, isUnpaired as boolean)
    print "Unpair successful: "; isUnpaired
    if isUnpaired then
        connectObj.pairWithCode(connectObj.pairingCode)
    else
        print "Something went wrong!"
    endif
End Function

Function constructMediaStateMessage (connectObj as Object) as Object
    message = CreateObject("roAssociativeArray")
    message.messageType = "mediastate"
    messageObject = CreateObject("roAssociativeArray")
    messageObject.playbackPosition = connectObj.this.
    playbackPosition
    if connectObj.this.player.getPlaybackDuration() = -1 then
        messageObject.mediaDuration = 0
    else
        messageObject.mediaDuration = connectObj.this.player.
        getPlaybackDuration()*1000
    endif
    messageObject.playbackState = connectObj.this.playbackState
    if connectObj.this.idleReason <> "" then
        messageObject.idleReason = connectObj.this.idleReason
    endif
    if connectObj.this.mediainfo <> invalid then
        messageObject.mediainfo = connectObj.this.mediainfo
    endif
    message.messageObject = messageObject
    print "--- - - - constructMediaStateMessage - - - -"
    print "message: "; message.messageObject
    print "--- - - - - - - - - - - - - - - - - - - - - - - - - - - -"
    return message
End Function
Function handleKeyPress (connectObj as Object, keyIndex as Integer)
    if keyIndex = 2 ' <Up> '    
    else if keyIndex = 3 ' <Down> '    
    else if keyIndex = 4 ' <LEFT> '    
    else if keyIndex = 5 ' <RIGHT> '    
    else if keyIndex = 6 ' OK '    
    else if keyIndex = 8 ' <RW> '    
        print "Rewind button pressed"
        ' Rewind 3 seconds (3000 ms) '    
        connectObj.this.playbackPosition = connectObj.this.
        playbackPosition - 3000    
        connectObj.this.player.seek(connectObj.this.
        playbackPosition)    
    else if keyIndex = 9 ' <FF> '    
        print "Fast-forward button pressed"
        ' Fast-forward 3 seconds (3000 ms) '    
        connectObj.this.playbackPosition = connectObj.this.
        playbackPosition + 3000    
        connectObj.this.player.seek(connectObj.this.
        playbackPosition)    
    else if keyIndex = 13 ' <PLAY/PAUSE> '    
        Print "Play/Pause button pressed"
        if connectObj.this.paused    
            connectObj.this.player.resume()    
        else    
            connectObj.this.player.pause()    
        endif    
    end if
End Function

Listing K.1: Roku DemoCast Proof of concept application.
## Appendix L

### Using the Samsung Multiscreen SDK

```java
public class DemoApplication extends ActionBarActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // Finding devices using DIAL
        Device.search(new DeviceAsyncResult<List<Device>>()) {
            @Override
            public void onResult(final List<Device> devices) {
                DemoApplication.getInstance().runOnUiThread(new Runnable() {
                    public void run() {
                        /* OMITTED CODE */
                    }
                });
            }
            @Override
            public void onError(final DeviceError error) {
                DemoApplication.getInstance().runOnUiThread(new Runnable() {
                    public void run() {
                        /* OMITTED CODE */
                    }
                });
            }
        };
        ...
        // Getting an application on a device
        device.getApplication(runTitle, new DeviceAsyncResult<Application>>() {
            @Override
            public void onResult(Application app) {
                DemoApplication.getInstance().runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        /* OMITTED CODE */
                    }
                });
            }
        };
    }
}
```
```java
@Override
public void onError(DeviceError e) {
    DemoApplication.getInstance().runOnUIThread(new Runnable() {
        @Override
        public void run() {
            /* OMITTED CODE */
        }
    });
}

... // Launching an application
application.launch(parameters, new ApplicationAsyncResult<Boolean>() {
    @Override
    public void onResult(final Boolean result) {
        DemoApplication.getInstance().runOnUIThread(new Runnable() {
            @Override
            public void run() {
                /** OMITTED CODE **/
            }
        });
    }
    @Override
    public void onError(ApplicationError e) {
        DemoApplication.getInstance().runOnUIThread(new Runnable() {
            @Override
            public void run() {
                /** OMITTED CODE **/
            }
        });
    }
});

// Connecting to the communication channel
device.connectToChannel(channelId, clientAttributes, new DeviceAsyncResult<Channel>() {
    @Override
    public void onResult(final Channel channel) {
        DemoApplication.getInstance().runOnUIThread(new Runnable() {
            @Override
            public void run() {
                /** OMITTED CODE */
            }
        });
    }
    ... @Override
    public void onError(final DeviceError error) {
        DemoApplication.getInstance().runOnUIThread(new Runnable() {
            @Override
            public void run() {
                /** OMITTED CODE **/
            }
        });
    }
});
```
APPENDIX L. USING THE SAMSUNG MULTISCREEN SDK

Listing L.1: Demo application using the Samsung Multiscreen SDK.

```java
...
    channel.send("Play", false);
}
```

Listing L.1: Demo application using the Samsung Multiscreen SDK.
Appendix M

Using the Proposed Framework

```java
public class DemoApplication extends ActionBarActivity{
    @Override
    protected void onCreate(Bundle savedInstanceState) { 
    ...
        DemoApplication.initializeConnectHandler(context);
    ...
    }

    @Override
    protected void onResume() {
        LocalBroadcastManager.getInstance(getApplicationContext()).registerReceiver(mDiscoveryReceiver,
        new IntentFilter(CastApplication.getConnectHandler().DISCOVERY_NAMESPACE));
        LocalBroadcastManager.getInstance(getApplicationContext()).registerReceiver(mDeviceReceiver,
        new IntentFilter(CastApplication.getConnectHandler().DEVICE_NAMESPACE));
        DemoApplication.getConnectHandler().startActiveScan();
    ...
    }

    @Override
    protected void onPause() {
        LocalBroadcastManager.getInstance(this).unregisterReceiver(mDiscoveryReceiver);
        LocalBroadcastManager.getInstance(this).unregisterReceiver(mDeviceReceiver);
        DemoApplication.getConnectHandler().stopActiveScan();
    ...
    }

    public void onDeviceSelected(Device device){
    ...
        device.connect(pairingCode);
    ...
    }

    public void onMediaSelected(Media media){
```
... device.sendCommand(playCommand); ...
}

public void onDeviceUnselected(Device device){
    ...
    device.disconnect();
    ...
}

private BroadcastReceiver mDiscoveryReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle b = intent.getExtras();
        Device device = b.getSerializable("device");
        /** OMITTED CODE **/
    }
};

private BroadcastReceiver mDeviceReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.hasExtra("message"){
            /** OMITTED CODE **/
        } else if ...
    }
};

Listing M.1: Approximate amount of code lines required to use framework.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida 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.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman 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/

©Emil Bergwik