Graduation thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in de Toegepaste Informatica

CrossWoW: Towards Rapid Prototyping of Cross-Device Interaction

TOM DE RYCKE

Academic year 2016 - 2017

Promoter: Prof. Dr. Beat Signer
Advisor: Audrey Sanctorum
Faculty of Science and Bio-Engineering Sciences

Afstudeer eindwerk ingediend in gedeeltelijke vervulling van de eisen
voor het behalen van de graad
Master of Science in de Toegepaste Informatica

CrossWoW: Towards Rapid Prototyping of Cross-Device Interaction

TOM DE RYCKE
Academiejaar 2016-2017

Promoter: Prof. Dr. Beat Signer
Advisor: Audrey Sanctorum
Faculteit Wetenschappen en Bio-Ingenieurswetenschappen

Abstract

Over the last decade, we have seen a significant increase in new electronic devices dealing with digital information and services. Just think of the wearable technology with the well-known smart watches and the Google Glass technology.

Professionals, such as a photographer or a designer, own a lot of different electronic devices, like a digital camera, a tablet, an electronic drawing board, a laptop or a big screen, and all these devices try to interact with each other. The management of digital information has become particularly challenging due to the diversity of devices and the lack of effective ways for communication between these devices.

The goal of this thesis is to start searching for effective ways for the development of cross-device interaction. After all, the problem is that interactions across multiple devices are either completely decoupled or predefined in a non-customisable manner. We start by exploring the state of the art of distributed user interfaces and cross-device interaction and identify certain shortcomings. Thereafter, we implement the proof of concept CrossWoW application that supports cross-device interactions with a high granularity and the support for offline interaction.

Finally, we will do some reverse engineering to propose possible guidelines for a cross-device framework. In our proposal, we will try to find a framework that is extensible, flexible and customisable.

Acknowledgements

Writing this dissertation has only been possible thanks to the support of many people. First of all, I would like to thank my advisor, Audrey Sanctorum. I had the pleasure of working with her and she helped me to put together this research in a structured, qualitative way. I would like to sincerely express my gratefulness for her continuous feedback, proofreading and the time that she has spent over the past year to help me writing this thesis. I would also like to thank my promotor Prof. Dr. Beat Signer to promote this thesis and for his enormous enthusiasm during his lectures and his contribution for this thesis.

Last but not least, I also received support from my family and especially of my dear wife Ilse Van de Velde and my three sons: Wout, Vik and Jeff. Their love and support have helped me a lot during the past four years at the VUB to obtain my Master's degree. My wife has to give me the opportunity and the patience to study again. It must not have been easy to live with a student and working husband who had to go to class in the evening and to work for the school during weekends.

Contents

1 Introduction

1.1 Nation of Multi-Devices

Mark Weiser had described already in 1991 a world in which people would be surrounded by devices of different sizes [65]. In public spaces like stations, airports or museums, we can find a lot of big screens to show information for the customers and sometimes we can even communicate with the different devices [23, 10]. At home, we find more and more devices that are connected to the Internet and form part of the so-called Internet of Things (IoT). Fitness trackers, thermostats or door locks are just a few examples of IoT devices that have already become part of our everyday life. Last year, Google has invented the Google Home[1], a voice-activated speaker powered by the Google Assistant. Amazon Echo[2] and Lenovo Smart Assistant[3], which is based on Amazon Alexia, are other voice-activated speaker products like Google Home.

Figure 1.1[4] shows that the number of global users on a mobile device is higher than on a desktop. Mobile first is the new pillar in a lot of companies.

[1]`https://madeby.google.com/home/`

[2]`http://www.amazon.com/oc/echo/`

[3]`http://blog.lenovo.com/en/blog/raising-the-iq-of-todays-smart-home/`

[4]`http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/`

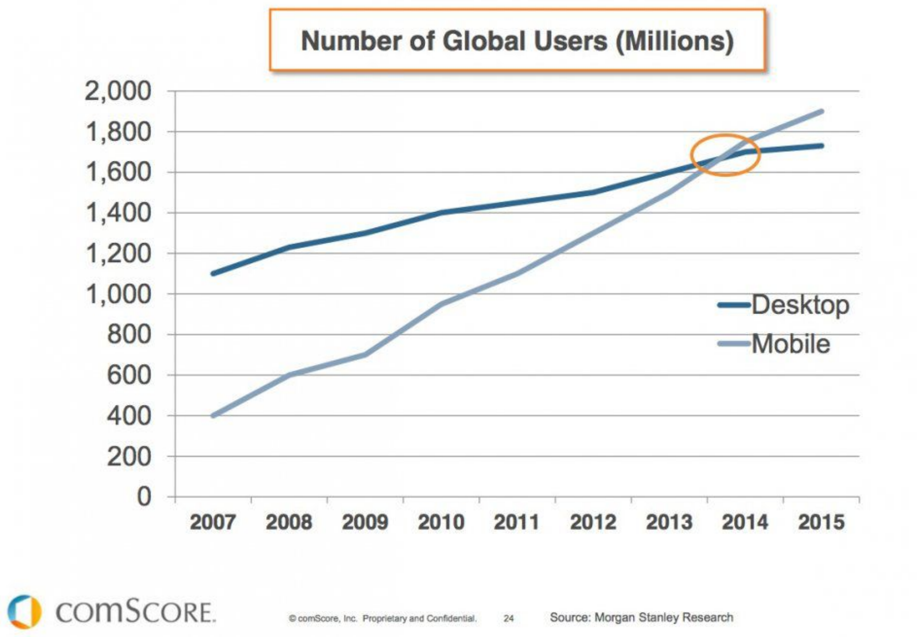


Figure 1.1: Mobile vs. Desktop users

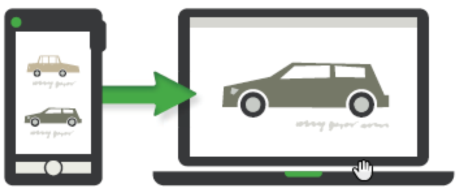
New projects will start first in Android and iOS because the clients increasingly use the mobile applications instead of the web page through a browser.

Mobile devices like smartphones are not only used for making a phone call but they are now also adapted for many other tasks. They are used for making pictures, socialising, searching on the Internet or bank transfers. Hence, they are used a lot for accessing digital media and services on the Internet [45, 55]. Many applications have been moved from the desktop to mobile platforms. Smartphones are used for daily activities and for playing popular games, like Pokémon Go[5]. Smartwatches are used for frequent task activities or as a remote control. Tablets, smart eyewear or laptops are all acting for different purposes. Users have now access to a variety of devices and they will choose the device that best suits their purpose. Some of these devices can also be shared with family and friends [20]. The explosion of all these new devices has led to an ecosystem of displays [62] supporting a wide variety of social setups going from individual use to a many to many social group interactions. Santosa and Wigdor [52] describe that users will select devices on their appearance. Tablets for reading or watching a movie on a

[5]`http://www.pokemongo.com`

train because of their size, weight and portability. Smartphones are more used for quicker, smaller tasks. But, if we want to do more productively work, laptops or desktop computers are the preferred devices.

If we make a SWOT analysis, we can determine the strengths, weaknesses, opportunities and threats for each device. After all, it is important to get the most out of each device. We can then combine the strengths of each device to enable rich and fluid interaction behaviours across multiple devices. Google and Facebook, two important players on the Internet, have already done a lot of research to understand the cross-platform consumer behaviour. Google has conducted research in the US and has published their results in "The New Multi-Screen World" [25]. Their conclusion is that we are now a nation of multi-screeners. They talk about two main modes of multi-screening: sequential and simultaneous screening (Figure 1.2).


(a) Sequential screening

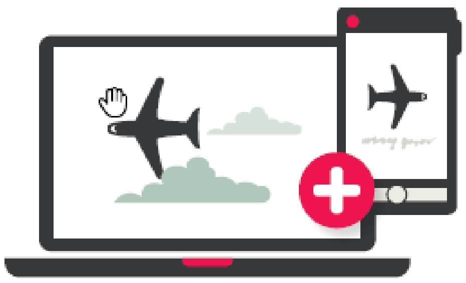

(b) Simultaneous screening

Figure 1.2: Two main modes of multi-screening

What a lot of people do nowadays is to start a search operation on their smartphone. If they want to do something more productive with their search results, they complete their task on another device with a bigger screen and with an easier input like a keyboard. This is an example of sequential screening where we continue our task on another device. We could do this by sending the URL via email from our smartphone to the mailbox on our laptop, where we continue the task by opening that URL in our browser. We are going from one computer to multiple heterogeneous devices per user [15]. Deep Shot [9] is a framework for migrating such tasks across devices by using the mobile phone camera. Deep Shot captures the user's working state that is needed for a task and resumes it on a different device. For example, a user is looking up a direction on their desktop computer in Google Maps. But now, they would like to shift the same map to their phone for navigation in their car. With Deep Shot, users can migrate the same map on their phone by simply taking a picture of it using the mobile camera of the phone. Deep Shot will not only capture the pixels, but also the information behind these pixels, which is the working state of an application. Hence, the captured map remains interactive on the phone. In addition to pulling out the interactive map to the phone, Deep Shot can also push information back to a computer. For example, a user would like to give a review about a restaurant that they have just visited. They have opened the page for writing the review on their phone, but it is pain to type on a mobile phone. So, again they would like to shift the review page to the desktop computer, where they can easily type their review.

It becomes more challenging when we look to the simultaneous screening where we use more than one device at the same time. If it is an unrelated activity, we could speak of multitasking [25]. If we do a related activity, we could then speak of complementary-usage. A lot of research is already done to investigate a variety of behaviours that can be formed by orchestrating a smartphone and a watch [28] or a personal digital assistant and a television [49]. Television companies have begun to support some basic cross-device behaviours, such as navigating media from a phone or tablet to a television. Therefore, we can use some current media sharing protocols, such as Universal Plug and Play (UPnP) or the Digital Living Network Alliance (DLNA)[6] which was founded by a group of consumer electronics companies and is still very popular today. Casting content from one device to a television can be done by Miracast, which is a standard for wireless connections from devices to displays. It is a bit like "HDMI over Wi-Fi", where the cable is replaced by the Wi-Fi connection. Apple Airplay and Samsung AllShare are other techniques for casting content to a television. Therefore, we must connect an extra device like a Chromecast, Apple tv, Google tv or Amazon Fire TV Stick at the target device. At these moments, we use different devices in combination, rather than using each device in isolation. These are some simple examples of distribute our interfaces. We dive deeper into the term Distributed User Interfaces (DUI) and some related work in Chapter 2.

Hence, as Weiser imagined more than two decades ago and as if has been confirmed by GSMA Intelligence[7], we are currently living in a multi-device world, where the number of mobile devices has surpassed the world population.

[6]`https://www.dlna.org/`

[7]`http://gsmaintelligence.com`

1.2 XDI is Still Effort Consuming

The management of all the digital information is a challenge due to the diversity of devices and the lack of effective ways for communication between these devices. Cross-Device Interaction (XDI) continues to be an everyday challenge for users to interact and is therefore still a hot topic. The problem is that interactions across multiple devices are either completely decoupled or predefined in a non-customisable manner [51]. A lot of current solutions have a client-server architecture [58, 66, 26, 9] or are camera based [51] to facilitate the communication between different devices. Often we see that only a part of the data can be shared across devices, for example, only web pages or web elements can be shared. These are then more web-based architectures with a low granularity [51]. In other cases, the data can only be transferred to a fixed set of devices. Finally, most of these interactions cannot be changed or modified by the users. In Chapter 2, we focus on all these current shortcomings.

Developers need to capture the user input, sensing events from multiple devices and have to distribute the different user interfaces to a (pre)defined set of devices. The realisation of all these different tasks requests a lot of effort and is still time consuming. All these device-specific constraints, synchronisation and communication are a technical challenge. It distracts developers from their main development for creating the application logic. And, developers will create an application that is often difficult to maintain and adapt to new types of device combinations. Software maintenance is one of the biggest costs in software development [3, 2]. But if we could, for example, write less code, avoid reworking and avoid complexity that would help to reduce the software maintenance costs significantly [8]. Therefore, we propose an initial possible setup of a framework for rapid prototyping of cross-device interaction (XDI) with a high granularity and support for offline interactions. We expect that we could reduce the effort for the developers and help to improve the maintainability and readability. So, we can reduce the budget of a company. Frameworks are made for complex manufacturing problems [33]. The last decade, a lot of frameworks are already made which aim at decreasing development time and improving interoperability. We expect that with our framework, developers need less programming skills to create, change and understand the logic of the asked programs and will improve their productivity. So, the software development and maintenance will be lower when the applications are created with our framework. Hence, a framework can help to reduce the size and the complexity of the implementation of software [67], and increase the readability and maintainability.

The problem is that there is no framework for hands that can connect different devices based on a logical name for each device and that supports the distribution of media files with a certain form of granularity. Our contributions can be summarised as follows. We will develop a wrapper of an existing library that will make it easier for developers to create cross-device applications with a high distribution granularity. This wrapper will not only support the connection between different devices in any location. It will also have the necessary interface with corresponding messages to distribute a list of media files with some form of granularity and some accompanying actions on these media files. In order to demonstrate our wrapper, we provide a proof-of-concept application called CrossWoW. As last contribution, based on our experience, we will propose some design guidelines to develop a framework for the prototyping of cross-device applications.

1.3 Thesis Outline

We will start in Chapter 2 with a history and some definitions about cross-device interactions. We will mention some related work and identify certain shortcomings and existing approaches. With that information, we propose two proof of concept applications (POCs) in Chapter 3 and the methodology that we used to develop these applications. We will explain the ideal solution of our two POCs. Thereafter, in Chapter 4, we will explain how we have implemented our two POCs, which problems we have encountered and how we have solved these problems. We will try to create a common folder for the two applications. With all this information, we will evaluate our implementation and propose an initial possible setup of a framework in Chapter 5 and explain how it would be easier if we had already a framework that can do the challenging job for the interactions between our different devices. In the last Chapter, we will give a conclusion with some possible future work of our research and mention some design guidelines.

2 State of The Art

2.1 History

2.1.1 One User - One Computer

Throughout the 1980s and 1990s, desktop computers became common and more and more popular. They were intended for the use of a single non technical user. Because of the size and the power requirements, desktop computers were placed at a specific location near a desk or a table. They were certainly not portable. A desktop computer was also quite expensive and was used by several people of the family. A desktop computer still exists today but can have the ability to manage multiple user profiles. In the meanwhile, a lot of other devices like laptops, tablets and smartphones have been added. Most of these devices are usually used by one specific person of the family and so we can speak of a more personal device.

2.1.2 Ubiquitous Computing

In 1991 Mark Weiser [65] had described a world in which people would be surrounded by devices of different sizes. Devices would be integrated into the environment and allow users to easily use computing power for everyday interactions with the world. Mark Weiser referred his vision of modern com-

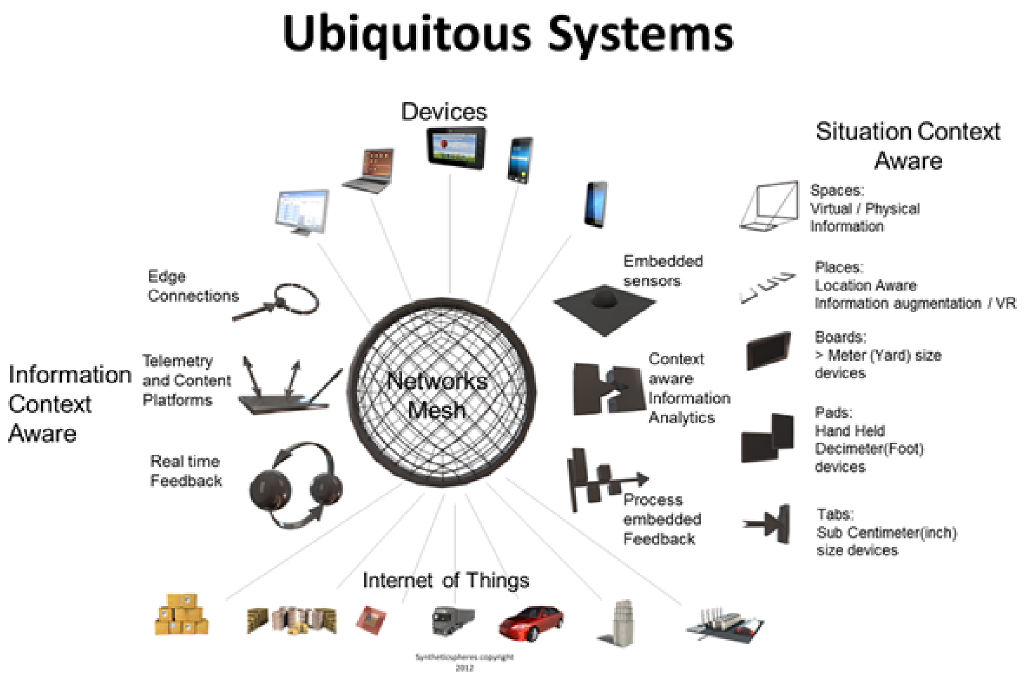

Ubiquitous Systems
Devices
Situation Context Aware
Spaces: Virtual / Physical Information
Places: Location Aware Information augmentation / VR
Boards: > Meter (Yard) size devices
Pads: Hand Held Decimeter(Foot) devices
Tabs: Sub Centimeter(inch) size devices
Embedded sensors
Context aware Information Analytics
Process embedded Feedback
Edge Connections
Telemetry and Content Platforms
Real time Feedback
Information Context Aware
Networks Mesh
Internet of Things

Figure 2.1: Ubiquitous Computing

puting as Ubiquitous Computing (UbiComp). Figure 2.1 shows that people will be surrounded by Devices, Networks, and Space [31]. In our daily tasks, a computer-enabled environment will be highly coupled with us where heterogeneous devices interact with each other to make different content (eg. video, audio or documents) available for their users. Weiser also predicted that computers would have to deal with the problem of interconnection, communication and of combining their capabilities to achieve a seamless experience for users and disappear into the background [65]. Cross-Device Interaction can be seen as a part of their vision because we interact with multiple different devices in a certain environment that need to connect and work together to have a spontaneous user experience. Another term that means the same is Pervasive Computing. Posland [46] describes UbiComp in three requirements: (1) Computers need to be networked, distributed and transparently accessible; (2) Human-computer interaction (HCI) needs to be hidden more; (3) Computers need to be context-aware in order to optimise their operation in their environment.

Saha and Mukherjee [50] described six fundamental challenges for Ubiquitous Computing and distributed interactions:

> ***Scalability:*** how to deal with a lot of more devices where explicitly distributing and installing applications will become unmanageable, especially across the wide geographic area?
>
> ***Heterogeneity:*** as heterogeneity increases, how to develop applications that run across all system platforms?
>
> ***Integration:*** how to integrate all the already deployed components to many environments into a single platform? How to deal with the coordination between the distributed spaces?
>
> ***Invisibility:*** how to find the right balance between human intervention environment and self-tuned smart environments? How to find automated techniques to dynamically support the connection of devices and applications?
>
> ***Perception - Context Awareness:*** implementing perception from different sensors introduces significant complications. How can we know that the information that defines context awareness is accurate?
>
> ***Smartness - Context Management:*** smartness involves accurate sensing(input) followed by action (output) between machine and human. How can we translate this into something meaningful and understandable to the end user?

A wide range of multimedia and office applications are difficult to run in a pervasive computing environment. Therefore, cloud services from big companies like Amazon, Google and Microsoft try to make certain platforms available to overcome these restrictions.

2.1.3 Cloud Services

Cloud services can help us to share data (eg. documents, pictures or videos) across multiple personal devices. We can start editing a document on our desktop computer and continue our task on our laptop. This is a typical example of a sequential cross-device task. Big players of such file-hosting services are Microsoft Onedrive[1], Apple iCloud[2], Google Drive[3] and Dropbox[4].

[1]https://onedrive.live.com/about/en-gb/

[2]http://www.apple.com/icloud/

[3]https://www.google.com/drive/

They allow us to store files or certain settings in the cloud, that can be synced to all our personal computers who are connected with the Internet. But, cloud services have one important shortcoming that is also described by Hamilton et al. [26]. We cannot utilise multiple devices at the same time for the same user. For example, editing a document in real time on our laptop and our desktop computer. The main reason is that all these services need an account to log in and a high-bandwidth internet connection. A remote centralised server will handle all the communication between the different devices. With an invitation to other users, we can share our workspace with other users, so that we can edit the same document at the same time. We can speak of a collaboration of multiple users on the same document, where we can see who is logged in and where he is typing text in the shared document. This feature works very good in Google Docs and MS Word if both users use the same application and each user is working on only one device at the same time. Another shortcoming of all these services is that they are unaware of devices that are surrounding them. This lack of awareness makes it difficult for the users to make a spontaneous connection. Finally, cloud services have a lot of security and privacy challenges [61]. A lot of (banking) companies are reluctant to place data in the cloud because they have a lot of sensitive information about their clients. Hence, they are using their own data centres. There have been already a few incidents with data that is stored into the cloud and which have fallen into the wrong hands. Therefore, a lot of privacy-protection mechanisms must be provided.

To support a much more cross device interaction for multimedia content, each company has their own hardware that can make a connection with the television: Microsoft Xbox One[5], Apple TV[6] and Android TV[7]. Thanks to these hardware solutions, users can also add media content into the cloud for watching them later on the television. For example, Dropbox has recently made a Windows app for the Xbox One to play the cloud multimedia content with the help of the Microsoft game console to the television for users who have not yet an Apple TV or Android TV or Chromecast[8].

[5]`http://www.xbox.com/en-GB/one`
[6]`http://www.apple.com/uk/tv/`
[7]`https://www.android.com/tv/`
[8]`https://www.google.com/intl/en_uk/chromecast/`

2.1.4 Distributed User Interfaces

User Interfaces (UI) are the space where interactions between humans and machines occur. Mostly, we have the classical Graphical User Interface (GUI) which is designed for interaction with input (keyboard, mouse) and output (screen) devices that are attached to the single computing device unit. Hutchings et al. [30] talked about the term Distributed Display Environment (DDE) when they refer to a certain system that presents output to more than one display. In the 1980s we have the popular Nintendo Donkey Kong game (Figure 2.2a), which had two small screens above each other. We have one single device with multiple screens. Hence, we could also speak about the other more frequent term multi-display environment. Instead of one single device, we could use more than one device, a Multi-Device Environment or Multiple-Device Environment (MDE). Most of the time both terms will overlap. MDE refers to a physical workspace where multiple devices are networking together. For example, we can couple multiple screens to a desktop computer to create a multi desktop environment. Figure 2.2b is an example of TDome[9], a special touch-enabled input and output device that facilitates the interactions between multiple displays. Multiple devices will interact with each other to complete a task. So, the execution of the task must span more than one device.

In order to achieve Distributed User Interfaces (DUI), it is important to have responsive UIs. A lot of current websites have already a responsive UI when the UI is presented on one single device and where the UI is adapting the layout depending on the size of the screen. A popular framework on the web to create responsive UI is Bootstrap[10]. DUIs is where we have different UI elements that can be distributed across users, platforms, and environments. For example, by coupling different monitors to the same workstation, we will distribute the UIs across the monitors. A definition for DUIs is given by Elmqvist [19]: "*A distributed user interface is a user interface whose components are distributed across one or more of the dimensions input, output, platform, space, and time*". In other words, we are distributing the UI over multiple places. These places can be other devices, but multiple screens of the same device are also possible. If we have one big single UI, it can be moved or copied between different devices. On the other hand, if we have small single UI elements that are grouped, we have the possibility to distribute the small single parts across various devices. Then, we are talking about the granularity of a DUI. We can even have a duplicated distribution.

[9] `http://www.marcanudo.com/mde.html`

[10] `http://getbootstrap.com`

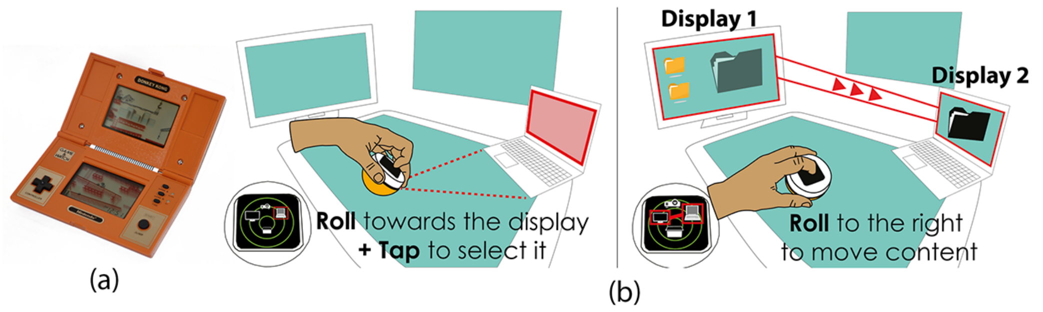


Figure 2.2: (a) DDE - Donkey Kong
(b) MDE - TDome a novel touch-enabled input and output device

Hence, we talk about DUI when we are distributing parts or the whole UI across various devices according to the input and the output possibilities of these devices.

DUI separate interfaces or content and will cast, migrate or merge seamlessly these interfaces between a set of devices. DUI can also adapt the interface to different platforms [36] and try to achieve a continuous application experience [64]. We have (at least) two devices involved in the rendering of the DUI. The distribution of the UI can be dynamic [7] when the UI elements can vary their location to the device during a user session. The opposite of dynamic is static when the distribution configuration cannot change during a session [7]. The big advantage of DUI is that it allows users to interact with an application through multiple devices at the same time, we can speak about Cross-Device UIs. Deep Shot [9] is a framework that will help the users to continue their task in its current state on another device by taking a picture with the camera of their mobile device of the computer screen where they were working on. This framework is based on a client-server architecture and we will see that a lot of frameworks have a client-server architecture. The shortcoming of such systems is the lack of support for offline interactions.

Sanctorum and Signer [51] described a classification of DUIs by looking to the supported *granularity* for DUI components in combination with the interaction *space*. They have also highlighted the systems that support the distribution of *state*. In Chapter 1, we have for example mentioned Deep Shot [9] where the state can be distributed by using the mobile phone camera. For the space classification, they have described four categories of space restrictions. The most limited space is with a table or camera [47]. The second category is restricted to a room because for example the need

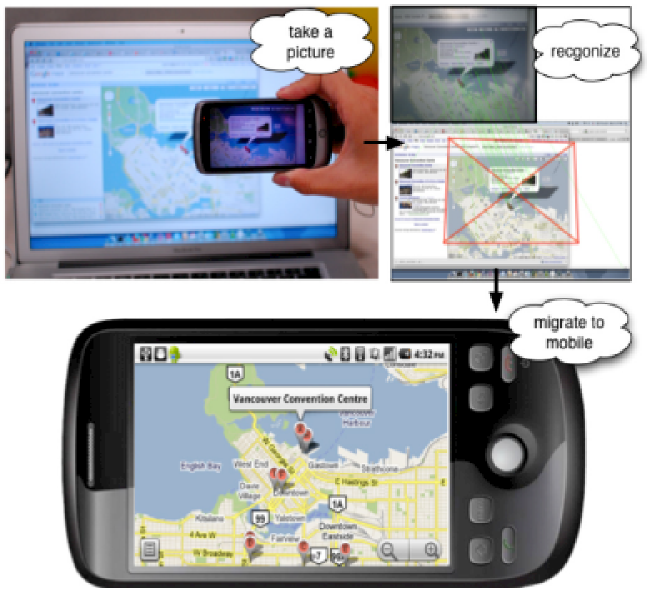


Figure 2.3: Deep Shot: continue the state

of multiple cameras [6]. The third category is when there is a network connection to a server [58, 66, 26, 9]. The last category is the space without constraints where DUI systems can be used anywhere [54, 11, 4, 35]. Sanctorum and Signer also determine two types of granularity. The first type can only distribute the data as a whole while the second type can support a distribution of parts of the data. For example, the distribution of some pixels of an image and not only the whole image. If we want to have the most flexible DUI solution with the least restrictive possibilities, then we must search for a DUI that can be used anywhere with a support for high or fine granularity. Weave [11] is an example of such a framework for developers of web applications to create cross-device wearable interactions by scripting based on JQuery[11]. Weave has some APIs to distribute UI and combine sensing events and user input across mobile and wearable devices. Weave has also a web-based programming IDE (Integrated Development Environment) for helping developers to create and test Weave applications.

Instead of looking to the supported granularity and the interaction space, Demeure et al. [16] described another classification of DUIs. Their approach is based on the four dimensions of computations, communications, coordination and configuration. We will use the classification of Sanctorum and Signer.

[11]https://jquery.com

2.1.5 Cross-Device Interaction

Cross-Device Interaction (XDI) is still an ongoing investigation because more and more new devices are invented and there is a need for communication between all these different devices. Just think of the wearable technology, with the well-known smartwatches which need to communicate with the smartphone. XDI remains a challenging research like already mention in Chapter 1. Scharf et al. [53] give us a definition for XDI: "*Cross Device interaction (XDI) is the type of interaction, where human users interact with multiple separate input and output devices, where input devices will be used to manipulate content on output devices within a perceived interaction space with immediate and explicit feedback*". Robertson et al. [49] presented a first one-directional interaction from the stylus pen of a personal digital assistant to a television. Rekimoto et al. [48] showed us a technique to exchange information in both directions between one computer screen to another screen with a stylus pen. Handheld devices are often used as a remote control to manipulate digital content on a large display. Instead of using the stylus pen as a remote control, we can use the stylus pen to make the connection between the multiple devices. Stitching is an example of a more recent framework to connect two pen-based tablets by stitching them together [27].

Instead of using a stylus pen, there exist a lot of other interaction techniques. Some applications make use of one or more cameras to provide interactions [9, 47, 42, 35]. Another popular technique is the use of QR codes to make the interaction between multiple devices [22, 66, 18]. We still see that technique today for transferring mobile payments between two different users or to complete an online payment by scanning the QR code on the screen with the user's banking mobile app. We can have different interaction methods: touch gestures, rotation change, shake events and motion gestures [51]. We can use these to trigger a cross-device interaction. Paternò and Santoro [44] described different possible triggers. The first one is user-initiated, where the user is triggering the interaction by selecting the elements and select the target device and what he likes to change. When the system triggers the interaction, we speak of system-initiated. They speak about mixed-initiation when the trigger is partially determined by the user and partially suggested automatically from the system.

In the device ecology, we have multiple devices that can interact with another device and they are in physical proximity to each other. These devices will work together to perform a certain task for a user. We can, for example, have a *car device ecology* where all the devices in the car may perform the task to listen to some music that is located on a certain device. Cross Device Interaction is more the description of the interaction between the devices of a device ecology. Before an interaction can occur, the devices get aware of each other and start to transfer information that is useful for the users. This can be a combination of their hardware capabilities. We can speak of an ad-hoc XDI when the device ecology is created without an installation of extra software by the users or setting up some special hardware. Ad hoc is when it is made only for a particular purpose without a planning ahead. Other terms that are also sometimes used for XDI are cross-surface, multi-surface and multi-device interaction.

2.2 XDIs in Everyday Activities and Tasks

Jokela et al. [32] have done a study on how the combination of multiple information devices is done in everyday activities and tasks. We will notice that the diary study has a lot of similarities with the research of Google their publication "The New Multi-Screen World" [25]. We have already mentioned their publication in Chapter 1 to show that we are living in a nation of multi-devices. Google talked about sequential and simultaneous screening. Jokela et al. [32] have identified four main ways on how people used their devices. They have only studied the area of single-user with multi-devices, not for multi-users. The distribution of the four main ways is shown in Figure 2.4.

Sequential Use: 37% of multi-device cases where sequential. This form is corresponding with the sequential screening of Google [25]. Users are completing their single task by changing their devices during a task. But they do not use these devices at the same time. For example, they are searching some information about a product they want to buy on their smartphone, but they complete the task by buying that product on their laptop.

Resource Landing: 27% of multi-device cases where resource landing. The focus of the users is on one specific device while they are borrowing resources from other devices. For example, connecting a laptop (a resource of another device) to a television (focused device) to watch the movie located on the laptop on a bigger screen.

Related Parallel Use: 28% of multi-device cases where related parallel use. This is corresponding with complementary-usage of Google [25]. All devices are involved in a single task at the same time. For example, using our smartphone for a slideshow on our television.

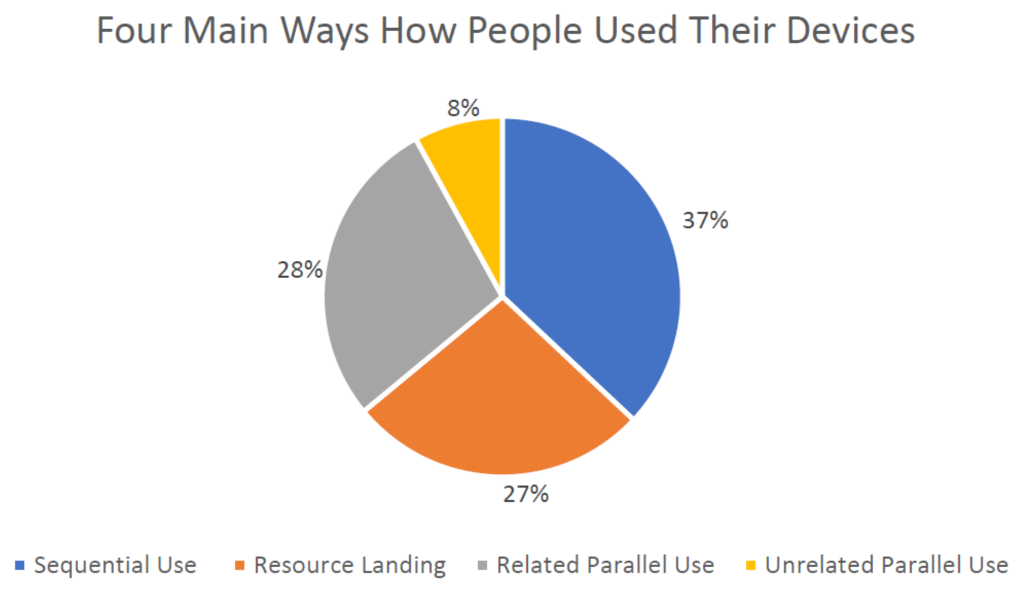


Figure 2.4: Four main ways on how people used their devices

Unrelated Parallel Use: 8% of multi-device cases where unrelated parallel use. It is corresponding with multitasking of Google [25]. Devices are involved in different tasks. For example, playing a game on a smartphone and watching a boring football match on the television. We have a foreground task (the game) and a background task(watching TV). Another possible example is working on a laptop, while the user is listening to music with their smartphone.

Supporting the "parallel use" continues to be a challenge. The same conclusion was made by Santosa and Wigdor [52]. They called it the "barriers to parallelism" [52]. In their field study of multi-device workflows, they have described four workflow patterns, one sequential pattern and three parallel patterns:

Producer-Consumer: a sequential pattern where the searched information is going from one device to another device.

Performer-Informer: a parallel pattern where a device is used for reference while the main work is done on another device.

Performer-Informer-Helper: the same as the second pattern but with an additional device (helper) as another helper device. This helper device can be used for example for extra calculations or quick look up.

Controller-Viewer/Analyser: a parallel pattern where various aspects of a single task were executed on multiple devices. The best example here is using our smartphone as a remote controller for a slideshow on a television.

Santosa and Wigdor have also mentioned some interesting shortcomings for parallel patterns. The first shortcoming is the lack of support for moving documents or applications states between different devices. They formulate it as follows "*participants with many devices available have the opportunity for rich parallel usage, yet they remain limited by the lack of cross-device interaction supporting this*" [52]. The second mentioned shortcoming it the difficulty to know the location of nearby devices without additional hardware. They formulate it as follows "*parallel patterns in particularly would benefit from devices being mutually aware of each other's location, proximity, and orientation*" [52].

2.3 Challenges of Cross Device Interactions

Many questions arise when we need to develop a particular solution with an interaction between different devices. We usually deal with different types of devices, ranging from laptops, fixed computers, head mounted displays, smart watches, but also interaction with IoT devices or real objects via RFID tags. We can have devices that can only track input request, like sensors, eye tracking, gestures or voices. Other devices can maybe produce only output request, like information screens without input devices like a mouse or keyboard. But, most of the devices can managing input and output requests, like smartphones or tablets. We see, what is already mentioned by ubiquitous computing, an evolution where these devices become more and more a part of our daily life. Which technology can we use for these interactions that take place between these devices? Can we do an ad-hoc connection? Do we need some client-server architecture to store some data in the cloud?

Hereby, *Detection*, *Tracking* and *Communicating* are three important terms for cross-device interactions. Marquardt et al. [38] have described the Gradual Engagement Pattern, where many interaction techniques and approaches have been proposed for the distributed configuration problem. Before a device can interact with other devices, it must be aware of the other devices and their capabilities. Some systems will only do a detection[22, 27, 26], others will also track the devices[40, 29, 47, 39]. Understanding what information can be used on each device. Marquardt et al. have summarised three important challenges within multi-device environments [38]: (1) Device Pairing, (2) Awareness and revealing of information and resources and (3) Transferring information from one device to another.

2.3.1 Detection - Device Pairing

For Device Pairing, we typically think about two important steps. First, we must identify the target devices and afterwards we must setup a secure communication channel between these devices. Both of them are already big and difficult challenges to treat. If we also want that the different devices have some awareness of each other's position then we are facing another additional challenge that we will discuss in more detail in the next section. For example, as a person, we can see with our eyes that two or more devices are lying next to each other on a table and we can see some mutual structure between them. But, it is a challenge to find some algorithm so that each device can detect another device and vice versa but also to know their mutual positions.

Shortcomings

A pairing process that is frequently used for device pairing is an explicit pairing system. For a Bluetooth connection between a smartphone and a Bluetooth speaker, we can create some shared secret key, like a pass phrase or some PIN code between these two devices. The disadvantage of this process is that each pairing process requires some human input to manually connect these two devices. Bluetooth is one of the more popular short-range wireless technology today for device pairing. The mobile devices can make a connection with each other using radio waves technologies such as Direct Wi-Fi, Bluetooth or Near Field Communication (NFC), where each connection has their own advantages and disadvantages. Apple Continuity[12] is very popular because it allows a data exchange with nearby devices and we can even transfer the current application state from one device to another. But, it only works between iOS-devices. A similar tool exists for Android and Windows devices, namely Airdroid[13].

Hence, the **spontaneous device sharing problem** is a limitation for a lot of current technology. We have to know the other device name so that we can manually configure the connection between the different devices. Most of the time, extra software installation is needed or we need some special hardware, like a camera, or some synchronous gestures to make the connection. Examples of such a gestures are the following, shaking the devices, using a

[12]`http://www.apple.com/macos/continuity/`

[13]`https://airdroid.com`

pen, bumping or simultaneously pressing a button or some other movements to connect the devices. For every new device, we need to repeat these gestures. For example, Frosini et al. [22] can dynamically add or remove devices with the use of a QR code. Stitching [27] has an easy device pairing because they are connecting the two devices with their pen. Hence, no computer names are needed to connect the different devices. The position of the devices has an important relation. We can copy images by stitching from one device to the other with a feedback of the transfer to the users.

Existing Solutions

While Apple Continuity is using the Bluetooth connection, Conductor [26] is a framework that uses NFC for making devices to join a cross-device application. This will work when two NFC-enabled devices come close to each other. At that moment they establish a communication channel to exchange information over Wi-Fi. This will allow the devices to connect to the same remote WebSocket server. Certain companies use nowadays Bluetooth beacons to *detect* how long clients are in their branch office and to propose them to give some feedback after their visit.

2.3.2 Tracking - Awareness

Awareness is another important concept when we are using multiple devices. It is handier for the user if there is good awareness of device presence. Device augmentation can be a way of making devices aware of surrounding devices.

Shortcoming

If we could have some physical awareness of the devices, we could also send parts of an image to the different devices so that we can, for example, spread one picture on four devices to see one larger picture. This option is also missing in Conductor and Sanctorum and Signer [51] described that Conductor and a lot of other systems support only the distributions of applications as a whole. Users cannot transfer only parts of an image or a document. In certain situations, this may be a shortcoming and it is better to have a system that at least supports a fine granularity. Therefore, the devices must be first aware of each other and we will describe some existing solutions.

Existing Solutions

The most effective solutions for making devices aware of their surrounding devices is by using some sensors. Infra red sensors can be used. It will only work

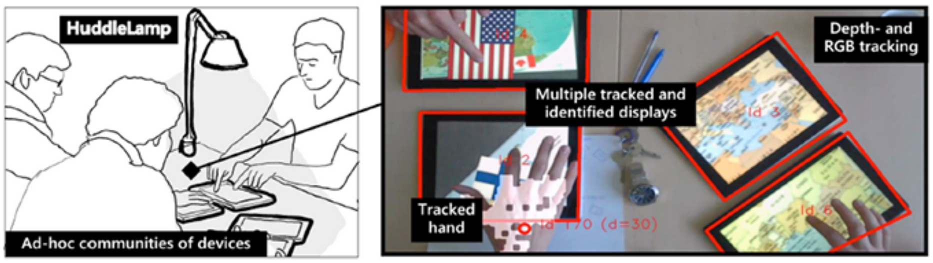

HuddleLamp
Ad-hoc communities of devices
Multiple tracked and identified displays
Depth- and RGB tracking
Tracked hand

Figure 2.5: HuddleLamp: a desk lamp with an integrated RGB-D camera

if there is a direct link between the sender and the receiver. Merrill et al. [40] described Siftables which are small cubical devices with a colour LCD, an accelerometer and four infrared receivers. This last one will allow them to detect other Siftables. Instead of infra red sensors, MagMobile [29] uses magnetism for detection. Detecting and tracking other devices can be done because bringing two magnetic fields close to each other makes them distort. They have developed a external magnetic sleeve to detect other devices. Therefore, to continuously track and calculate the relative position of other devices, they have attached an Arduino board to the sleeve that sends measurements to a shared server.

Another technique for device pairing is by using one or more camera(s). The cameras will detect and track people and devices by using image processing algorithms. An example of such a system is HuddleLamp [47] which mounts a camera into a desk lamp and the camera will track all the devices that are lying on a flat surface like a desk (Figure 2.5). This system has a very limited space range, but on the other hand, it is cheap. The Proximity Toolkit [39] is a more expensive alternative because this system requires an entire room with cameras and a 3D model of the room. But, we have a greater space range and it can track devices, people, gestures, posture and non-digital objects. Therefore, there are many more options compared to Huddlelamp.

Device awareness can also be met by the use of Acoustic Sensing, which is a technique that uses the microphone of the devices to track device positions. SurfaceLink [24] proposed an approach for the detection of devices using acoustic sensing. All the devices must be first placed on a flat, rough, suitable surface. We can then connect two devices by moving with the finger on the surface from device A to device B. Both devices have detected that sound on the table and by communication that information with each device on the table, both devices are able to determine their relative position.

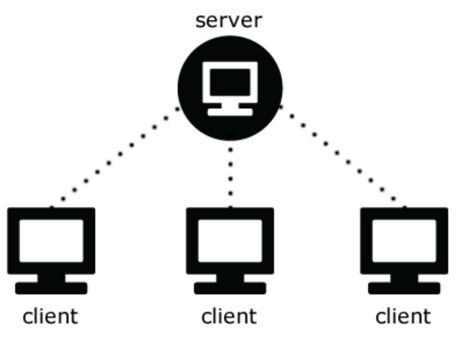


Figure 2.6: Client-server architecture

2.3.3 Communication - Transferring of Data

In the two previous sections, we have described different technologies to make a spontaneous connection and the awareness between various devices. In this section, we will dive deeper into how we could exchange such data information. We can divide this is in three sorts of communications: client-server, same wireless network and short-range wireless communication. We will mention some shortcomings and existing solutions for each of them.

Client-Server Architecture

The *client-server architecture*, which is shown in Figure 2.6, is the most frequently used setup for communication between different devices. The setup is not difficult and straightforward. We have a shared remote server and all the devices know already how they must connect to that remote server. They have a knowledge of TCP connections, WebSocket or REST to make that connection.

A lot of applications and frameworks [26, 18, 9, 66, 58] are using this architecture. Hence, they need to know the server beforehand and a shared login is needed to make the connection to the remote server. Hence, it limits the possible interaction space. Most of these frameworks are web applications and are deployable on a wide variety of web-enabled consumer devices. Direwolf [34] needs an internet connection and it uses widgets that can be shared, reused, mashed up and personalised. A widget is a self-contained mini-application, but it only works for web applications. Instead of widgets, we can find a little bit the same principle in Panelrama [66], where the complete UI is divided into different panels by surrounding a specific group of HTML5 code with the panel tag which is illustrated in Listing 2.1. Properties are attached to each panel using JavaScript (Listing 2.2), that will allow to distributing each panel to different devices that have an internet connection. For example, a video to a large device and the remote controller of the video to a proximal device. The web applications are following the single user on a single device computing model. It is a pity that a lot of frameworks support multi-devices that only works for a single user. To have a good interaction between a lot of different users and devices, we must have a support for a multi-device, *multi-user collaboration*.

```
<panel name="VideoPanel">
  <div class="{{panelType}}" id='{{panelId}}'>
    <iframe id='player' class="youtube-player"></iframe>
  <div>
</panel>

<panel name="PlaybackPanel">
  <div class="{{panelType}}" id='{{panelId}}'>
    <button class="button" id="play">Play</button>
    <button class="button" id="pause">Pause</button>
    <button class="button" id="stop">Stop</button>
  </div>
</panel>
```

Listing 2.1: Sample HTML template with panelType and panelId

```
function VideoPanel(stateVariables, id) {
  this.type = "VideoPanel";
  this.typeName = "Video Panel";
  this.affinity = {physicalSize: 5};

  // playbackState may alternate between "play",
  // "pause", and "stop"
  this.stateVariables = {
    playbackState: {defaultValue: "stop", sync: true}
  };

  Panel.call(this, id);
}
VideoPanel.prototype = new Panel();
```

Listing 2.2: A panel definition

Instead of using a client server architecture, Frosini et al. [22] are using a client side (presentation) and an engine side (distribution manager). Hence, an internet connection is not always needed and it supports offline interactions. Weave [11] and Connichiwa [54] are other examples of frameworks that can be used anywhere. There are using a peer to peer (P2P) connection that can be realised with a Bluetooth[14] connection or with a Wi-Fi connection where one device is the host. Stitching [27] is also not limited in space, because the stitching server is hosted on one of the devices. ReticularSpaces [4] is based on the concepts and principle of Activity-Based

[14]`https://www.bluetooth.com`

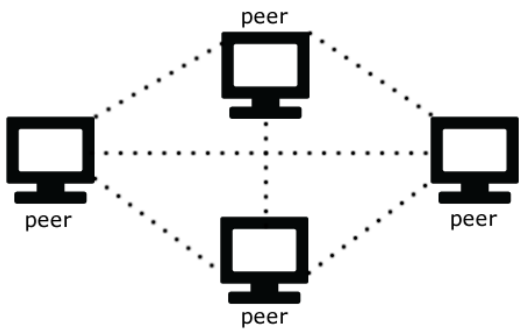


Figure 2.7: P2P Architecture

Computing (ABC) [12]. In contrast to the 3rd and 4th generation of the ABC infrastructure where the Activity Manager is located on a central server, each peer (client) has an instance of the activity manager running. Each client can also run the UI client which knows the network address of its activity manager. So, we have much more flexibility for adding and removing devices without a space limitation.

Same Wireless Network

Instead of using a shared remote server like in the client-server architecture, all the different devices can make a connection to the same wireless network. An internet connection is not really needed and devices can communicate directly and send information to each other. A popular system, therefore, is a Peer-to-Peer (P2P) networking system which is illustrated in Figure 2.7. If we look to the evolution of P2P, we can distinguish two different generations of P2P. The first popular P2P network system example was Napster. A very popular P2P network for sharing music files. It has a cluster of dedicated servers that maintain the index for all the surrounded peers. All the peers in the second generation have the same functionality, which makes it a pure P2P architecture. Each peer can discover and query for resources. A remote server like in a client-server architecture is not needed, each peer can act as a server as well as a client. Gnutella was the first decentralised protocol for P2P applications. Decentralised means that there are no servers, but there are only clients in the network. Unlike Napster, where the entire network relied on the central server, Gnutella can not be shut down by shutting down any one node and it is impossible for any company to control the contents of the network. Another example of a popular P2P network is the Bitcoin network which is described by Satoshi Nakamoto [41]. This is a very robust digital payment system which has no central authority like in a traditional bank transfer. And it has solutions against malicious entities and entities who goes offline, thanks to their P2P network in combination with cryptography and the Block chain.

Working on the *same wireless network* can be useful if an internet connection is not really needed to fulfill the device interaction. A problem here is that the user still needs some *knowledge* about the network and the needed configuration, which can be a challenge to detect the correct device on that network. Another challenge is security. Prevent eavesdropping is more difficult for wireless communications, then for a closed wired network. For working on the *same wireless network*, Lucero et al. [37] have proposed a solution so that devices do not need to know some knowledge about the number of devices or their network addresses. They will broadcast all received packets to all connected devices. Conductor [26] uses the principle "Cues" to broadcast a message to all the devices. Each user can accept the message and can continue their work on that device. There is also the possibility to select a target device to send the message. We can clone the session to another device and use one keyboard on multiple devices (duet).

Short-Range Wireless Communication

Short-Range Wireless Communication is already mentioned for device pairing. Bluetooth and NFC are the most used connections in commercial products. The Bluetooth Special Interest Group (SIG) has launched a new version of Bluetooth, called Bluetooth 5[15]. The enhancements of Bluetooth 5 is focused on increasing the functionality of Bluetooth for the IoT. With 4x range, 2x speed and 8x broadcasting message capacity (255 bytes instead of 31 bytes), it continues to evolve to meet the needs of the industry as the global wireless standard for simple, secure connectivity. Bluetooth 5 and beacons can broadcast promotions of shops, restaurants, etc. at the moment that someone is passing them or if someone leaves the shop they can open a website or app asking to give a little feedback. Beacons can also be used for indoor-navigation, where there is no GPS signal. Bluetooth 5 has again a reduced energy consumption version: Bluetooth 5 Low Energy. This version can create connections without user authorisation which makes Bluetooth again more interesting for new applications in commercial products.

NFC is not so popular, because not all commercial products have this option and a close proximity is needed. The connection is established when two NFC devices are bought within the distance of 4 centimeters. NFC is more used for making a handshake information between devices. NFC is very

[15] `https://www.bluetooth.com/specifications/adopted-specifications`

intuitive, because they only require a simple touch to make the handshake. Conductor [26] for example is using NFC for the initial handshake and thereafter the devices will join a common, remote WebSocket server. Therefore, Conductor avoids large amount of data over NFC, but on the other hand, Conductor requires at that moment an internet connection. Hence, Conductor can not be used anywhere. Android Beam[16] will share the current application state of another device, by putting the devices back to back. For example, to share the currently active website. So, with *Bluetooth and NFC*, it is easier to make a device pairing, but transfer *a large amount of data* is difficult. There is also a limitation of devices that can *connect simultaneously.* NFC only works for a really short distance between the devices. Bluetooth 5 tries to solve some of these problems. Bluetooth LE will be able to increase its stability and range in the future through a mesh network like for Wi-Fi. SIG has announced their support of a mesh technology[17] for establishing a many-to-many device communications. The big advantage is that Bluetooth and NFC are not depending on space and can be used anywhere.

2.3.4 Flexible and Customisable

If we investigate the frameworks for distribution interactions, we can determine most of the time that these are *built for specific applications* [58, 27, 18, 22, 34, 28], and therefore, they are not so flexible or customisable. For example, with Stitching [27], we can only interact between two devices with a pen and is not always accurate. Watchconnect [28] is an application specific for smartwatches who can interact with other devices.

A good approach to be flexible and customisable is the idea of component based software development (CBSD). We can then combine the different components to more complex systems [60]. The concept of active components (ACs) can be used to link pieces of components [57, 59]. Each AC has some program code that may be an application on their own or that can be coupled to have a richer application. Developers can reuse these ACs without having to write any code themselves. Only, when they need some new functionality, some programming effort is required to write a new AC. These ACs are standalone lightweight components that can perform some operations on the server or on the client side, see Figure 2.8. One advantage of ACs is that we can store this programming code in a DB and do some remote method invocation. The concept of active components, therefore, provides a clear separation of interaction design and application logic.

[16] `https://developer.android.com/guide/topics/connectivity/nfc/nfc.html`
[17] `https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works/`

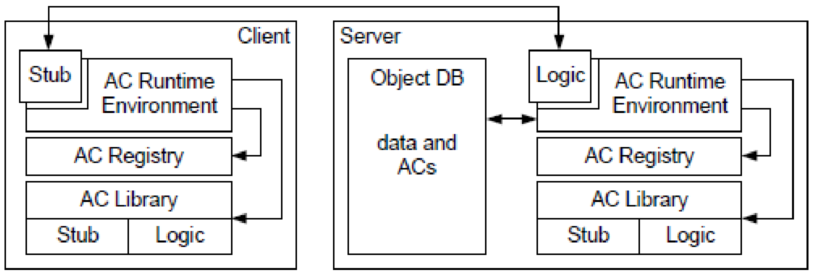


Figure 2.8: Active Components

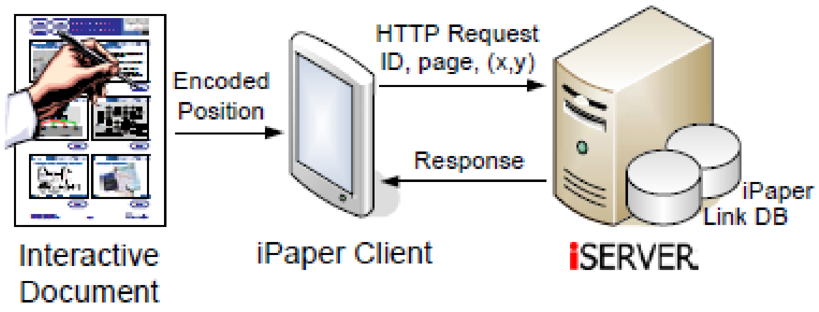


Figure 2.9: Paperpoint: client-server architecture

PaperPoint [58] is an interactive paper application that uses the idea of ACs (Figure 2.9). It is a cross-media information platform that enables links between arbitrary digital or physical resources based on a resource plug-in mechanism. We can turn the development of cross-media interfaces into an authoring activity rather than a programming activity for a maximum flexibility [57, 59]. It is easily extensible with the concept of *Selectors* and *Resources* [57, 59]. The selectors representing the elements within the resources. Hence, in PaperPoint, we have the shapes (selectors) on the pages (resources) (Figure 2.10). For a movie, we can have the time span as the selector and the source video as the resource.

Direwolf [34] describes widgets as a self-contained mini application with some limited functionality. There the user can mash-up multiple widgets in widget containers, that can be shared, grouped and personalised. Direwolf is based on a publish-subscribe mechanism. Android has an url based navigation for opening a browser (http:) or calling a telephone number (tel:) or opening a map location (geo:), which could be interesting.

`le-mesh`

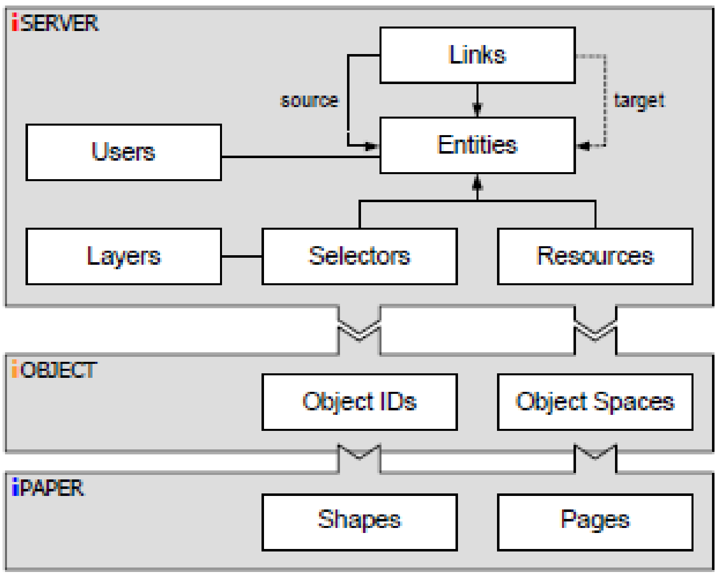


Figure 2.10: iServer Resource Plugin

Authoring tools are interesting because we need less programming skills and it can improve the maintainability and the rapid prototyping of applications. It can help to limit the gap between the concept design and the full implementation, but on the other hand, it is more difficult to make very specific application in order to meet the requirements of the stakeholders. This is also typically for software packages. We buy some core functionality. If the user customises that package, it will be more difficult to upgrade with the new possibilities of some future functionality. Maintainability will be better if we do not customise the software packages.

2.4 Interaction Techniques

There exist a lot of techniques to establish a connection between devices. We will give a small selection of commonly used techniques, but there exist many others. We have already mentioned some of them in the history section of cross-device interaction and in the section of device pairing and awareness. Table 2.1 gives an overview of the treated interaction techniques in this paper.

HuddleLamp [47] uses the camera inside the lamp and digital image processing to track devices on the flat desk, but also to track the hands of users. They have the *"pick, drag and drop"* gesture. Pick up an object of one device,

Interaction Techniques	System
Cross-Device pinching	Pinch [43]
Deep shooting, deep posting (camera)	Deep Shot [9]
Digital Pen	Stitching [27] Pick-and-drop [48] Dual Device User Interface Design [49]
Motion Based: tilting, tapping and jerking	JerkTilts [1] Tilt-And-Tap [17]; IntelliTilt [63]
Peek and pop	Apple 3D touch
Pick, drag and drop (camera)	HuddleLamp [47]
Real object (for example a pen)	Proximity Toolkit [39]

QR code scanning (camera)	CTAT [18];
	Frosini et al. [22];
	Panelrama [66];

Swiping, touching and tapping	All touchable mobile devices [51]
Voice	Siri Cortana Alexa Voice Service Google Now Google Assistant

Table 2.1: Interaction techniques

drag it and drop it on the other device. Proximity Toolkit [39] can detect with all their cameras a lot of device and user movements and non-digital objects. The authors describe a media player application, that can stop playing when users divert their attention to another user or a non-digital object such as a magazine. They describe how certain objects like a pen can be used to pause and resume a movie or show information about the movie when a user enters the room. Stitching [27] describes how a pen can be used to connect two different devices. It is a very intuitive pen gesture that is easy to understand for users. Pinch [43] describes a cross-device pinching gestures that are based on synchronised swiping gestures across two devices. A cross-device pinch is when there is a swipe to the edge on one device and at the same time a swipe to the edge of another device. Pinch knows the relative position of all the devices from these gestures. Tilt-And-Tap [17], IntelliTilt [63] and JerkTilts [1] explain some motion-based interaction techniques. They described tilting and tapping in combination with motion sensors. Tilting gestures are defined based on the speed an orientation of the device as measured by accelerometer and gyroscope sensors. Jerking action is more a rapid movement from one position to another. Often it is a small movement and in some cases may involve a movement forth and back so that the device returns to a resting position. Jerk tilting can be used as a toggle gesture for displaying or hiding menu items. Continuous tilting is when the system performs an action continuously according to the speed of the device. Both can be used in isolation or in combination with different other kinds of touch actions.

Another popular interaction technique is a voice. Apple has Siri[18], Microsoft has Cortana[19], Amazon has Alexa Voice Service[20] and Google has Google Now[21] and Google Assistant[22]. Google Now works from within the Google Android and iOS app, while Google Assistant is found in Google's Allo chat app[23] and integrated into Google Home[24] and Google's Pixel phones. Maybe in the future, the two virtual assistants will merge into one. Bank companies have now the possibility to make bank transfers by talking with Siri to our smartphone[25].

Swiping, touching and tapping are already frequently used gestures. Apple has unveiled since its iPhone 6S and iPhone 6S Plus a new type of screen that can detect different pressing levels with 3D touch and add consequently new gestures. Apple's new iPhones now recognises force as well as gestures and offer more accurate haptic feedback. This means apps are now more accessible thanks to variations in pressure offering previews, quick swiping and more. With Peek and Pop[26], users can have now more menus. “Peek” is when a user presses a little bit harder on the screen. “Pop” is when the user presses more. “Peek” can for example opening a pop-up with the content of an email. “Pop” will bring the user to a new place in the operating system. We can, for example, jump straight to a selfie cam from the home screen. There exist much more possible interaction techniques. Hence, Table 2.1 contains only a small part of the various possibilities.

[18]`http://www.apple.com/ios/siri/`
[19]`https://www.microsoft.com/en-us/mobile/experiences/cortana/`
[20]`https://developer.amazon.com/alexa-voice-service`
[21]`http://www.google.co.uk/landing/now/`
[22]`https://assistant.google.com`
[23]`https://allo.google.com`
[24]`https://madeby.google.com/home/`
[25]`https://www.youtube.com/watch?v=Gflp16fWTog`
[26]`https://developer.apple.com/ios/3d-touch/`

2.5 Design Guidelines for DUIs

Guidelines can provide us with a good structure with a corresponding standardisation around possible design issues. They can help us to propose some responses that may lead to the most wanted solution. It will help the developers and the designers to have some common principles to achieve their solution. Therefore, they must be aware and learn the different principles to apply them in their projects.

Fisher et al. [21] have presented three peer-to-peer distributed user interface applications: a media player, a multiplayer Tetris and a multi-device collaborative search system. They have derived general design guidelines for peer-to-peer DUI design:

Consistency: Keep a consistency between the user interfaces of multiple devices.

Synchronisation: All the actions of one device must be reflected on the other devices.

Heterogeneous Hardware: Allow various mobile and desktop systems to participate.

Volatile Device Ecosystem: Applications must cope with devices that can join or leave an application at any time.

Limited Resources: Applications must cope with limited resources on (mobile) devices.

Data Transfer: Applications should distribute limited resources automatically when necessary.

Physical Space: Devices can join or leave an application at any time, so they must be autonomous. One device should not be dependant of another devices.

Asymmetric Functionality: Sometimes it can be interesting that each device have some different functionality depending of their current device ecology. So, it make sense to distribute functionality asymmetrically between various devices.

2.6 Conclusion

Most cross-device frameworks are web based. A more general-purpose framework is missing and most of the frameworks are made for special purposes or require some special setup for example camera's, remote server, which made them difficult for commercial everyday products. Most of them are also made for special operating systems or special hardware.

Sequential interaction across devices is no problem anymore with all the already existing cloud services. But, none of them have some support for the parallel use of devices on the same task. And, cloud services can also be a security risk for data that is pushed to servers of different companies.

The focus is more on mobile devices instead of laptops and desktop computers (still used for more productive tasks). Cross-Device Interaction is very tied to mobile devices.

Cross-Device interactions can only work if they are fast and reliable. Remote servers can solve this, but then it is more difficult to make interactions that can work anywhere.

3 Proof of Concept Applications

3.1 Towards Rapid Prototyping of XDI

3.1.1 Focus

The focus is to start searching for effective ways for the development of cross-device interactions. We have seen in Chapter 2 that it is still a challenge for developers to create applications that work with multiple devices on the same time. Developers have to deal with the problem of how they will setup the different protocols to do the communications between devices and how the users can connect them in a user-friendly way. Adaptability and awareness are two important principles. Maybe the users have no Internet connection, but they still want to communicate between different devices close to them. Or they only want to broadcast a part of the video file. Hence, support for off-line interactions with a *high granularity* is the first requirement of our focus.

Reusabiltiy is the second important requirement. Don't repeat yourself (DRY) or Duplication is Evil (DIE) is an important principle for the maintainability of a software program. If we have the same code in multiple places, it is more difficult to change some logic in the repeated coding. It will take more time and a risk for higher potential errors in the coding. And if we have different reusable components, we can maybe combine these to have richer applications without a lot of needed coding. We want to reduce complexity. The principle of component based software programming can help to divide a system into pieces with a single responsibility. Pieces that can be put maybe into a framework. It is better to have many specific pieces than one general purpose piece. It is not needed to create pieces that will never be used. We create only a component when it is useful, we do not want to overwhelm users with too many pieces. Keep it Simple Stupid (KISS) and You Aren't Gonna Need It (YAGNI) are other principles for software engineering that we need to take into consideration where possible.

Having a *Distribution Manager* is the third important requirement, because we want to have a system that is customisable at run-time. A visual authoring tool can help users of our system to connect all these different components instead of programming them. Creating a distribution manager as a sort of visual programming language where users can easily adapt the functionality and distribution of the different components without modifying the programming code of the different pieces.

3.1.2 Prototyping and Rapid Prototyping

It is not so difficult to get a description of the clients to know what they want in global lines. But, getting complete, precise and correct requirements, is much more difficult. A possible approach is to create a Proof of Concept (POC) where we built and try some possible systems. Therefore, prototyping can help to create these trial versions of a system. Further, it will help to clarify all the functional and usability requirements for the user. Most of the time a POC is used to identify some possible critical design issues and to see if everything could work that way before the final version will be built.

We can have two different approaches for rapid prototyping (RP): iterative and evolutionary prototyping [5]. The goal of RP is to create quickly a wide variety of possible systems and choose one specific system and throw the other away. RP will help to detect and find problems long before the real start of the development.

Rapid Prototyping of XDI

The best way to help the developers to make a rapid prototyping of an application is by making a framework with parameters to create something rapidly.

Because we want to know for which problems a framework can help the developers and which components can be interesting for a cross device interaction application. We will first create two different proof of concept

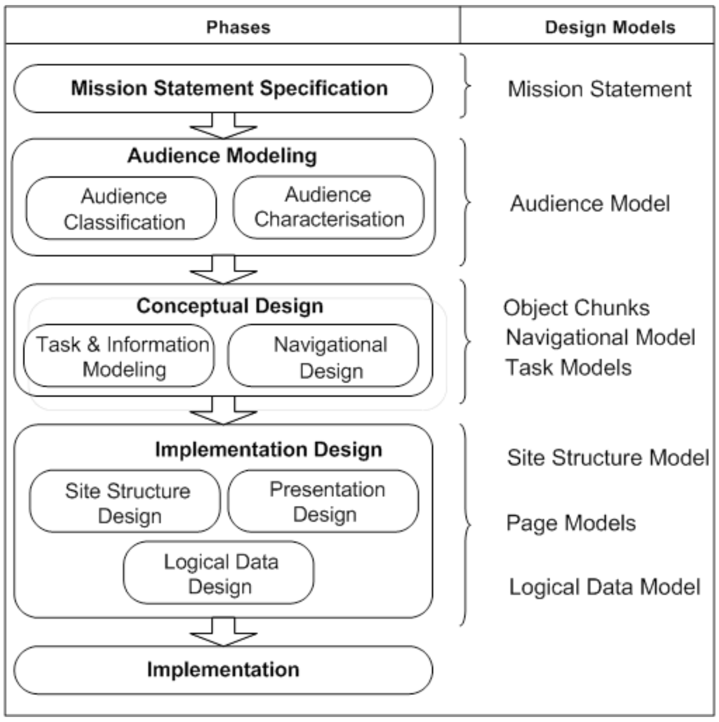


Figure 3.1: Steps for WSDM (Web Semantics Design Method)

applications that will have a lot of interactions with a possible high granularity and the support for offline and online interactions. We will explain the idea and the implementation of the two POC applications in Chapter 3 and 4. And finally, we will do some reverse engineering to propose a possible solution for a framework that is extensible, flexible and customisable. Hence, we will give some design guidelines for building such a framework.

3.2 Methodology

We will create two POC applications, that are not easy to implement or where we did not find any alternative solution. “CrossWoW Auto” is the first POC application that we will create. The second POC application is “CrossWoW Home” and is created to find some common source code that exists in both applications. With this information, we will propose a possible future library that can be created and investigated for future work. The library must have the ideas that are mentioned in the previous section.

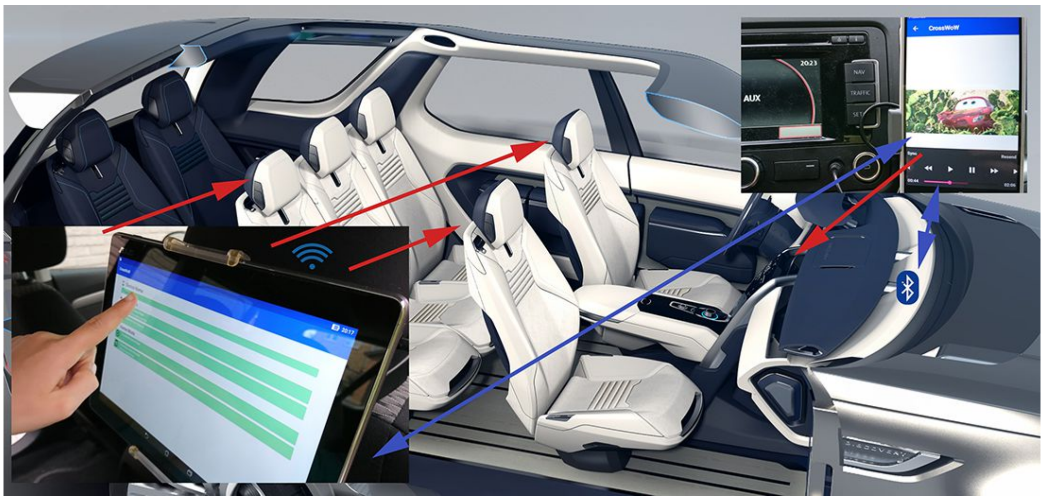


Figure 3.2: POC: CrossWow Auto

Before we will start with the real implementation, we will use a part of the Web Semantics Design Method (WSDM)[1] methodology [13, 14] to come to our paper prototypes. Figure 3.1 shows the different steps of the WSDM methodology. There is also a cycle in this methodology. For example, we have adapted our first prototypes because of logical and more technical reasons that we have encountered during development. We will explain each step of this methodology in the following sections.

3.3 Description of the POC Applications

3.3.1 CrossWow Auto

The purpose of the CrossWoW Auto application is to distribute media files from one device to all the other connected devices with some fine granularity and without the need of an internet connection in a user-friendly way.

We want to use it in the following scenario where a family with 3 children goes on a holiday trip with their car. The father and the mother have both a smartphone and the kids have all their own mini-tablet. The parents want to share media content (video, images and audio) to the multiple devices in their car. Figure 3.2 shows a visual representation of the wanted scenario in our car where the main device is the smartphone of one of the parents which is in front of the car. The kids have attached their tablets on a special "tablet holder" which is attached to the front seats. The parents want to broadcast

[1]`https://wise.vub.ac.be/content/wsdm-web-semantic-design-method`

their media content to some specific devices at the back of the car. They want to decide which device may have the remote controller (RC) that can decide which content will be played on the different devices. The image of the video will be played on the different devices, but the audio of the movies must come from the radio boxes. Hence, the main device will send the audio to the radio boxes and send the images of the video to the different client devices (publish subscribe principle). RC must work with gestures on the device or with some limited voice control if the main device is controlled by the driver of the car. It is also handy if we could copy configurations between the different devices of the parents.

Why

If each child has their own DVD player and they played all their own movie, the parents hear a mix of audio in the car, which is a small room. Furthermore, the audio of each device goes always higher because each child want to hear clearly the audio of their movie and this can lead to different discussions in the car.

Goal

Broadcast the image of a video file to the different devices and send the audio of that video to the radio boxes (with some possible granularity) with support for offline interactions. The app must have some *Distribution Manager* on the engine side, which can block or allow some actions on all their client sides. The *Distribution Manager* can allow certain devices to have a remote controller with support for voice commands.

3.3.2 CrossWow Home

The goal of the CrossWoW Home application is to find some common source code because this application will have the same purpose as CrossWoW Auto. Namely, distribute media files to other devices or to give them a remote controller to decide how they want to interact with the media files of the main device. Figure 3.3 shows a visual representation of the wanted scenario at home where a user has a lot of media content on their mobile device and wants to show these content on another device, like a SmartTv or tablet for example. The main user can also decide to show a RC with another user of a device. The targeted user(s) can then decide when they want for example to see the next picture of a slideshow or even download some of the pictures.

In Figure 3.3, the right device has got the RC and can decide how he wants to see the video in the middle that is located on the left device.

Why

People can show their vacation images on the television of their friends or parents, where they can decide when they want to see the following image. They can also copy the wanted images directly to their smartphone if the main user allows this.

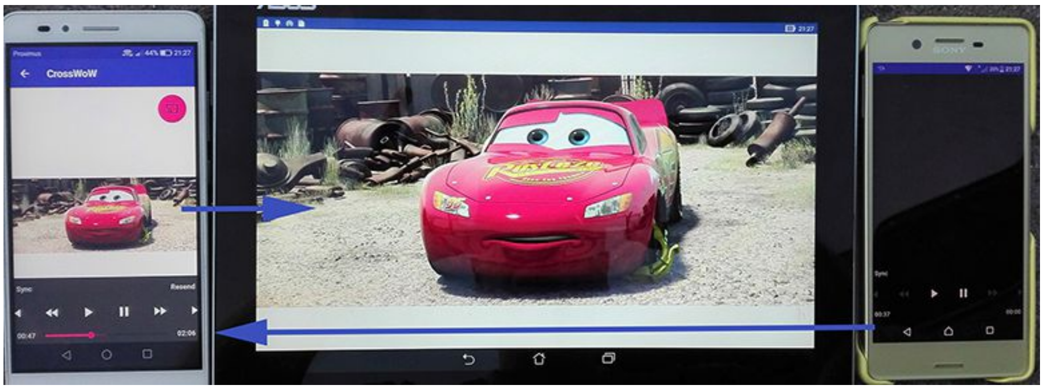


Figure 3.3: POC: CrossWow Home

Goal

Show and share media content with other people in a friendly way.

3.4 Mission Statement Description

The purpose of the POCs is to provide a mobile app where users (Head Users) can show and share media content with other users (Client Users) in a user-friendly way. Client Users can also become a Head User if they want to share their media. The subject is a different kind of media of the head user with a high granularity. Hence, the target audiences are the head user and the client users. The head users are the people that will share or show their media content. The client users are the people that can download and control the media of the head user.

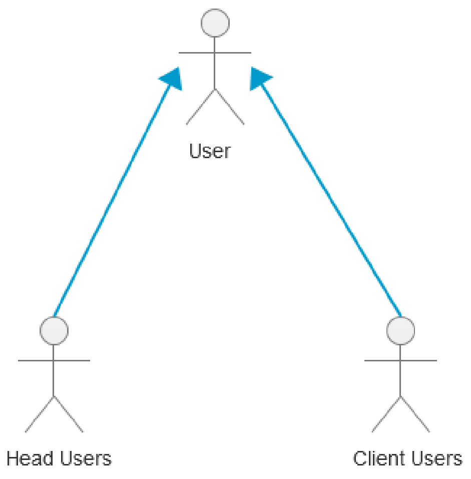


Figure 3.4: Audience Classes: Head Users and Client Users

3.5 Audience Modeling

3.5.1 Audience Classification

Figure 3.4 shows that the Audience Classification consist of two types of Audience classes: Head Users and Client Users.

3.5.2 Audience Characterisation

Table 3.1 gives an overview of the different characterisation for the head users and the client users. Basically, there is no difference between the two Audience classes. For CrossWoW Auto, the Head User is an Adult and the Client Users are children. They are both using the app in the car. Hence, for usability, big buttons are needed. For CrossWoW Home, both audience classes are normally adults or teenagers.

3.6 Requirements

The data that is needed in our application are media files. We must at least support an interaction with the local media files of the main device. The application must support MP4 video files, MP3 audio files, and the local images. In the ideal solution, we would like to support multiple devices on multiple OS. In our POC, we will only create the application in the native Android language. But, we must search for an implementation that also could work on iOS for example.

Characterisation	Head Users	Client Users
Type of User	Direct	Direct
Motivation	Share media content	Download, share or stream media content
Task Experience	Spontaneously	Spontaneously
Frequency	Occasionally	Occasionally
Task Knowledge	Spontaneously	Spontaneously
Use	When needed	When needed
Computer Literacy	Average	Average or Low
Number of Users	No limitation	No limitation
Training	No	No

Table 3.1: Audience characterisation for head and client users

3.6.1 Information Requirements

In the POC we will only support the English language. Table 3.2 gives an overview of the used information.

ID	Description
IR001	Information about the goal of the app
IR002	Information about the different available media content like video's, images and audio files
IR003	Information about the connected head and client users and their restrictions

Table 3.2: Information requirements for CrossWoW Auto and Home

3.6.2 Functional Requirements

Based on the descriptions of the POC applications, we have made a list of all the functional requirements (FRs) with their corresponding usability requirements (URs) in the ideal solution. The result of the analysis can be found in Table 3.3. We have described the functionalities of some Distribution Manager for the main user who can block or allow some actions on the target devices with support for some granularity.

Besides these requirements, there are some obvious requirements that are typically needed for mobile apps: (1) users can download the mobile application for free via the app store; (2) users must be aware of new versions of the app and (3) new versions of the app must work with an older version of the app (not for the new features).

FR	UR	Description
FR001		Allow to give a specific device name and modify information about the device
	UR001a	Allow to define the name of the device when the application starts the first time
	UR001b	Allow to see the name of the device on each screen
	UR001c	Show some usability information in the settings which can be changed at any time (default startup screen, default filtering)
FR002		Allow to easily browse and search for the different available media
	UR002a	Allow to search in the media in a user-friendly way
	UR002b	Allow to have only the necessary buttons and big buttons in car modus
	UR002c	Allow to search in less than 0.5 second
	UR002d	Allow to have some sorting (name, date, ascending, descending)
	UR002e	Allow to reminder the last chosen sorting (settings)
FR003		Allow to find other devices/users
	UR003a	Allow to find other device in a user-friendly way
FR004		Allow to see the restrictions (audio, video, remote) for each device
	UR004a	Group for the head device and a separated group for all the client devices. Eventually, group the client devices by type of connection
FR005		Allow to switch between a head user (player device) and a client user (receiver)
	UR005a	Allow to define a default startup view. Always start as a player device or as a receiver device
	UR005b	Allow to change if needed in a user-friendly way in the menu of the settings

Table 3.3: Functional and usability requirements - Part 1

FR	UR	Description
FR006		Allow to manage the restrictions for each device as a head user (distribution handler)
	UR006a	Allow to manage in a user-friendly way which device can hear some sound
	UR006b	Allow to do the same for the video track of a video
	UR006c	Allow to manage in a user-friendly way which device can have a remote controller and which device can have a download button to share a file on a user friendly way (only in home mode) (see FR008)
FR007		Allow to broadcast media content to different client users
	UR007a	Allow to do this in a user-friendly way with a maximum of five clicks
FR008		Allow to receive media content from the head user
	UR008a	Allow to receive content in a user-friendly way taking into account the restrictions
FR009		Allow to control media with a remote controller (if allowed)
	UR009a	Allow to have a user-friendly way controller and in Home mode with the ability to adapt it by creating and modifying different controllers (FR101)
FR010		Allow to have some granularity for video files by disabling the sound or the image of a video file
	UR010a	Allow to do this with FR006 and the related URs
FR011		Allow to send the sound of a media file to the speaker of the car (BT, cable)
	UR011a	Allow to do this when sound is activated on the head device with default output
FR012		Allow to have a remote controller with voice commands
	UR012a	Allow to do this with only one click to accept voice command

Table 3.4: Functional and usability requirements - Part 2

We also like to mention some URs that are not linked to a functional requirement. The first one is *learnability*. We want that all users can use the system without any form of training, therefore the system should be easy to learn. Other important URs are *consistency* and *attractiveness* . Our style guide must be followed on all the different OS to have a consistent look with an appealing layout and colour. We must, of course, implement a reliable application with a certain degree of performance so that we can interact with a least five different devices at the same time. *Efficiency* is the last requirement that we want to mention and that we can measure eventual with some questionnaire where we test the satisfaction of the target users.

3.6.3 Navigation Requirements

The navigation must be clear and must follow the mobile principle of Android, or target system where we will implement our POC applications.

3.7 User Object Model

In our application, we must store the information of the engine device with all their connected client devices. Figure 3.5 show the engine device, which is the `CrossDevice` entity with all their attributes. `DisplayName` will be the logical device name. `UniqueName` will be the generated technical device name from the framework that we will use which is explained in Chapter 4. `StartPlayerActivity`, `SortOrder` and `SortKey` are needed to optimise the usability of our application and can be changed in the settings of our application. `IsAudioEnabled`, `IsVideoEnabled` and `IsImageEnabled` are needed to know if we must show the media windows. These properties are managed by the Distribution Manager screen in our application. The `IpAddress` is needed so that the connected clients can find the media files of the engine device. Why this attribute is needed, is explained in Chapter 4.

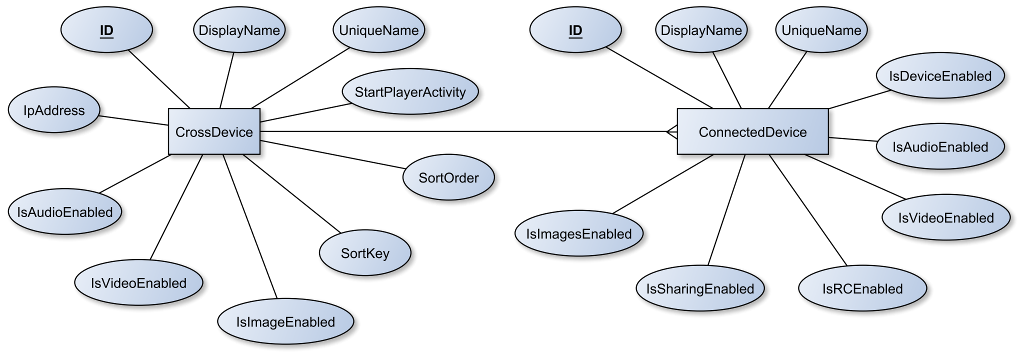


Figure 3.5: ER diagram for the engine device with the connected clients

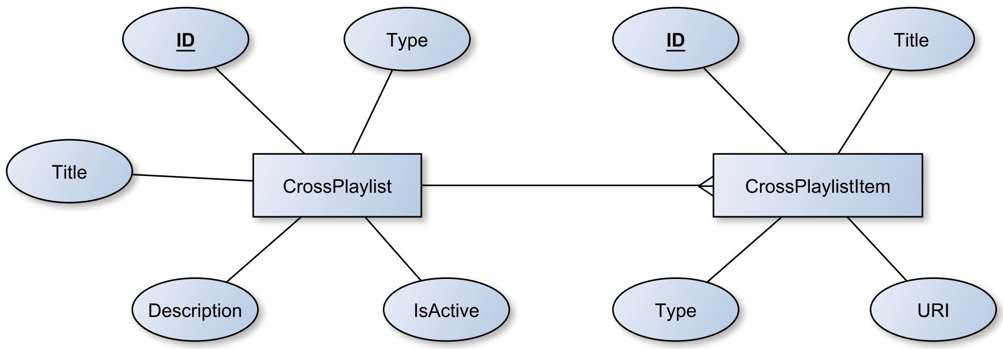


Figure 3.6: ERD for the engine device with the connected clients

Figure 3.6 shows the Entity Relationship diagram for the playlist functionality. A playlist belongs to a specific media `Type`: audio, video or images. And each playlist has their own `Title` and `Description`. With the attribute `IsActive`, we can delete temporary a certain playlist. In one playlist, we can have zero or multiple items with each their own `Title` and `URI` to specify where that media file can be found on the storage of the device. With the attribute `Type`, we could maybe mix audio and video files in one playlist. But, we will not allow this mix in our final implementation.

3.8 Style Guide

We have used the Material Design for Android[2]. We added a logo which is shown in Figure 3.7 and follows the colour style guide of Android as well. The logo represents the wireless interaction between devices.

3.9 Design

3.9.1 Paper Prototypes

Before starting with the paper prototypes, it was important to get acquainted with the Android Design Principles[3] and with the Android Language. Since Android is based on the Java language, it was important to start learning the basics of Java. That is why I started reading the book Head First Java [56]. After that, I started learning Android by writing some small test applications

[2]`https://developer.android.com/design/material/index.html`
[3]`https://developer.android.com/design/get-started/principles.html`

Figure 3.7: Logo of our POC application

in Android Studio, the tool to develop Android apps. These test applications have learned me some specific Android Material Components, like Floating Action Buttons, Cards, Navigations, Lists and the different layout possibilities. With this background, I have created my paper prototypes. These paper prototypes have gone through several cycles to get to the paper prototypes that are findable in the appendix A.1. We have focused our paper prototypes on the POC CrossWoW Auto, because that was the idea with the highest challenge and without knowing whether we could actually realise our idea.

These paper prototypes were still changed during implementation phase because it was logical and technically a better solution to adjust our first idea a little bit. We will explain why in the next Chapter. But this cycle of adaptation of the paper prototypes is a normal evolution in the design of an application.

On the next page, we show some paper prototypes examples. Figure 3.8 show our first design of our Welcome Screen. Figure 3.9 shows the first design of the list of all the local video files of the engine device. We can `play directly` or `Add to Playlist` each video item. Figure 3.10 demonstrate our first menu and settings screen. All these prototypes can be found in the appendix A.1.

In the next Chapter, where we explain our implementation, we will start by developing our Android Prototypes.

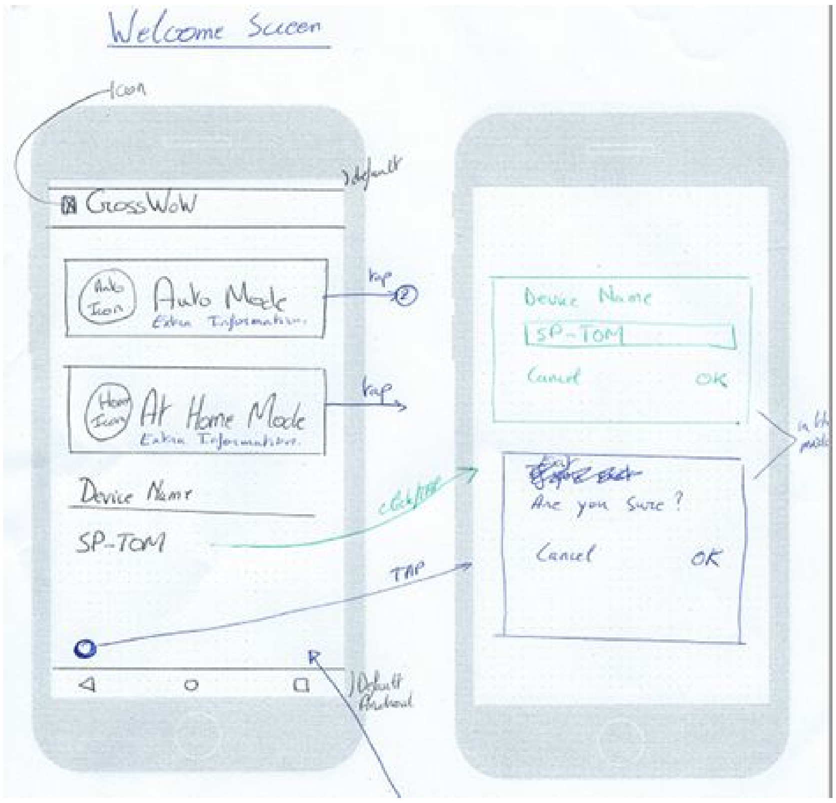


Figure 3.8: Our first welcome screen

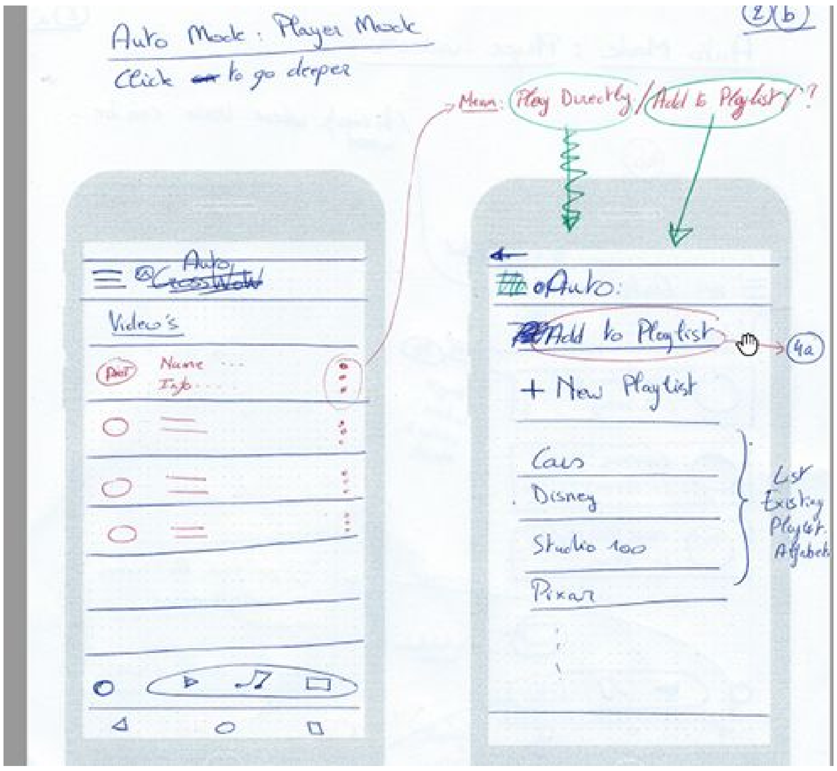


Figure 3.9: Player mode screen

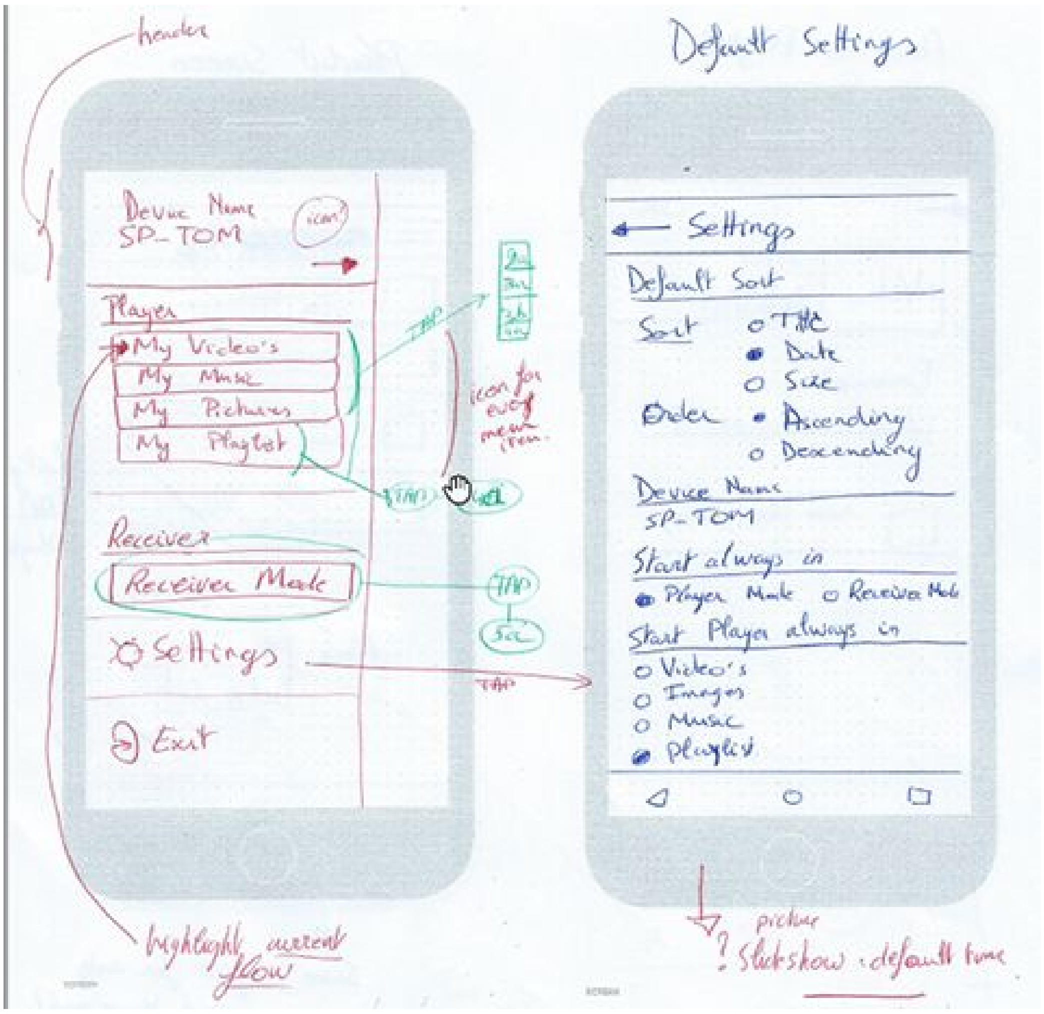


Figure 3.10: Menu and settings screen

4 Implementation

In the previous chapter, we have designed our first paper prototypes. Using these prototypes we go to the next phase of our design cycle, which is the implementation of the prototypes. We will separate our implementation into different sub-problems. Thereafter, we will delegate each sub-problem to its own analysis and implementation. The goal, to distribute a media file to different devices with some interactions, has multiple challenges where we must find a solution for: try to find other devices; send messages between them; communication technology; streaming; synchronising; transfers; granularity and a remote controller.

4.1 Android Prototyping

4.1.1 Why

We have ended the previous chapter with our Paper Prototypes, which is a variant of a throwaway prototyping or rapid prototyping. Hence, we have created a model that will not become part of our final delivered POC application. Instead of creating the screens in a software tool like for example Axure[1], we want to have some evolutionary prototyping or breadboard prototyping, where the main goal is to make a strong prototype that we can use

[1] `https://www.axure.com`

and refine in our target POC application. The benefit of this technique is that we can still make some little changes during the sub-analysis that we will make in our implementation phase. It is a more Agile approach where the solutions can evolve during the process of development. Because our target development will be Android, we will make these prototypes in Android. Hereby, it is also possible to understand better the corresponding tool Android Studio[2], which is the official IDE for Android, along with the different Android concepts.

On the next page, we show the results of two of our first Android prototypes. Figure 4.1a was our first Android Welcome Screen. We have modified this prototype in a later phase, because of a more technical and logical reason. The Receiver Mode which is visible in the menu of Figure 4.1b, will move to our Welcome Screen. Another proof that prototyping is indeed an iterative process.

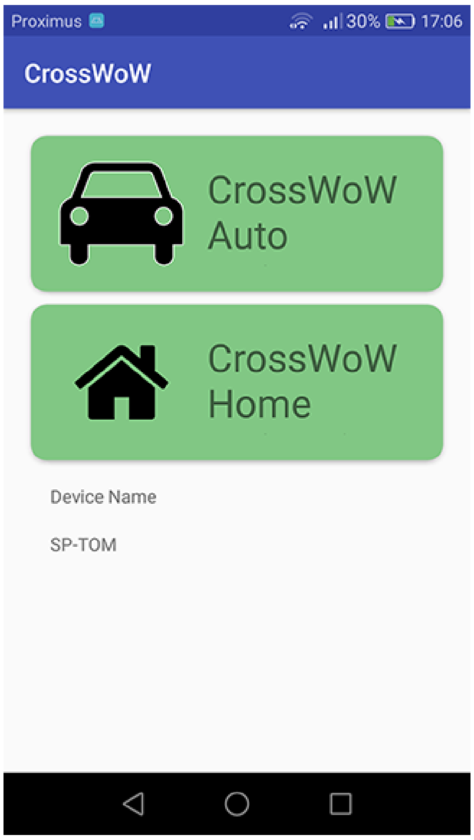
4.1.2 Obstacles and Solutions

During that iterative process, we have encountered already some problems. In our paper prototypes, we have foreseen big action buttons instead of the default Android buttons. After some research, we have used the Android Support Libraries[3] in our development projects for these features. This phase was interesting to learn some of the basics of Android, like the different layout possibilities, fragments, activities, lists, adapters and the Android manifest file. Some problems were solved by a simple adaptation of the manifest file.

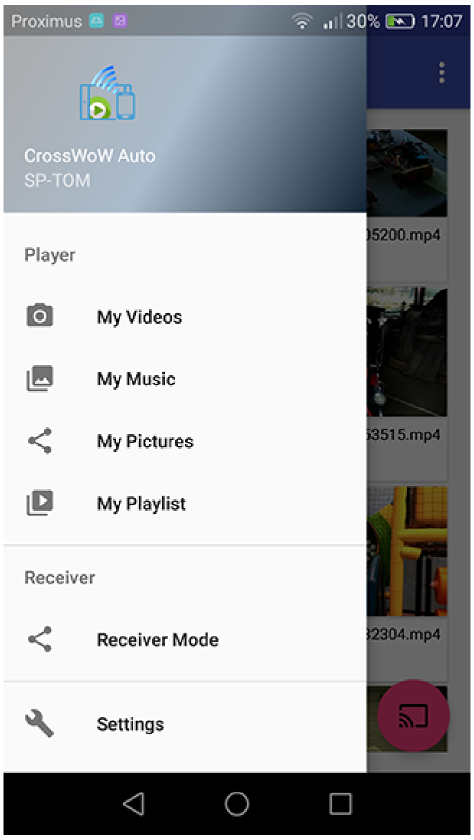
```
Extra used libraries:
compile 'com.android.support:appcompat-v7:25.1.0'
compile 'com.android.support:support-v4:25.1.0'
compile 'com.android.support:design:25.1.0'
compile 'com.android.support:recyclerview-v7:25.1.0'
compile 'com.android.support:cardview-v7:25.1.0'
```

[2]https://developer.android.com/studio/index.html

[3]https://developer.android.com/topic/libraries/support-library/setup.html



(a) Our first welcome screen



(b) Player mode screen

Figure 4.1: First screen of CrossWoW and CrossWoW Auto

4.2 Local Media Files

If we want to distribute some media files, we must also have access to them in order to show them in our application. To achieve this goal, we have first made several separated test applications, and we have implemented the most effective and visual solution in our POC application. In our design, we have foreseen three different types of media: video, audio and picture files. To limit the complexity of the set of all the different formats that there exist, we have filtered these files so that we only take MP4 and JPG files. The performance was another issue, but therefore we have used Glide[4], an open source image loading library for Android. Glide has helped us to make a list of media images that makes the scrolling of our list fast and smooth.

[4]https://github.com/bumptech/glide

Before a user can use certain system data and features, like for example the camera of a phone, Android requires that applications request a permission before they can use them. Depending on the Android version, the application can ask all the permissions at the installation time of the application. Therefore, the developer must mention all these permissions in the manifest file. But, from Android 6.0 (Marshmallow) certain permissions must be explicitly asked the users the first time they will need that permission. Hence, these permissions are granted at runtime. This is very similar to how iOS has worked for years. Users can feel safe that their camera or microphone will only be used if they explicit give their consent. Another runtime permission is to read the media files from the Android device, which we are doing in this phase.

```
Extra used library:
compile 'com.github.bumptech.glide:glide:3.7.0'
```

4.3 Device Pairing and Communications

4.3.1 Wireless Technology

Now that we have our Android prototyping and that we have access to our files, we can start to analyse how we can communicate between the different devices. In our POC application, we are sitting in a car. So, most of the cars have not yet a local area network, like we may have at home. In some of the new cars, there is indeed an optional expensive mobile data subscription possible that can share the internet connection with multiple devices. We want to meet everyone and so we look for a solution that can be used in all the cars.

If we look to the current wireless technology, we have Wi-Fi, Wi-Fi Direct, Bluetooth (BT), Near Field Communication (NFC) and Bluetooth Low Energy (BLE). Android O will have another technology, namely Neighbourhood Aware Networking (NAN) mode for Wi-Fi. NAN has the possibility to find other devices and communicate over Wi-Fi without an access point. NAN is a sort of a combination of Wi-Fi Direct and Nearby. NAN is also the abbreviation for Near-me Area Network or Neighbourhood Area Network, which focus is on the communication between wireless devices in close proximity. The difference with a Local Area Network (LAN) is that the device in a LAN network must share the broadcast domain. In a Near-me Area Network, the devices can belong to a different network. Hence, the communication can go from one LAN to another LAN via the internet. We have not investigated in the NAN communication because this is a technology that only works with Android and only with the very recent version of Android.

At this moment, BT is too slow to stream a video file to different devices. If we also want to support the interoperability between them with different operating systems in the future. BT is also not possible because Apple has their own version of BT. So, if we want a BT connection, we must investigate our research in BLE which can communicate between an iOS and an Android device. But, BLE is also too slow. We have also skipped NFC, because of the needed distance between two devices. With NFC we can establish a communication by bringing the devices within four cm of each other. NFC can be used to pair two devices or in contact-less payment systems. But for our POC application where the devices are at fixed positions in the car, it is not a useful protocol.

Hence, we will investigate our analysis for a Wi-Fi hotspot or use Wi-Fi Direct. We start first with Wi-Fi Direct because that seems to be the best solution to transfer data at long distances without connecting to a LAN network.

4.3.2 Wi-Fi Direct

We start by investigating in the-low level Android API's by consulting the Android developer website about Wi-Fi Direct[5]. Our first conclusion was immediate that it will be a tough solution to work with the Google APIs. But, we give it a change and start by testing the Wi-Fi Direct demo[6] from the Google Wi-Fi samples. The problem was that sometimes it takes a long time before our test devices have found each other and sometimes there was no problem.

Thereafter, we have searched some literature about Wi-Fi Direct and we have tested some frameworks. An interesting article[7] was found on the internet as part of the Thaliproject. The conclusion of that article is that they have done a lot of research about Wi-Fi Direct but that they gave it up after a time and used BLE or BT. Wi-Fi Direct Peer Discovery seems to work but it was not possible to send enough useful information around. Wi-Fi Direct Service Discovery, on the other hand, seems to be unreliable. Phones do not find each other especially if they are from different manufacturers.

[5]`https://developer.android.com/training/connect-devices-wirelessly/Wi-Fi-direct.html`

[6]`https://github.com/android/platform_development/tree/master/samples/WiFiDirectDemo`

[7]`http://thaliproject.org/androidWirelessIssues/`

Hence, Wi-Fi Direct seems not to be a good solution. To be sure, we have tested two other frameworks that use Wi-Fi Direct: Wi-Fi Buddy[8] and Salut[9]. Wi-Fi Buddy is a Wi-Fi Direct functionality that uses the android APIs. Salut is also a wrapper around the Wi-Fi Direct APIs. For Salut, we found some limit support and apparently there is also a Wi-Fi connection needed on the host. Wi-Fi buddy seems to be a recent research and the readme file contains enough information on how to use Wi-Fi Buddy. But, again if we test the demo application Wi-Fi-direct-tester[10], we find the same problems as we have encountered with the sample of Google. Our test devices are coming from different manufacturers with different Android versions.

Another disadvantage that we have found later in our research is that Apple has also a proprietary version of Wi-Fi Direct. So, we can not interoperate between an iOS device and an Android device with Wi-Fi Direct.

4.3.3 Wi-Fi Hotspot

Hence, turning one device into a Wi-Fi Access point or hotspot seems to be the preferred way if we want to transfer a lot of data over a wireless communication. The only disadvantage of this approach is that one user has to manually setup a Wi-Fi Access Point and the other users must manually change their Wi-Fi endpoint to the hotspot. So, we lose some usability with this approach. A big advantage is that we can interoperate between devices with different OS.

4.3.4 Conclusion

Hence, the best solution that we have found now is using a Wi-Fi Access Point. So, we must investigate how we can easily find and communicate between the other devices over a Wi-Fi connection. After some further research and testing some samples, we will use the Alljoyn library for the communication between our different devices. The decision is made on a positive testing of one of their samples, a lot of online documentation and some support on different blog sites. The Alljoyn library is used by LG and Microsoft and has now an alliance, called Allseen alliances. Which is a consortium of companies like Qualcomm, Cisco, Panasonic, Sharp, LG, and HTC. The other advantage of Alljoyn is that they can work over cross OS and cross hardware.

[8] `https://github.com/Crash-Test-Buddies/Wi-Fi-Buddy`
[9] `https://github.com/markrjr/Salut`
[10] `https://github.com/Crash-Test-Buddies/Wi-Fi-direct-tester`

4.4 Alljoyn Library

4.4.1 What is the Alljoyn Library

The open source API framework Alljoyn[11] is made for the IoT, but we will use it for our cross-device interactions to send a message between the different devices. AllJoyn does not only support communication over Wi-Fi but also over Bluetooth. In the past, they have also supported Wi-Fi Direct. Wi-Fi Direct has become deprecated at a certain moment and now it is completely removed from the library. For the pairing, if all the devices are connecting to the same Access Point, then they are also connected to the same area. When the setup is correct then we can find the other devices who want to be found.

Figure 4.2 show a typical example where the freezer door is still open and consequently it will send an alert message to another device to keep the user informed. The freezer is in the Alljoyn framework a thin client because it has no UI. It uses an Alljoyn router to “onboard” the local wireless network. Because the freezer has no display, it needs to inform the user of events. Therefore, it runs an Alljoyn client and uses another Alljoyn device with a display for event notification.

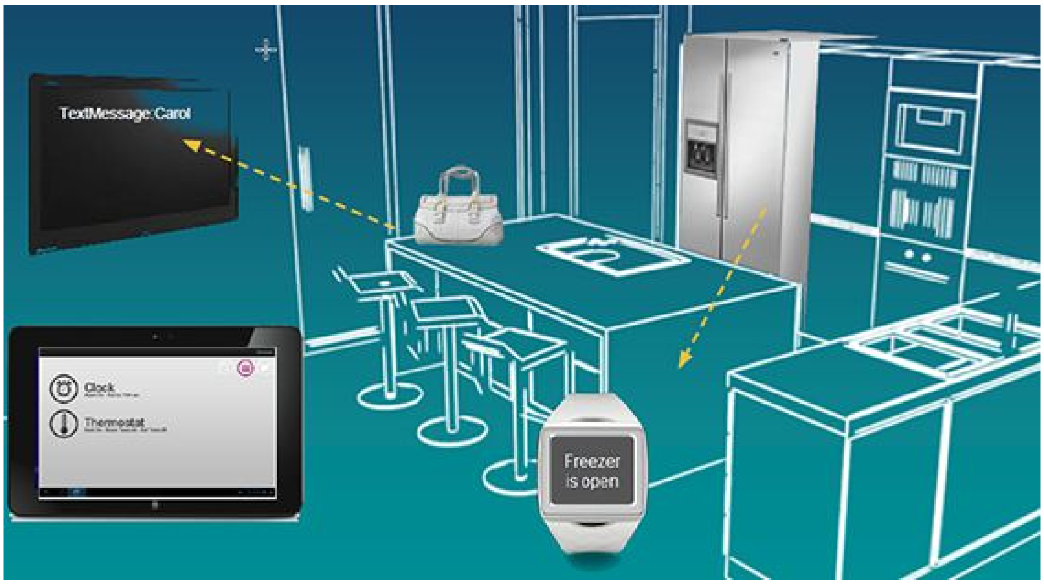


Figure 4.2: Freezer Alljoyn example

[11]https://allseenalliance.org/framework

4.4.2 Architecture

Before we can use the library or made a wrapper for it, we must understand some concepts about the Alljoyn framework. In the next section, we will talk more technical about the wrapper that we will write.

Figure 4.3 shows the high-level architecture of Alljoyn, where we have `Routing Nodes` and `Leaf Nodes`. Leaf nodes can only connect to routing nodes and routing nodes can connect to other routing nodes. The applications that expose the APIs are services. The consumer of these services are called clients and the nodes that expose and consume are called peers.

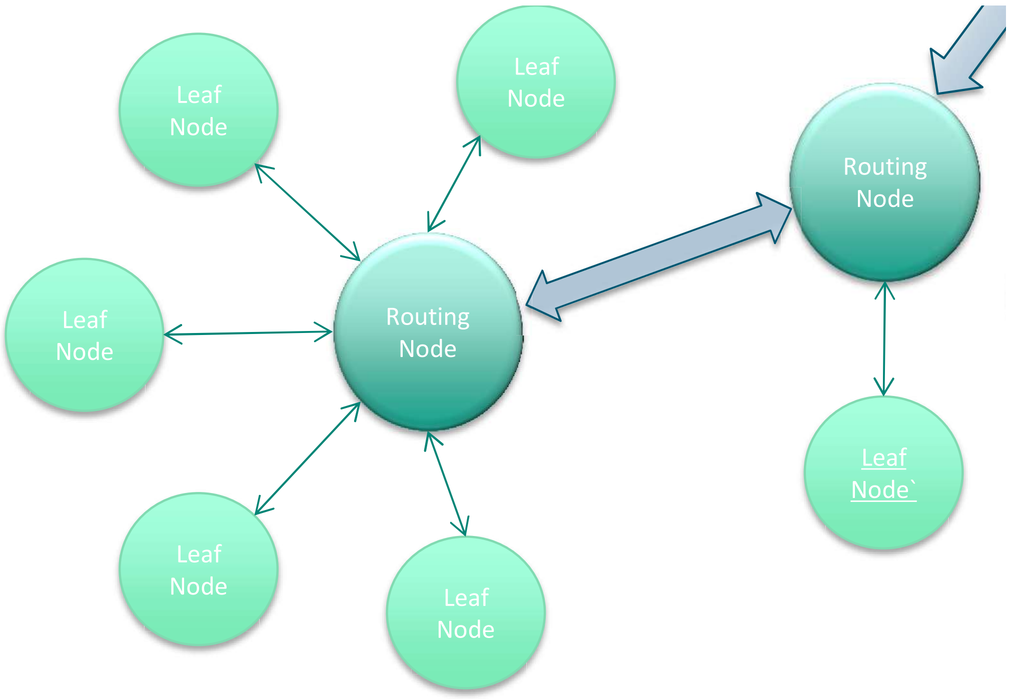


Figure 4.3: Routing and leaf nodes

An important concept is `Advertising`. `Advertising` is the process to let other applications know we are on the network and they can connect with us. The interface that can be used between the different devices is called the `About feature`, which is the service level discovering. `Discovery`, let us find other applications that are in the same area and hence nearby. When a group of applications is connected, they have all the same `Session` which is allowing them to exchange data.

The Alljoyn Bus is a way to move messages around in the distributed system. Alljoyn is using D-Bus[12] to exchange the data which can run over any medium: Wi-Fi, Wi-Fi Direct, Ethernet, plc and Bluetooth. D-Bus is a message bus system to talk to one another. An important term is marshaling, which is the process of transforming a value into a sequence of bytes (wire format). Unmarshalling is the other way, from the wire format to a specific type. The D-Bus specification[13] are important to know if there are some problems by sending a message between devices.

The device that will distribute the media files in our POC application is acting as a server, we will call it the engine router. This engine router is needed to distribute the media files to all of their client routers. Each router must make a connection to the bus and let the bus know which interface the router will support (`About::Announce`). Thereafter, each router must setup some listeners to capture certain events that are coming from the bus. The engine router is making the session and does a
`BusAttachement::AdvertiseName` so that the other routers can find that specific engine router. The client routers can find the engine router with a
`BusAttachement::FindAdvertiseName(prefix)`. The prefix is needed for some filtering.

A client router can connect to a specific session on the bus, by calling `BusAttachement::BindSession`. When a session is created, the Bus is extended. All peers are notified on join and leave events and can interact via their APIs or with multicast events. Be aware that sessions must be created between the client and the engine routers before they can interact with each other. All routers are defined with some options, but the engine router will define the name of the bus and the port of the session. Each node gets a specific unique name. Hence, when there is a join between the engine and a client router, we know the name of the bus, the unique name of the client and the used session id.

```
Engine router: [options, bus name, session port]
Client router: [options, unique name, session id]
Join: [options, bus name, unique name, session id]
```

In the About interface we can define three types of members: *methods*, *signals* and *properties*. *Methods* are like in regular programming languages, a function that can be called and return a certain result. *Signals* is the way to

[12]`https://www.freedesktop.org/wiki/Software/dbus/`
[13]`https://dbus.freedesktop.org/doc/dbus-specification.html`

broadcast, multicast or point to point asynchronous event notifications. It is a one to many message that is shipped to all the connected devices where no feedback is asked. *Properties* can be accessed by getter and setter methods. We will use signals to broadcast messages from the engine router to their client routers and we will use methods to call or trigger some functions from the client routers to the engine router.

In order to effectively use the library in multiple Android applications, it seems to be useful to write a personal wrapper around the library. We can then import this Android library into our project as a module dependency.

4.5 CrossDevice Wrapper

4.5.1 Installation of Alljoyn

For the wrapper we have created an Android Library that we have called the CrossDeviceWrapper. First we must install the Alljoyn library in our library. Therefore, we must go through some steps. First, we have to create two folders: "/app/src/main/**jniLibs**" and "/app/src/main/jniLibs/**armeabi**". Under the folder jniLibs, we must put the "`alljoyn.jar`" file. Under the folder armeabi, we must put the "`liballjoyn_java.so`" file. Both of them can be found on the Alljoyn website. In the dependencies window of our project we add a link to the "`alljoyn.jar`" file. This will add an additional compile line to the manifest line of the project. In the file where we will use the Alljoyn code, we have to add the snippet shown in Listing 4.1. Now, we can use the Alljoyn SDK in our wrapper.

```
static {
 System.loadLibrary("alljoyn\_java");
}
```

Listing 4.1: Load the native alljoyn java library

4.5.2 Evolution and Structure

Creating a wrapper for the Alljoyn library was a huge challenge and it was an iterating process that has take some time to reach the end result. Our first step was to write the necessary code where we can make an engine with a specific bus name and create the coding for the advertising of the engine. After that, we have done the same for a client router, so that the client can

Method signature	Type
(1) void sendMediaOnSignal(CrossDevMedia media)	S
(2) void sendEngineActionOnSignal(String action)	S
(3) void sendEngineSeekActionOnSignal(DeviceSeekAction seekAction)	S
(4) void sendConfigChangedOnSignal()	S
(5) CrossDevConfig getDevConfig(DeviceName deviceName)	M
(6) void sendDevInfo(DeviceName deviceName)	M
(7) void sendDevState(DeviceState deviceState)	M
(8) void sendDevAction(DeviceAction deviceAction)	M

Table 4.1: BusInterface: BusSignals (S) and BusMethods (M)

find the engine. The next step was to send a simple signal message with only a string as parameter to a client. And in the last phase, we have make the necessary models and the signals and methods that we need for our POC application.

The Java package name is `be.vub.crossdeviceservice`. In our library we have created 3 sections: `device`, `end` and `util`. The `util` section is used for our constants and logging helper functions. In the `device` section, we have the necessary coding for the About interface. Hence we have created the Alljoyn `@BusInterface` with the name `be.vub.crossdeviceservice.Device.Interfaces.IDeviceMediaInterface` and the corresponding implementation with the necessary `BusSignals` and `BusMethods`. Table 4.1 gives an overview of all the signals and methods that we have foreseen in our wrapper. The signals are going from the engine to all the connected clients. The methods are going from one specific client to the engine. Because we have not found a way to pass multiple parameters, we have created specific models that can be passed or return as one parameter for the signals and the methods. An overview of the different models can be found in Table 4.2. The model `DeviceName` for example contains the properties `displayName` and `uniqueName`. UniqueName is how the device is know in Alljoyn and the displayName is a more logical name that the user can give to their own device. The logical name is much more useful to know to which device we are communicating with. For example, the user specify the displayName = “Tablet Tom”, so we know now the user is talking with Tom’s tablet. The unique name is an eight alphanumeric string and Alljoyn will create a mapping of that unique name with the IP or Bluetooth address of the device.

CrossDevConfig	
DeviceName name	To identify the device
boolean isDeviceEnabled	Is the device enabled to receive
boolean isAudioEnabled	Can the device play audio or not
boolean isVideoEnabled	Can the device play video or not
boolean isRemoteContrEnabled	Can the device have a remote controller
boolean isSharingEnabled	Can the device download the media file
boolean isImagesEnabled	Can the device view an image or not
CrossDevMedia	
DeviceName engineName	To identify the engine device
String engineDomain	The domain URI of the engine device
int currentWindowPosition	Current video or audio position
MediaItem[] mediaItems	Playlist of media items
DeviceAction	
DeviceName clientName	To identify the client device
String clientAction	Remote Controller action
DeviceName	
String displayName	Logical device name
String uniqueName	Unique name of Alljoyn
DeviceSeekAction	
int windowIndex	Index for the playlist
long positionMs	Position of the media file
DeviceState	
DeviceName clientName	To identify the client device
String clientState	Client state
MediaItem	
String name	To identify the client device
String type	Type of media (MP4/JPG/IMG)
String URI	Location of the media on the engine device

Table 4.2: The different created models that are used in the BusInterface

In the `end` section, we find the necessary coding for talking with our library and the communication with the Alljoyn library. We have created two different service endpoints, one for the engine (`EngineEndService`) and one for the different clients (`ClientEndService`). The communication with the Alljoyn library can be found for the engine in the `EngineCommMgr` and for the clients in the `ClientCommMgr`. This is for the communication to Alljoyn. The other way is done with some listeners. We have created some interfaces that can be used by the application to listen to some events that are coming from the bus system. Table 4.3 gives a brief overview of them and we will explain most of them in the following section.

Both managers implement the `BusListener` and the `SessionListener`. The `BusListener` is used to keep track of the different engine routers and their advertise name. The `SessionListener` is used to keep track of the different Members in a session.

4.5.3 Installation of our Wrapper

The only thing that must be done to use our wrapper in other Android application is to import our .aar file. This can easily be done by choosing "New Module" and thereafter the "Import AAR package" option. As a last step there must be determined a dependency to it that will update the gradle file with an extra line to compile the `project(":CrossDeviceWrapper")`.

In the POC application, we have created a separated folder structure that we have called the `"crossdeviceservice"`. In that package we have made the same distinguish between the coding for the engine (`CrossEngineService.java`) and the client (`CrossClientService.java`). Both of them have to code the init, stop and start of their service and must load the Alljoyn library. Beside of those two service files, we have made some custom interfaces for the listeners in the application and some helper functions. The engine service has to implement some circle methods, bus signals and the engine listeners from Table 4.3. The implementation of the interface `IAlljoynMsgListener` contains code to know if the service is running or not and it will save the Alljoyn unique name in our DB. The implementation of the interface `IBusDataEngineListener` contains several functions. The function `RecvDevConfig` will return the config information of a specific device that is stored in our DB. The function `RecvDevInfo` on the other hand will save a new connected client device into our DB with some default options. A new device will not yet have access to show video, audio, images or the remote controller. The engine device must first enable these options in a specific manager screen. The two other functions are more like

IAdviceListener	Engine
void getAdvStatus(boolean status)	
IAlljoynMsgListener	Client + Engine
void onSucc(String msg, int msgCode)	
void onFail(AlljoynErr err)	
IBusDataClientListener	Client
boolean RecvURIBusData(String URI)	
boolean RecvMediaBusData(CrossDevMedia media)	
boolean RecvEngineActionBusData(String string)	
boolean RecvEngineSeekActionBusData(DeviceSeekAction seekAction)	
boolean RecvConfigChangedBusData()	
IBusDataEngineListener	Engine
CrossDevConfig RecvDevConfig(DeviceName deviceName)	
boolean RecvDevInfo(DeviceName deviceName)	
boolean RecvDevState(DeviceState deviceState)	
boolean RecvDevAction(DeviceAction deviceAction)	
IGetServiceListener	Client
void foundAddService(String name, short port)	
void foundRemoveService(String name, short port)	
IJoinCircleListener	Client
void getJoinStatus(boolean status)	
IJoinListener	Engine
void addJoiner(long sessionId, String joinerName)	
void delJoiner(long sessionId, String joinerName)	
void getSessionStatus(String serviceName, boolean sessionStatus)	
IServiceFoundListener	Client
void addServiceFound(ServiceFound service)	
void removeServiceFound(ServiceFound service)	
ISessionStatusListener	Client
void getSessionStatus(String serviceName, boolean status)	
void addJoiner(long sessionId, String joinerName)	
void delJoiner(long sessionId, String joinerName)	

Table 4.3: The list of listeners to be implemented by the POC application

a door hatch to pass the information. The function `RecvDevAction` aims to get different commands coming from one of the clients' remote controller. These commands are specified by the enum `ActionPlayer` that is specified in our wrapper. The possible commando's are: PLAY, PAUSE, PREV, NEXT, REW, FFWD, NEXT_IMG, PREV_IMG, RELEASE and SYNC. The `IJoinListener` serves to know if some client devices have added or have leaved our circle. If the client device is disconnected, we will remove that device from our DB. The engine service has also public methods that will call the four signals of Table 4.1 and one method to the Alljoyn Bus to get the current connected clients.

The client service must also implement some circle methods, bus methods and the client listeners from Table 4.3. Like for the engine service, the implementation of the interface `IAlljoynMsgListener` has code to know if the service is running or not and it will save the Alljoyn unique name in our DB. The implementation of all the other interfaces are more a door hatch to pass the information. The methods of the interface `IBusDataClientListener` have different goals. Function `RecvMediaBusData` will treat the different media files and put them for example in a array list to play them later on. Function `RecvEngineActionBusData` must contain code to execute the action that is send by the engine. Function `RecvEngineSeekActionBusData` will jump to a specific file and a specific position in that file. Function `RecvConfigChangedBusData` will notice all the already connected clients that the engine has changed some restrictions for the different devices, so the code must contain a call to get the current config settings. The implementation of the interface `IServiceFoundListener` shows a dynamic list of the active engine routers. Each client can connect to one of them. The implementation of the interface `ISessionStatusListener` implements code to act when a client is joining or leaving a circle. In our application we have used the `addJoiner` function to send the current client device information to the engine.

We have created three service listeners. The `IEngineSourceListener` is corresponding with the `IBusDataEngineListener` listener of our wrapper. We have split the `IBusDataClientListener` into two listeners because we have two different activities on the client. One activity (`IClientMediaListener`) to make the connection and receive the media files, and one activity (`IClientActionListener`) to receive the different actions for the media list. The client service has also public methods that will call the four methods of Table 4.1.

4.5.4 Obstacles and Solutions

First of all, we had to understand the Alljoyn library. Fortunately, there exist a lot of online documentation and support. The simple but well-structured examples have also helped us a lot to understand the FW even better. In our setup, there are a lot of development files involved to send a signal from the engine to all of their clients or a method from the client to the engine. On top of that, there are also a lot of listeners who listen to events coming from the Alljoyn bus. Hereby, testing and finding bugs was not always an easy task. For example it was not possible to send an object where one of the properties had a null value. We then got an Alljoyn java error which is showed in Listing 4.2:

```
%E/ALLJOYN_JAVA: 6.313 ****** ERROR ALLJOYN_JAVA lepDisp1_2
        .../jni/alljoyn_java.cc:11445 | 0x0001
%W/System.err: org.alljoyn.bus.MarshalBusException: cannot marshal null into '(ss)'
```

Listing 4.2: Passing NULL values

Luckily, when we have found the second rule in our logging. We knew that there was something wrong with the marshalling of a null value. Afterwards, this seems logical. But, it is not so obvious to think about that problem immediately, because an object whose properties contain null values usually do not give any problems if we call other methods for example.
Another challenge for logging and testing was the multi threading issues. In our solution we have made some variables volatile to indicate that a variable's value can be modified by different threads. By declaring a volatile Java variable, the value of this variable will never be cached thread-locally, all reads and writes will go straight to the main memory. Another problem we encountered was when we send a signal to all the clients and the moment that the clients receive the signal, they will send a method to the engine. At that moment we get the Alljoyn error of Listing 4.3:

```
ER_BUS_BLOCKING_CALL_NOT_ALLOWED
```

Listing 4.3: Callback error

The solution was here to call the
`BusAttachement::EnableConcurrentCallbacks()`. The
`EnableConcurrentCallbacks` tells the code to make the method call on a different thread than the current Alljoyn thread. Otherwise, our code will deadlock and never return taking up one of the threads from the thread pool indefinitely. So, when there is a call to another AllJoyn method inside these handlers then we should call `EnableConcurrentCallbacks()` before making the call. Hence, we create a public method for `EnableConcurrentCallbacks`

in the `ClientCommMgr` that will execute the `EnableConcurrentCallbacks()` on the Alljoyn Bus. Once the wrapper was finished, it was easier to understand the communication via our wrapper. So, our wrapper can help developers to make interactions between devices easier, but a disadvantage is that the wrapper is now still specific made for our POC application.

4.5.5 Use of our Wrapper

On the next pages, we will show some snippets on how we have used our wrapper. In our application, we have created two services that will communicate with our wrapper. In Listing 4.4 and Listing 4.5 we describe on how we have init the player mode by starting our `CrossEngineService.java` that will call the endpoint `EngineEndService.java` of our CrossDeviceWrapper. In receiver mode, we do almost the same thing, but now we will communicate with the endpoint `ClientEndService.java` and the corresponding `ClientCommMgr.java` that will communicate directly with the Alljoyn library. We have defined some listeners that will listen to all the existing engines. In our application, we will call the `joinCircle` function to add the receiver device to our circle which is show in Listing 4.6. All devices that are connected to the Alljoyn bus can now communicate with each other.

```
//Snippet Code of our Player Main Activity
public static final CrossEngineService localEngineService = new CrossEngineService();

//Create EngineService
CreateCircle(devName);

private void CreateCircle(String devName) {
  mModelName = devName;
   ...
  if (localEngineService.isBound()) {
    Toast.makeText(AutoPlayerMainActivity.this, "new circle!", Toast.LENGTH_LONG).show();

    // called in another thread
    Thread stopThread = new Thread(new Runnable() {
      @Override
      public void run() {
        localEngineService.stopService(AutoPlayerMainActivity.this);
      }
    }, "stopthread");
    stopThread.start();
  }

  Thread startTread = new Thread(new Runnable() {
    @Override
    public void run() {
      while (localEngineService.isServiceRunning()) {
        try {
          Thread.sleep(1000);
          Log.i(TAG, "waiting...");
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
      boolean ans =
      localEngineService.InitService(AutoPlayerMainActivity.this, false, mModelName);
      Log.i(TAG, "ans: " + ans);
      if (!ans) {
        // notify
        return;
      }

      mAdvListener = new AutoPlayerMainActivity.AdvListener();
      localEngineService.setAdvListener(mAdvListener);

      Log.i(TAG, "startService: " + localEngineService.isBound());
      localEngineService.startService();
    }

  }, "startThread");
  startTread.start();
  Log.i(TAG, "Bound: " + localEngineService.isBound());
}
```

Listing 4.4: Init engine service

```
// InitService in CrossEngineService
public boolean InitService(Context context, boolean isClient, String circleName) {

  Log.i(TAG, "startService");

  if (mIsBound) {
    Log.w(TAG, "already start service");
    return false;
  }

  mContext = context;
  Intent intent = null;
  intent = new Intent(context, EngineEndService.class);  //engine
  Bundle bundle = new Bundle();

  bundle.putString(EngineCommMgr.class.getName(), "server");
  if (null == circleName) {
    Log.e(TAG, "service role, but not set circlename");

    return false;
  }
  mCircleName = circleName;

  bundle.putString(AlljoynConst.SERVICE_NAME_KEY, AD_SERVICE_NAME);
  bundle.putShort(AlljoynConst.SERVICE_PORT_KEY, AD_SERVICE_PORT);
  bundle.putString(AlljoynConst.SERVICE_OBJPATH_KEY, BUS_OBJ_PATH);
  bundle.putString(AlljoynConst.CIRCLE_NAME_KEY, mCircleName);

  intent.putExtras(bundle);

  boolean ans = context.bindService(intent, mConnection, Context.BIND_AUTO_CREATE);

  if (!ans) {
    Log.e(TAG, "bindservice fail");
    return false;
  }

  mIsBound = true;

  return true;
}

//In the wrapper, we init the communication with Alljoyn
if (null == engineCommMgr) {
  engineCommMgr = new EngineCommMgr(this.getApplicationContext(), this.getPackageName());
  engineCommMgr.initAjEventHandler();
}
```

Listing 4.5: Init engine service

```
//CrossClientService
public static final CrossClientService localClientService = new CrossClientService();

//possible calls to our wrapper that will communicate with Alljoyn
mEndService.getAjCommMgr().setAlljoynMsgListener(mAlljoynMsgListener);
mEndService.getAjCommMgr().setBusDataListener(mBusDataListener);
mEndService.getAjCommMgr().setServiceFoundListener(mServiceFoundListener);
mEndService.getAjCommMgr().getSessionStatusListener(mSessionStatusListener);
mEndService.getAjCommMgr().connect();
//or
mEndService.getAjCommMgr().disconnect();


//join the circle that is created by the engine device
 localClientService . joinCircle (name, port);

//in the wrapper
private BusAttachment mBus = null; //connect: message bus
mBus.joinSession(name, contactPort, sessionId, sessionOpts, new SessionListener() {
 //coding for sessionLost / sessionMemberAdded / sessionMemberRemoved
);
```

Listing 4.6: Init client service

We will explain two different examples where we will use three devices in each scenario. One device will start the application in Player Mode, the two other devices will start the application in Receiver Mode. In order to easier explain the scenario and the corresponding diagram, we will give a name to each device. “P” is the name for the device in Player Mode. “R1” and “R2” are the names for the two devices that have started the Receiver Mode.

In the first scenario, we distribute our media list to all the clients. Figure 4.4 shows the communications between all the different entities. The gray boxes (EngineCommMgr, Alljoyn BUS and ClientCommMgr) are part of our wrapper. The other entities are part of our application code. In the appendix, we show the corresponding coding for this scenario. In Listing A.1, we describe the first part where we send our media list to the Alljoyn Bus. In Listing A.2, we register the `BusSignalHandler` that will catch the signal of P and send the list to all the clients, R1 and R2 in our scenario.

Where the previous scenario shows a signal example, we will now describe the next scenario where one client R1 clicks on the pause button. Figure 4.5 shows the corresponding scenario with the different communications between the entities. Listing A.3 shows the method call that is followed by a signal which is show in Listing A.4.

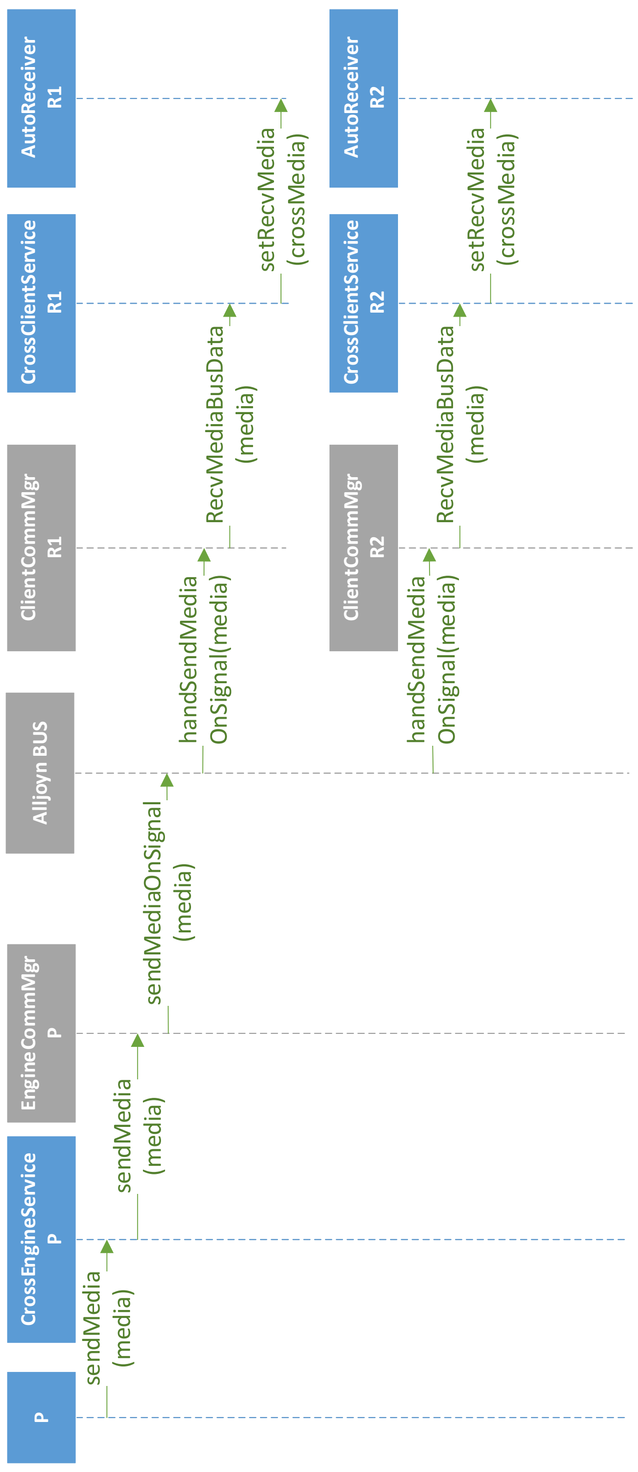


Figure 4.4: Scenario 1: distribute media list to R1 and R1

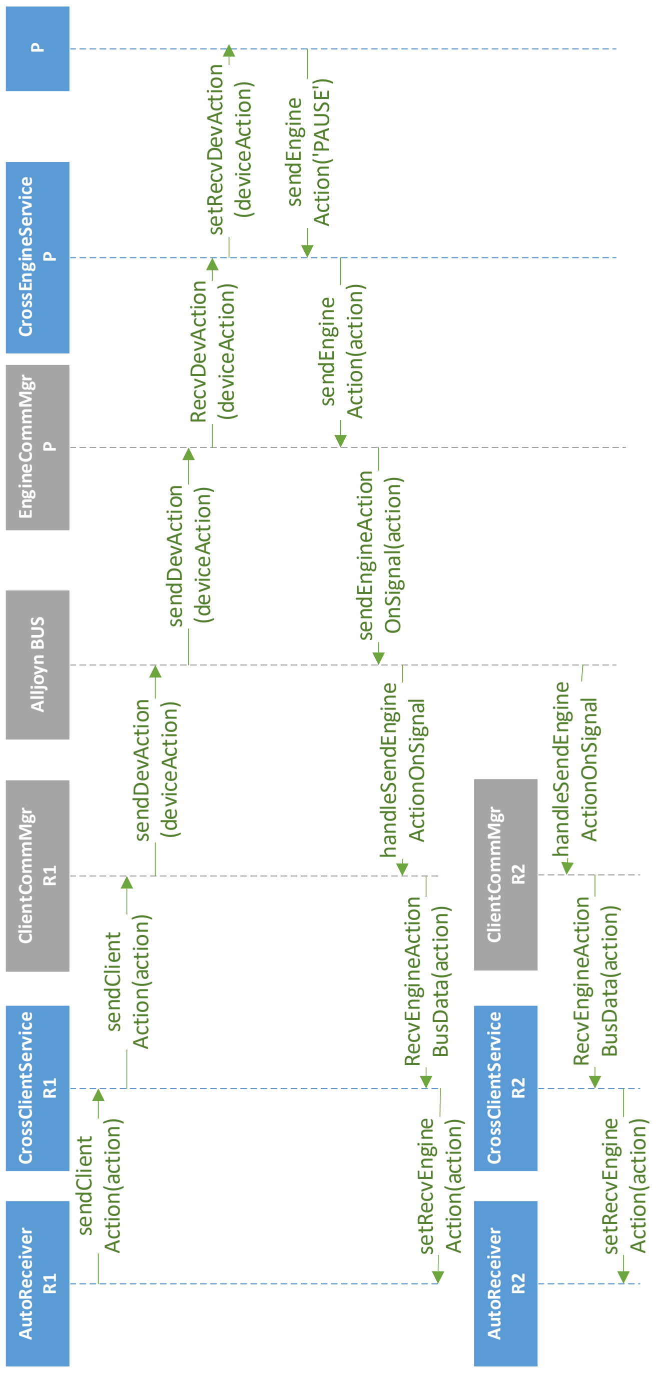

P
CrossEngineService P
EngineCommMgr P
Alljoyn BUS
ClientCommMgr R1
CrossClientService R1
AutoReceiver R1
ClientCommMgr R2
CrossClientService R2
AutoReceiver R2
sendClient Action(action)
sendClient Action(action)
sendDevAction (deviceAction)
sendDevAction (deviceAction)
RecvDevAction (deviceAction)
setRecvDevAction (deviceAction)
sendEngine Action('PAUSE')
sendEngine Action(action)
sendEngineAction OnSignal(action)
handleSendEngine ActionOnSignal
handleSendEngine ActionOnSignal
RecvEngineAction BusData(action)
RecvEngineAction BusData(action)
setRecvEngine Action(action)
setRecvEngine Action(action)

Figure 4.5: Scenario 2: PAUSE action from client R1

4.6 Multimedia Granularity

As granularity in our application, we want to split the sound and image of a video file so that we can manage what we want or not want to play on our target devices. When looking for a solution, we get a lot of specific audio and video vocabulary. Our first idea was to extract the audio or the image of a video file. After investigation, the most popular tool therefore is FFmpeg[14], “a complete cross-platform solution to record, convert and stream audio and video”. Or JAVE[15], a wrapper of FFmpeg that can for example encode a video into an audio file. And there exist a lot of other open source github projects around FFmpeg. But, we want to select a file and stream it directly without waiting before the video file is first encoded into an audio file. Another issue is that we want to make the decision when a video is already playing on the target devices. Hence, extracting in advance is not a good solution.

Another proposed solution is to route the audio from the target device, back to the main device. This may a good solution if we cast a video on our TV and we want to hear the sound on our mobile device. But, it is also not the right solution for our problem. Hence, we must find a solution at render time of our media file. Our first idea was to mute the audio on the target device only for a specific application. That may work if we always want to see the image of the video file on the target devices and that is not the case. But, luckily there seems to be a possibility to select certain tracks in a video file and since Android API level 16, there is an `android.media.MediaExtractor`[16], which facilitates extraction of demuxed, typically encoded, media data from a data source. Hence, we could do some track selection in video files. And in the next section, where we have searched for a streaming option, we have found the perfect solution for this problem.

4.7 Sending and Streaming Media

We have now a wrapper that can send signals to all the connected devices and send methods to the engine device. However, this way of working is not suitable for sending or streaming large video files. That is why we must further investigate to find a solution for this.

[14]`https://ffmpeg.org`

[15]`http://www.sauronsoftware.it/projects/jave/`

[16]`https://developer.android.com/reference/android/media/MediaExtractor.html`

4.7.1 Webserver

To stream a file, another device must reach the source file. In certain OS, we could work with shared folder, but this is not possible on an Android device. It is also not an option to send first the full (large) file, and afterwards start playing it. Certainly, if we want to work with a playlist of media files, this is not a possible solution. Hence, the only possible solution is to create a web or media server for the engine device. After some little research, we have selected NanoHttpd[17], because it is an open-source very light webserver that can be embedded in applications. When the webserver is active, the source file is for example "http://192.168.43.89:8080/storage/0123-4567/DCIM/Camera/VID_20170108_183812.mp4".

In Listing 4.7, we show some snippets on how we have used NanoHttpd to define an IP address and start our webserver.

```
//Add NanoHttpd as a gradle dependency
compile 'com.nanohttpd:nanohttpd-webserver:2.2.0'

//snippet to start our webserver
String ipdevice;
FileServer mediaServer;

DefineIpAddress();
StartFileServer();

private void DefineIpAddress() {
 WifiManager wifiManager =
      (WifiManager)getApplicationContext().getSystemService(WIFI_SERVICE);
 WifiInfo wifiInfo = wifiManager.getConnectionInfo();
 int ipAddress = wifiInfo.getIpAddress();
 ipdevice = String.format("http://%d.%d.%d.%d:8080", (ipAddress & 0xff), (ipAddress >> 8 & 0xff),
      (ipAddress >> 16 & 0xff), (ipAddress >> 24 & 0xff));
 Log.d(TAG, "IP device: " + ipdevice);
}

private void StartFileServer() {
 mediaServer = new FileServer();
 try {
  mediaServer.start();
 } catch (IOException ioe) {
  Log.d(TAG, "The server could not start.");
 }
}
```

Listing 4.7: Installation and use of NanoHttpd

[17] http://nanohttpd.org

4.7.2 Exoplayer

After some research for streaming video files in a P2P network, WebRTC seems to be the perfect solution. RTC stands for Real Time Communication (without plugins). WebRTC applications still need to do several things: (1) get streaming audio or video; (2) get network information (IP addresses, ports) to exchange with other WebRTC clients to enable connection; (3) coordinate some communication for errors and close connections; (4) exchange information about media resolution and codec and (5) communicate the streaming audio or video. Normally WebRTC is only designed for web browsers, but there exists some custom frameworks for Android like WebRTC for Android[18]. But, there is still a need for a WebView in the layout.

Moreover, there exists also an open source project from Google for streaming media files, namely Exoplayer[19]. On the internet, we have found a demo application and a developer guide[20]. It is the application level media player for Android, that is now also used for Youtube. It is an alternative for the Android MediaPlayer API, where we can modify and extend the player. Hence, we can modify the buttons of the remote controller and capture the events when one of the buttons are pressed.

To select a specific track, we have to use the `MappingTrackSelector`. With this selector we enabled or disabled a certain track. We show some snippets of the code in Listing 4.8 where we disabled the audio and the image track and some possible actions for the player. Exoplayer is an open source project and has a lot of online documentation and support.

```
import com.google.android.exoplayer2.*;
private SimpleExoPlayer player;
TrackSelection.Factory videoTrackSelectionFactory =
  new AdaptiveVideoTrackSelection.Factory(BANDWIDTH_METER);
trackSelector = new DefaultTrackSelector(videoTrackSelectionFactory);
//disable video track
trackSelector.setRendererDisabled(0, false);
trackSelector.clearSelectionOverrides(0);
//disable audio track
trackSelector.setRendererDisabled(1, false);
trackSelector.clearSelectionOverrides(1);
//Exoplayer v2 - start player
player.setPlayWhenReady(true);
//pause player
player.setPlayWhenReady(false);
//go to specific media file and position in that file
//used for PREV + NEXT + REW + FFWD + SYNC
player.seekTo(seekAction.windowIndex, seekAction.positionMs);
```

Listing 4.8: Google Exoplayer v2 - used for audio and video

[18] `https://github.com/SDkie/Webrtc-for-Android`
[19] `https://github.com/SDkie/Webrtc-for-Android`
[20] `https://developer.android.com/guide/topics/media/exoplayer.html`

Exoplayer v2 will only work for Android devices, but VLC media player[21] is also a free and open source cross-platform multimedia player that can execute the same functionalities to select a track of a video file. Exoplayer can only play audio and video files. So, in code 4.9, we demonstrate how to parse an image to an `ImageView`.

```
private ImageView imageView;

imageView.setImageURI(Uri.parse(mediaURI_List.get(mediaPosition)));
```

Listing 4.9: ImageView used to display images

4.8 DB Structure

To store data, Android has an SQLiteDatabase which has methods to do the CRUD (Create, Read, Update, Delete) operations and other common database management tasks. We have searched for an easy Object-Relational Mapper (ORM) tool, because we want to have an automatic mapping of our object-oriented models to a relation DB. Hence, we have to deal less with persistence-related programming because the ORM tool will generated the required SQL statements behind the scene and we do not have to write a single SQL statement. We first hit Sugar[22] but a one to many relationship was not that easy and therefore finally we have chosen ActiveAndroid[23] which is also available on github[24]. We describe some snippets in Listing 4.10 that shows how easy it is to use. In Listing 4.11, we show some snippets for the CRUD operations from our application code.

[21]https://www.videolan.org/vlc/
[22]https://github.com/satyan/sugar
[23]http://www.activeandroid.com/
[24]https://github.com/pardom/ActiveAndroid

```
//creation of the tables
@Table(name = "CrossDevices")
public class CrossDevice extends Model {
 //id is automatically generated
 @Column(name = "DisplayName")
 public String displayName; //also CircleName
 @Column(name = "UniqueName", index = true)
 public String uniqueName;
 @Column(name = "IpAddress")
 // all the other colunms
 //Relationship − belong to one CrossDevice
 //one to many
 // This method is optional, does not affect the foreign key creation.
 public List<ConnectedDevice> items() {
  return getMany(ConnectedDevice.class, "CrossDevice");
 }

 //some constructors
 public CrossDevice() {
  super();
 }
 public CrossDevice(String displayName, String uniqueName) {
  super();
  this.displayName = displayName;
  this.uniqueName = uniqueName;
  this.isAudioEnabled = false;
  // ...
  this.startPlayerActivity  = 0;
 }

 //selection
 public static List<CrossDevice>getAllCrossDevices(){
  return new Select().from(CrossDevice.class).execute();
 }
}
@Table(name = "ConnectedDevices")
public class ConnectedDevice extends Model {
 //id is automatically generated
 //all the other columns can be found in de coding

 //Relationship − belong to one CrossDevice
 @Column(name = "CrossDevice")
 public CrossDevice crossDevice;

 //some constructors

 //Select all
 public static List<ConnectedDevice> getAllConnectedDevices(){
  return new Select().from(ConnectedDevice.class).execute();
 }

 //Select one specific connected device based on uniqueName
 public static ConnectedDevice getConnectedDevice(String uniqueName) {
  return new Select()
  .from(ConnectedDevice.class)
  .where("uniqueName = ?", uniqueName)
  .executeSingle();
 }
}
```

Listing 4.10: Mapping Object-Oriented Models with Tables

```
//Some CRUD operations examples
//Create + Update
//if not exist, insert device record
device = new CrossDevice(devName, "");
device.save(); //insert + update

//Read
CrossDevice engineDevice = CrossDevice.getAllCrossDevices().get(0);

//Delete
new Delete().from(ConnectedDevice.class).where("uniqueName = ?", joinerName).execute();
```

Listing 4.11: CRUD operations

4.9 Used Libraries

Below, we mention all the extra libraries that we have used in our application.

```
compile 'com.android.support:appcompat-v7:25.1.0'
compile 'com.android.support:support-v4:25.1.0'
compile 'com.android.support:design:25.1.0'
compile 'com.android.support:recyclerview-v7:25.1.0'
compile 'com.android.support:cardview-v7:25.1.0'

compile 'com.github.bumptech.glide:glide:3.7.0'

compile project(":CrossDeviceWrapper")

compile 'com.nanohttpd:nanohttpd-webserver:2.2.0'
compile 'com.google.android.exoplayer:exoplayer:r2.2.0'
compile 'com.michaelpardo:activeandroid:3.1.0-SNAPSHOT'
```

4.10 Architecture

Figure 4.6 shows an overview of all the most important components that we have used in our `CrossWoW` application. We have the `CrossWoW Auto` application that will use our `CrossDeviceWrapper`. Our wrapper will manage all the communications with the `AlljoynLibrary`. In the application, we have created two services: `CrossEngineService` and CrossClientService. These will handle all the communications with our wrapper. Each engine device needs a file server. Therefore we have used the `NanoHttpd` library. The `Exoplayer v2` library is used in each device for playing our media content. The last component that is visible on our schema is the `ActiveAndroid` library that we have used for adding some usability functionality and the creation of our playlists.

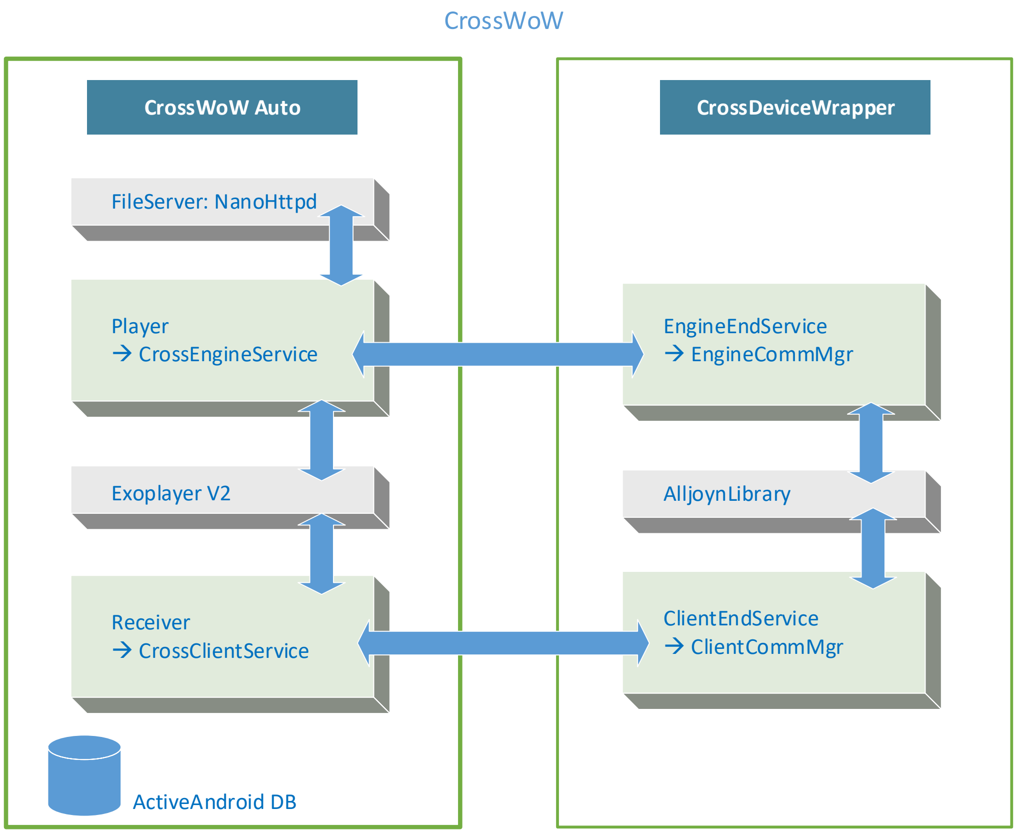


Figure 4.6: High Level Overview of CrossWoW

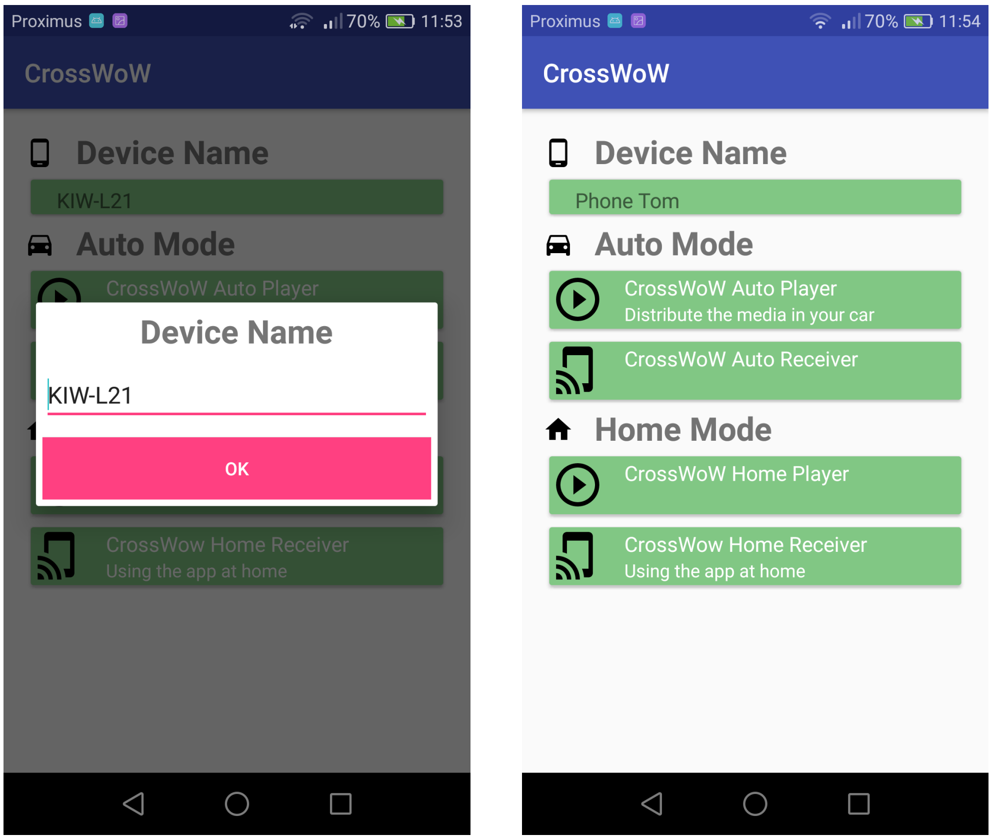
5 Results

In the previous chapter, we have described how we have implemented our CrossWoW Auto application. However, we have not yet mentioned how our tool can be used and what the benefits are. We will also mention how three children with different ages (13, 10 and 3 years) experience the application. At the end of this chapter, we will mention our second application CrossWoW Home, who intends to group the common code of the two applications to give a possible hint or proposition for a future library that we will propose in the next chapter. Hereby, we will also mention some shortcomings in our common code and some future design guidelines.

5.1 CrossWoW Auto

5.1.1 Particular Scenario

The application is written so that no additional manual is required. The different steps are very logical and are following the Android design principles. Below, we will speak about two different modes: the player mode and the receiver mode. The player mode is the engine device in our system and the receiver mode is made for the different clients.



(a) First time opening the application (b) Welcom screen

Figure 5.1: Welcome screen with the possibility to change the logical name

I will explain with some screen shots how a family that goes on a holiday trip with their car, can use our application. There is first some preparation needed. So, before leaving for travel, the father must put the necessary content on the external storage of their phone or tablet. How he does this, is not important for the use of our application which will always read all MP4, audio and JPG files from the external storage. Afterward, he must, of course, install and open our application. The first time every user who opens our application will see Figure 5.1a for determining a more logical name for that specific device instead of the default Android name. That logical name will help later on to know for which device they decide to set up the different restrictions. Thereafter or the next time the user starts CrossWoW, they will always see the screen of Figure 5.1b, which shows an option to choose between the player mode or the receiver mode. Starting the device in player mode will automatically start the engine mode in the background so that the device can be found by other devices. The father can already make some video, audio and images playlists. Figure 5.2 demonstrates how he can select and make these playlists.

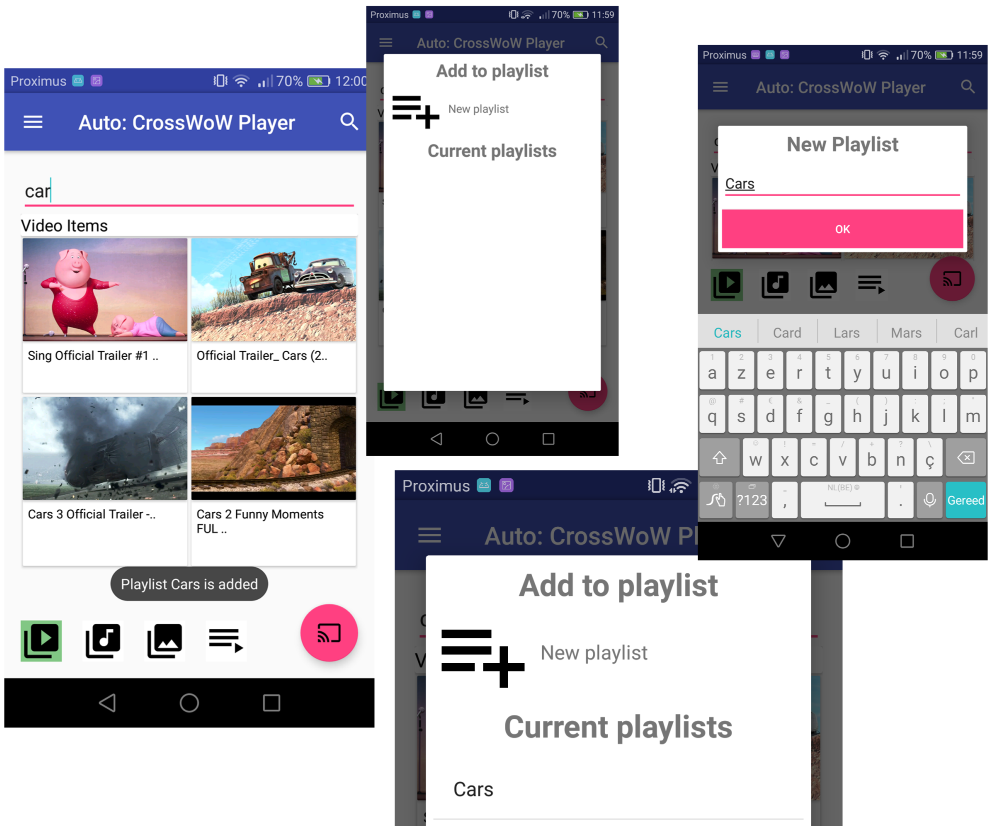
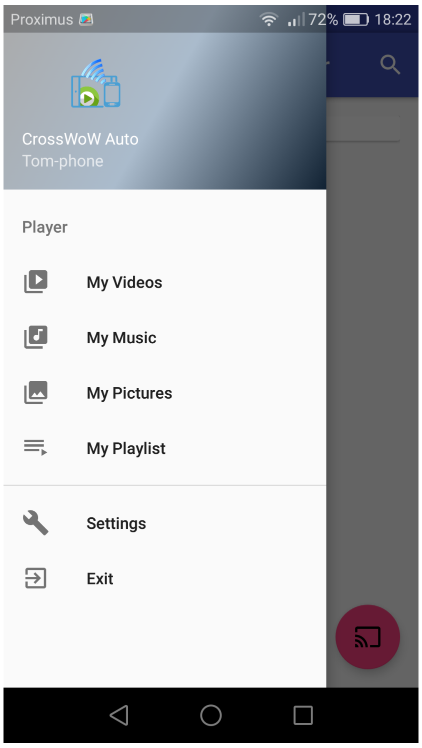


Figure 5.2: The different playlist screens to add items to new or existing playlists

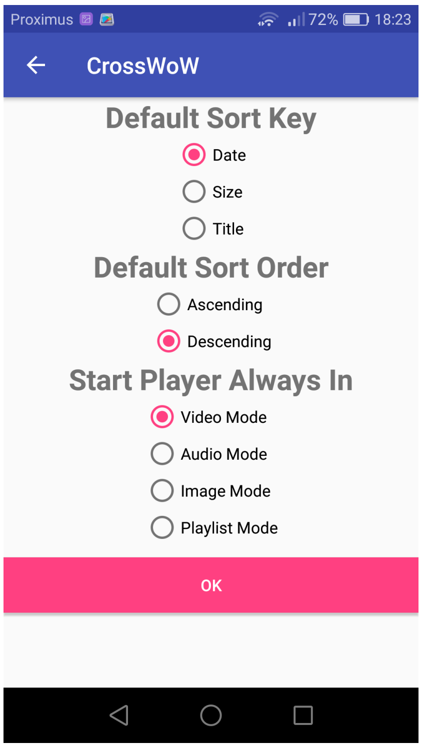
There is also a possibility to change in the settings screen the start activity and the sorting order for the items in the list, which is shown in Figure 5.3. Hence, the father can, for example, select the playlists as the default startup activity and order the items in descending order of the creation date.

CrossWoW must also be installed on all the client devices which will be the tablets of the different kids in the car. The father or the kids can give a logical name to each device. All the devices can be installed in the car on departure day. Holders exist here, like for example the picture 3.2 of Chapter 3 is showing.

The father connects the phone or tablet with a cable to the radio of the car or makes a Bluetooth connection with the car. Afterward, he makes a hotspot on one device and all the other devices must connect to it. Hereby, they are all on the same wireless network. Figure 5.4 shows the first screen



(a) Menu in Player Mode



Default Sort Key

[✓ Date] Size Title

Default Sort Order

Ascending [✓ Descending]

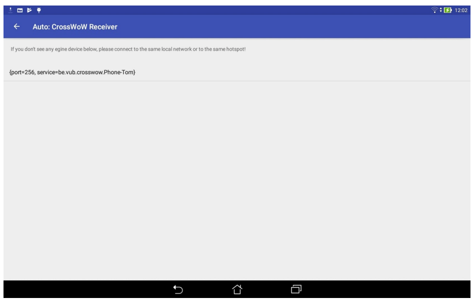
Start Player Always In

[✓ Video Mode] Audio Mode Image Mode Playlist Mode

(b) Settings screen in Player Mode

Figure 5.3: To improve the usability

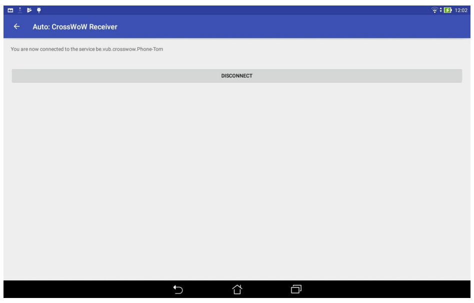
when each device opens the receiver mode. They will get a dynamic list of all the player devices that are connected to the same wireless network. A manual click is still needed to connect to a specific engine. We will explain why in one of the following sections.



If you don't see any egine device below, please connect to the same local network or to the same hotspot!

{port=256, service=be.vub.crosswow.Phone-Tom}

(a) List of engine devices

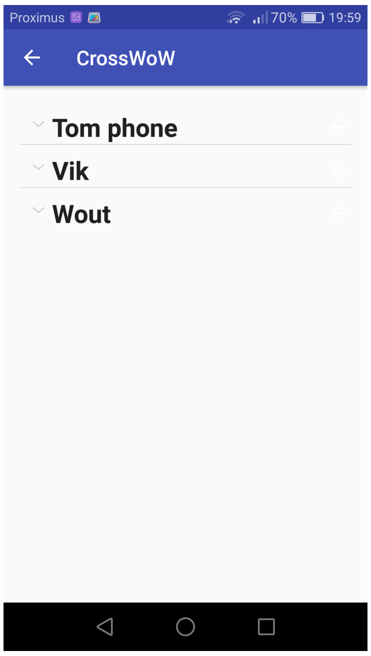


You are now connected to the service be.vub.crosswow.Phone-Tom

(b) After connection to one of them

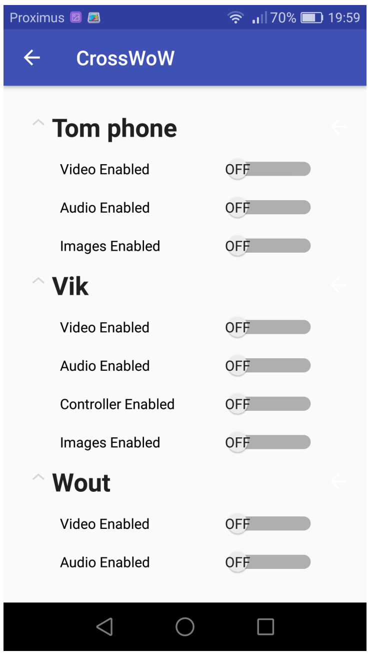
Figure 5.4: First screen in Receiver Mode

Now, the passenger can distribute the desired video for example to all the connected devices. Figure 5.5 shows the CrossWoW distribution screen where the different restrictions can be specified in. The first device is always the engine device. All the other devices are the connected clients. For the engine device, it is possible to determine which parts of the media file we want to render. This allows us to dynamically determine the granularity of the media file. For each client device, we can further determine if this device is enabled and whether it may have a remote controller.



Tom phone
Vik
Wout

(a) By default

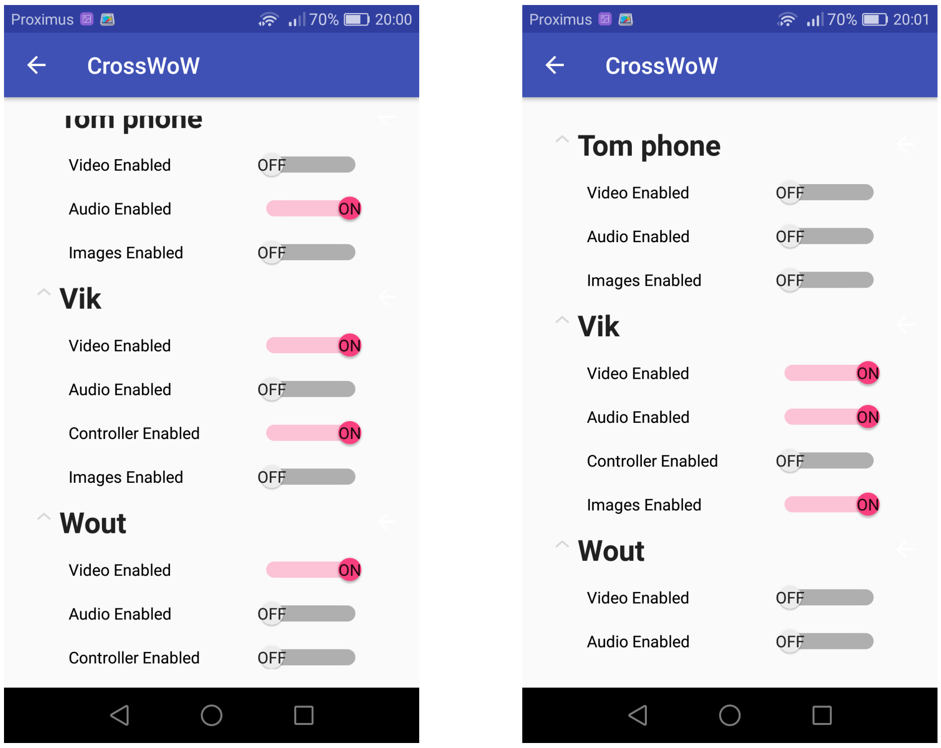


Tom phone
Video Enabled OFF
Audio Enabled OFF
Images Enabled OFF
Vik
Video Enabled OFF
Audio Enabled OFF
Controller Enabled OFF
Images Enabled OFF
Wout
Video Enabled OFF
Audio Enabled OFF

(b) Possible to open each device

Figure 5.5: Distribution manager in Player Mode

We will mention two different possibilities, but there exist many other possible setups. Figure 5.6a shows the settings where the engine device will only render the audio of the video file and all the clients will only render the image of that video file. One child (Vik) gets the remote controller. Another setup is showed in Figure 5.6b where only one child wants to see a specific video and we gave the full control to that device except for the remote controller. Hence, it is not really needed to render the audio and the image of the video on the engine device. The child may listen through headphones so that he does not interfere with the rest.


Proximus 70% 20:00
CrossWoW

Tom phone
Video Enabled OFF
Audio Enabled ON
Images Enabled OFF

Vik
Video Enabled ON
Audio Enabled OFF
Controller Enabled ON
Images Enabled OFF

Wout
Video Enabled ON
Audio Enabled OFF
Controller Enabled OFF

(a) Audio only on main device

Proximus 70% 20:01
CrossWoW

Tom phone
Video Enabled OFF
Audio Enabled OFF
Images Enabled OFF

Vik
Video Enabled ON
Audio Enabled ON
Controller Enabled OFF
Images Enabled ON

Wout
Video Enabled OFF
Audio Enabled OFF

(b) Only 1 device can see the video

Figure 5.6: Possible setups

5.1.2 Benefits

One of our challenges was to deal with the problem of interconnection and communication. For the users of our application all the necessary coding for combining these capabilities disappear in the background. They are not aware of the different technical ids like an IP address. They only work with the logical names of the different devices.

All devices are connected to the same network in a kind of a P2P network. There is no problem if one connected device is lost or added. Our application is robust for such changes and will continue to work. For an engine device, it is a little bit different when there are already some clients connected to this device. They will no longer be able to use media files from the engine and they will all have to connect with another engine. On the other hand, it is not a problem if a client device decides to become an engine device by choosing the player mode in the application instead of the receiver mode. All the devices can be both input and output devices depending on the settings in the CrossWoW distribution screen. The engine device is always an input device, but it can also render the media files. A client device can also be both. For example, it can contain only a remote controller to give input to all the other devices. But, it can also contain only the rendering of a media file without a remote controller.

Another benefit of our application is that our application can work in a space without constraints. The XDIs can be used anywhere because no internet connection or additional sensors are needed. If we look to the classification of Sanctorum and Signer [51], we can mention that our application has also some granularity. The user can decide for playing only the image of the video file and not the audio or vice versa or even so playing the video as a whole. Hence, our application can be used anywhere with some fine granularity. Together with that granularity, we have implemented a Multi-Device Environment where one task span more than one device. We can see our remote controller as a small single UI that can be distributed to one or more clients. Not only can we change the granularity and the distribution of the remote controller before we distribute the media file, but even in the middle of playback. Hence, the distribution of the UI is dynamic thanks to our distribution screen. All these mentioned benefits improve the usability of our application. We have not found a better solution in the Android Play Store that can execute the same scenario.

In addition to the benefits to the users, we have an advantage if we want to make similar applications in the future. We can reuse our current CrossDeviceService, which is a wrapper of the Alljoyn library, to find and communicate with the other devices. Hence, the advantage is that there is less code needed for future similar applications and that the development can happen much faster. In the next section, we will make a similar application to find some other similarities.

5.1.3 Evaluation

A family of two adults and three children have tested the CrossWoW Auto application for their auto trip from Belgium to France. Their first conclusion was a big satisfaction of our work because the purpose of the application was achieved. The distribution screen was very easy to use. They have tested the application with a cable to the radio and everything was in sync with each other. If this was not the case, they could put everything in sync with the sync option on the screen that broadcast the same time to all devices. There is still a sync problem when the engine is connected to a separate Bluetooth speaker. All images of the video were in sync with each other but the sound via the Bluetooth speaker came too late. The only solution that we could see here is an extra screen where we can define some offset with a certain range. Hence, with that option, we can play the sound via the Bluetooth speaker three seconds earlier, which is the time to make it all in sync again.

One missing usability was the option to easily put the media content on the player device. In order to increase its usefulness and not to work with cables or SD cards, it may be an option to investigate how to put the shared media on a hard drive which can make also a Wi-Fi hotspot. Another advantage of working with a Wi-Fi hard drive is that we have more memory to put the different media files on it. This is now fairly limited.

Another feedback is that the resend option was not used. This option was created to send the media list again to all the clients, especially some new clients. But, the settings of these clients must be first enabled in the distribution screen. And, by closing the distribution screen, we will resend the media file to all the peers, and after two seconds we will send the current seek time of the video file. In most of the cases, the user of the engine device will first define all the restrictions for each device and afterward sending the media list to all the devices. But, the possibility exists to hook on the current time in a movie that is already rendered on other devices.

There was one case where we have to use the resend button. That was the case when we power off the screen of the engine device. In the current coding, we stop playing and close the activity of all the clients. This is something that we can improve in the coding of our application. It is not an issue of our wrapper.

As for awareness, in the CrossWoW Auto application, it is not needed to know where a specific device is located. Because the logical name will contain for example "Tablet Wout". Hence, the user knows that "Tablet Wout" is the device of Wout who sits behind the passenger for example in the car.

Even we see some optimisations that are not currently provided in the application but may increase their usability. In the car, the client must manually connect to the engine. This can maybe optimised if we know that there is always only one engine. We can then make an automatic connection. Or, save the default engine so that the next time, we can make a direct connection with the default engine. A gesture-based binding like explained in Chapter 2 is less convenient because our devices depend on their holders. Making a binding by clicking on the logical name of the engine seems for this application the best solution. At this moment, the application will not remember the settings of the previous connections. If this is a usability requirement, then we must keep this based on the logical name of the device. However, this is not a uniquely defined name. The user can always put the logical name of another device. Hereby, there arise also some security questions. We could not do this on the unique name that is generated from the Alljoyn library, because it will always generate a new id, even for the same device. If really needed, we could, however, save some preferences into the cloud when an internet connection is available, because this is not a requirement in our application. We need then some cloud login and bind our device with that login account. A login account may be useful for keeping certain information on the server for future commercial purposes. It will also augment the usability, but we must be aware of some privacy issues.

Hence, we have created the CrossWoW Auto application, where initially we did not know yet if we could implement our ideal solution, with success. In the next section, we will further investigate which code is suitable for a possible future framework. We will do this by making a second similar application where we are going to look what the common code is for both applications.

5.2 CrossWoW Home

Our second POC application that we have also described in Chapter 3, has the same possibilities but it is now connected with the local wireless network at home, so with a possible internet connection. In principle, we can offer more functionalities for this application like for example filtering or using smaller or larger items in the lists or searching also on a Network Attached Storage (NAS) server for media files. In the ideal solution, we have also mentioned creating a custom controller. However, this is not the reason for creating this application. The goal for creating CrossWoW Home was to detect common code that can, later on, moved to a possible future XDI library. Ater creation of this application, we come to find to the next determination.

In our common code, we find three sections. The first one is our CrossDeviceService that will communicate with our CrossDeviceWrapper. This is also the most important part of our common code and must definitely be provided in a possible future library that we will discuss in the next chapter. The second part is the coding for the Android DB access. This is common code for our application, but it is not really code that will help us to improve cross-device interactions. The last common part in our code is the file server, for which we have used the NanoHttpd library. It is useful coding for our Android device, but it is not a good idea to put these coding in the core library that we propose in the next chapter. We can say the same about the Exoplayer library that we use.

6 Conclusion

In this last Chapter, we will first propose some suggestions for an investigation in a future XDI library together with some design guidelines. Hereby, we will base our suggestions on the common code that we have detected at the end of the previous chapter together with some more ideas and possible improvements. We will conclude this thesis with some conclusions.

6.1 Proposed Library and Future Work

Based on the common code that we have found, we propose a similar library like the CrossDeviceWrapper, but with more dynamic possibilities. Our library is now built for specific media applications. Instead of only supporting media files, the future library must find a solution to communicate for example contacts (address book of our device) between different phones. There exist a sample for the Alljoyn library where contacts can be transferred from one device to another device. One of our ideas is to have some authoring tool where we can combine the components that we need. Hence, a more component-based approach to make the library more dynamic. We can then make the library also extensible to other functionalities, like for example domotics.

In Chapter 2, we have mentioned three phases that are required for effective communication. These were detecting, tracking (or awareness) and communicating (or transferring information). In our library, we have implemented tracking and data communication. For the tracking phase, we have not foreseen something. Hence, our library is missing some know-how to be aware of nearby devices. We will leave this to a future investigation to review this further. But, what may help is that our system can broadcast messages to all the connected devices. The Alljoyn library that we use, call this feature a signal. Hence, the system can forward location or specific sensor data to all the other devices to implement some kind of awareness.

Something that should also be present in a possible library is security. A disadvantage of working with a web server is that everyone who can intercept the URI of our media file can access our media file. Even if they are not connected to the engine router. The Alljoyn system provides already a built in security framework for applications to authenticate each other and send encrypted data between them. Their and other systems can be further investigated in future research to have some secure communication channel. If we want that our application can interoperate between different OS and different devices, we have to provide the same CrossDeviceWrapper for Apple and Windows Devices. Alljoyn has the possibility to work with these OS.

6.1.1 Design Guidelines

With the experience of making our CrossWoW application and CrossDeviceWrapper, we propose some design guidelines. We are convinced that these guidelines could help other developers to develop a library for the prototyping of cross-device applications.

> ***Independence of Network Infrastructure:*** It is important to create a library that is not depending on one specific technology. In our solution, the communication with the Alljoyn library is done with a unique name. Behind the scene, Alljoyn will connect that unique name with the Bluetooth address or IP address of that device. So, if there is coming another connection in the future, we can connect that same unique name behind the scenes with the new address. The advantage is that the logic of the applications that use the library will still work and the developers should not engage with network addresses. Hence, the library must be written to work with different network infrastructures and must do the hard work to *detect*, *connect* and *communicate* with the other devices.

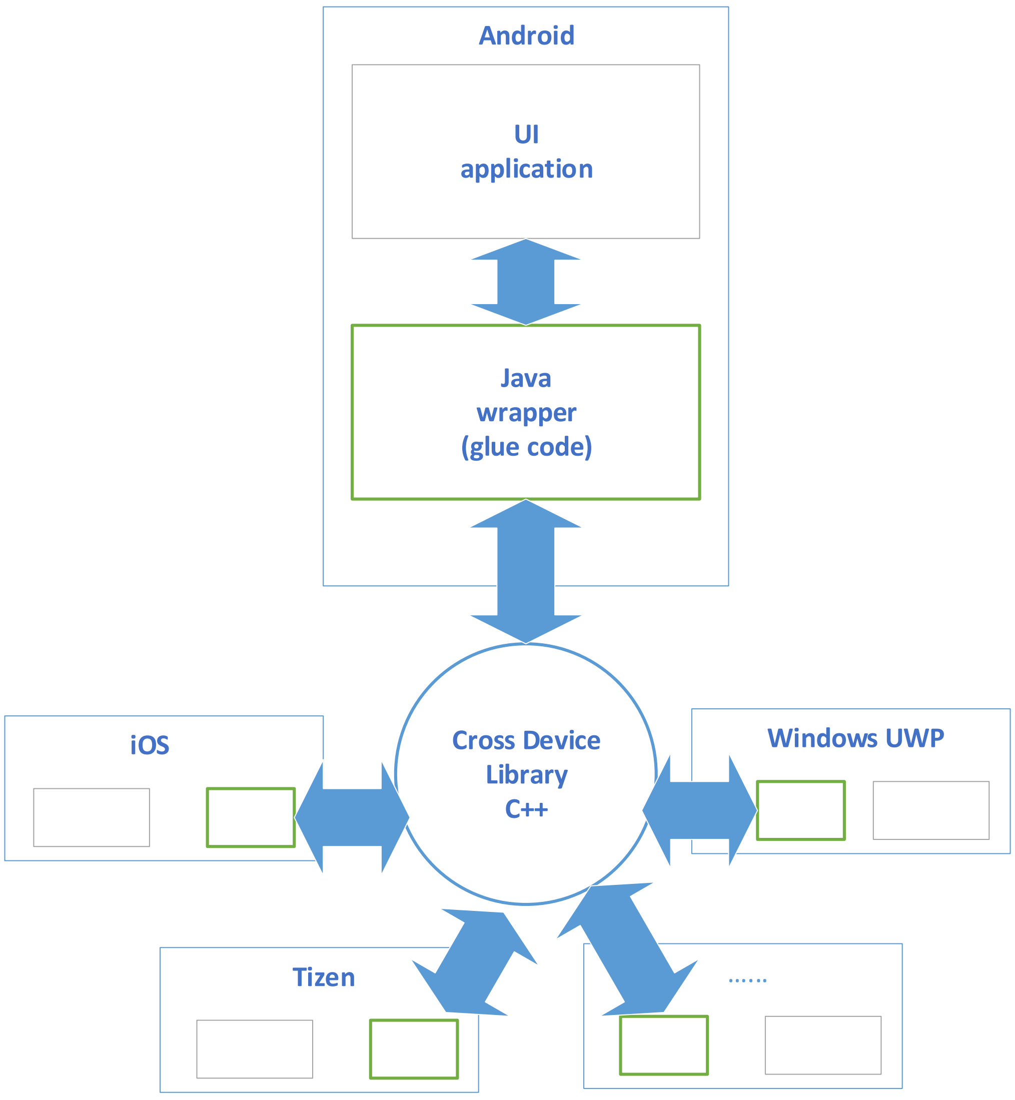


Figure 6.1: Interoperability between devices

Extensible: The library must cope with new additional requests. So, the library must be *extensible*.

Interoperability between Devices: The library must be developed in a way that different devices with different OS can communicate with each other. Figure 6.1 shows a possible approach where we *write once* our library and where we can run that library *everywhere*. The idea is that we write the library in a language that is runnable with multiple programming languages. In our proposal, we have chosen the C++ language. Android, for example, can run C++ code thanks to the Native Development Kit (NDK) and the Java Native Interface (JNI) framework. In iOS, we have the native programming language Objective-C++ which is a language that allows running Objective-C and C++ code. Figure 6.1 mention also a block where we can define some glue code so that it is easier to work with the library in the target languages. If necessary, we can put in this block also some specific coding for interacting with the OS. Otherwise, a reuse of our core C++ library for other devices is not possible. A possible benefit of working with a C++ library is that it also can work with IoT devices.

Versatility in Space: The library may not be designed for a particular space. Otherwise, we will limit the possibilities for using the library. We must let the user of our library decide if they want to use certain sensors or cameras. Hereby the users will limit the space, not the library. So, we want a library that has the ability to work without any space restrictions and can work anywhere, like our CrossWoW application.

Versatility in Granularity: In our CrossWoW application, we have the possibility to play video files with or without the audio track. If the library can support a fine granularity for the offered items, more cross-device applications are possible.

Flexible and Customisable: The library must be provided for the use of different types of applications. Our wrapper is only written for media applications. For example, another feature can be the communication of one or all contact persons of our address book to another device with some defined granularity. So, the library must provide different components where the users of the library can decide which components and combinations they want to use.

Synchronised Devices: If a particular event occurs in one of the devices, all other devices must be automatically notified. In our wrapper, each client device keeps the engine device informed of changes so that these changes can be broadcast to all the other client devices.

6.2 Summary and Contributions

The ultimate goal of this dissertation was to explore the possibilities for the rapid prototyping of XDI. Therefore, we have made two POC applications to find out which issues arise when creating an application that requires interactions between different devices. In doing so, we looked at what could have helped us in the beginning if we had some framework available. What were the biggest problems and what had cost us a lot of development time? Based on this information, we have proposed a possible framework that can be further investigated in the future.

In conclusion, we recapitulate a number of contributions that have been made in this thesis. Thanks to our CrossDeviceWrapper we have been able to implement our second proof-of-concept application CrossWoW Home very quickly. Our wrapper has helped us to avoid unnecessary rework. Hence, we did not have to start from scratch. Similar future media applications can use our wrapper to create cross-device media applications in a faster way. Last but not least, we provided some design guidelines, which will also guide future developers in the creation of a framework for the prototyping of cross-device applications.

6.3 Discussion

Because we have never made applications with cross-device interactions. This study was a challenge and has learned us a lot about potential interactions and what the important guidelines are to develop such applications.

We were very glad we have found a solution for our problem to communicate and stream media files with some granularity to different devices. A problem for which we have not found a better solution in the Android play store. In that solution, we have found a solution for detecting other devices and communicating with these devices in a user-friendly way. After our POC applications, we see also other possibilities of granularity for video files. For example, it would be perfectly possible to stream the same video file with multiple audio tracks to two devices where one device plays the video with the Dutch audio and the other one with the English audio track.

We have implemented a parallel use of related content where a single task is executed on multiple devices (controller-viewer/analyser). A solution where no internet connection is needed, so we can execute our task anywhere with some granularity.

If we compare our solution with the design guidelines of Fisher et al. [21], we see that we have met most guidelines.

Consistency: We have a consistent look on all our devices. For example, our remote controller will have a consistent look on all the devices, engine and clients.

Synchronisation: Our system can broadcast message, so all the actions of one device must be reflected on the other devices. If this action is coming from a client device, then we will send first that message to the engine who will broadcast that action.

Heterogeneous Hardware: At this moment we only support Android systems, but is perfectly possible to implement the same behaviour on other devices to participate. The Alljoyn library can work on different OS and there exists also alternatives for the other solutions. For example, an alternative solution for Exoplayer can be the VLC software[1] which has solutions for multiple OS.

Volatile Device Ecosystem: Our application can cope with client devices that can join or leave an application at any time. For engine devices, this is of course not always the case.

Limited Resources: In our test, we have tested our application on different mobile devices and we have not found any problem with possible limited resources.

Data Transfer: Every device in our application can become an engine device and can distribute the media files where necessary.

Physical Space: In our solution, not all the devices can join or leave at any time because our client devices are depending on our engine device. The circle is created by the engine device and is also broken when the engine device exit that circle. Hereby the connected client devices are not connected anymore with each other.

Asymmetric Functionality: Each device can change their functionality. An engine device can become a client device and vice versa.

Our dissertation was a prerequisite for making a future framework that is very important in a world where we are effectively surrounded more and more by all kinds of devices and RFID tags. So, we look forward to seeing the further evolution in XDI. Many companies encourage a paperless digital world with the use of a lot of mobile devices. The success of this digital world will certainly depend on the optimal interactions between these mobile devices.

[1]`https://www.videolan.org/vlc/`

A Your Appendix

A.1 Paper Prototypes

On the next pages, we show our first paper prototypes that we have created for our Proof Of Concept applications. We have mentioned this phase of our methodology at the end of Chapter 3.

A.2 Coding examples with the Alljoyn Library

We have explained in Chapter 4 two different examples where we will use three devices in each scenario. One device will start the application in Player Mode, the two other devices will start the application in Receiver Mode. In order to easier explain the scenario, we will give a name to each device. “P” is the name for the device in Player Mode. “R1” and “R2” are the names for the two devices that have started the Receiver Mode. In Listing A.1, we describe the first part where we send our media list to the Alljoyn Bus. In Listing A.2, we register the `BusSignalHandler` that will catch the signal of P and send the list to all the clients, R1 and R2 in our scenario. Listing A.3 shows the method call that is followed by a signal which is show in Listing A.4.

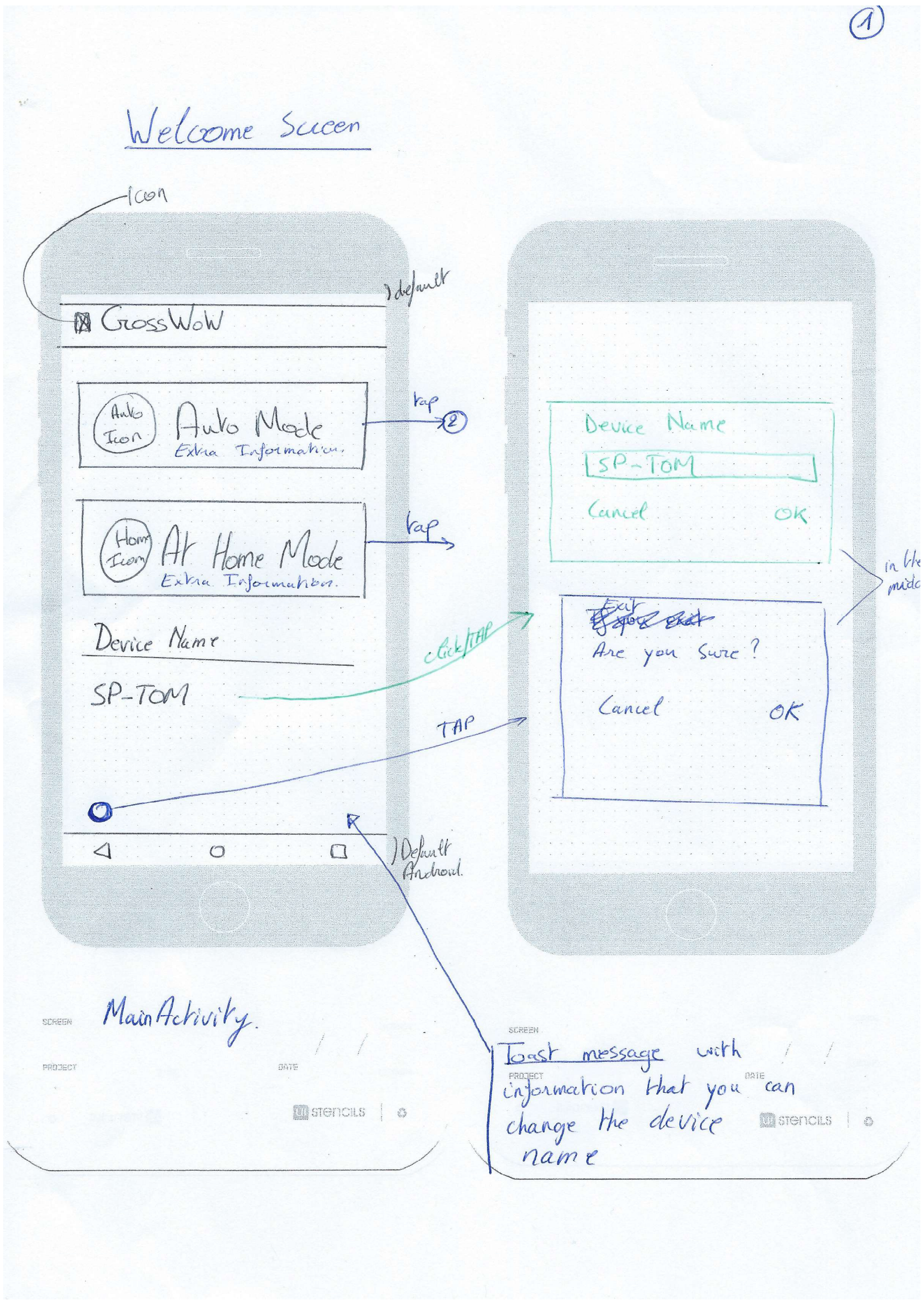
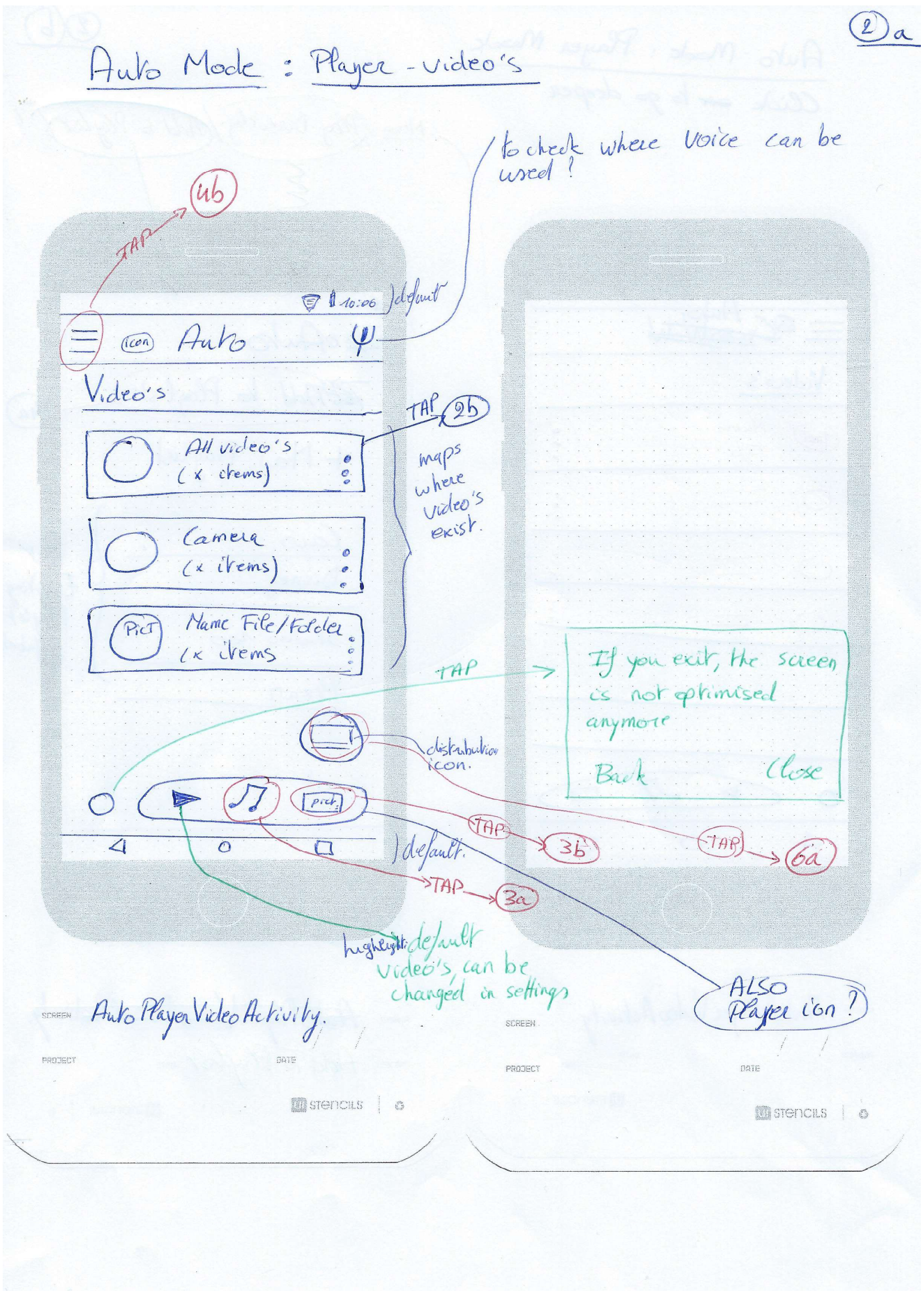


Figure A.1: AUTO: Welcome Screen, where we can change the logical device name



2a
Auto Mode : Player - video's
to check where Voice can be used ?
4b
TAP
10:06
default
Icon
Auto
Video's
TAP
2b
All video's (x items)
Camera (x items)
PicT
Name File/Folder (x items)
maps where video's exist.
TAP
distribution icon.
If you exit, the screen is not optimised anymore
Back
Close
Pict
TAP
3b
TAP
6a
default.
TAP
3a
highlight default video's, can be changed in settings
ALSO Player icon ?
SCREEN
Auto Player Video Activity
PROJECT
DATE
stencils

Figure A.2: AUTO: First screen in Player Mode

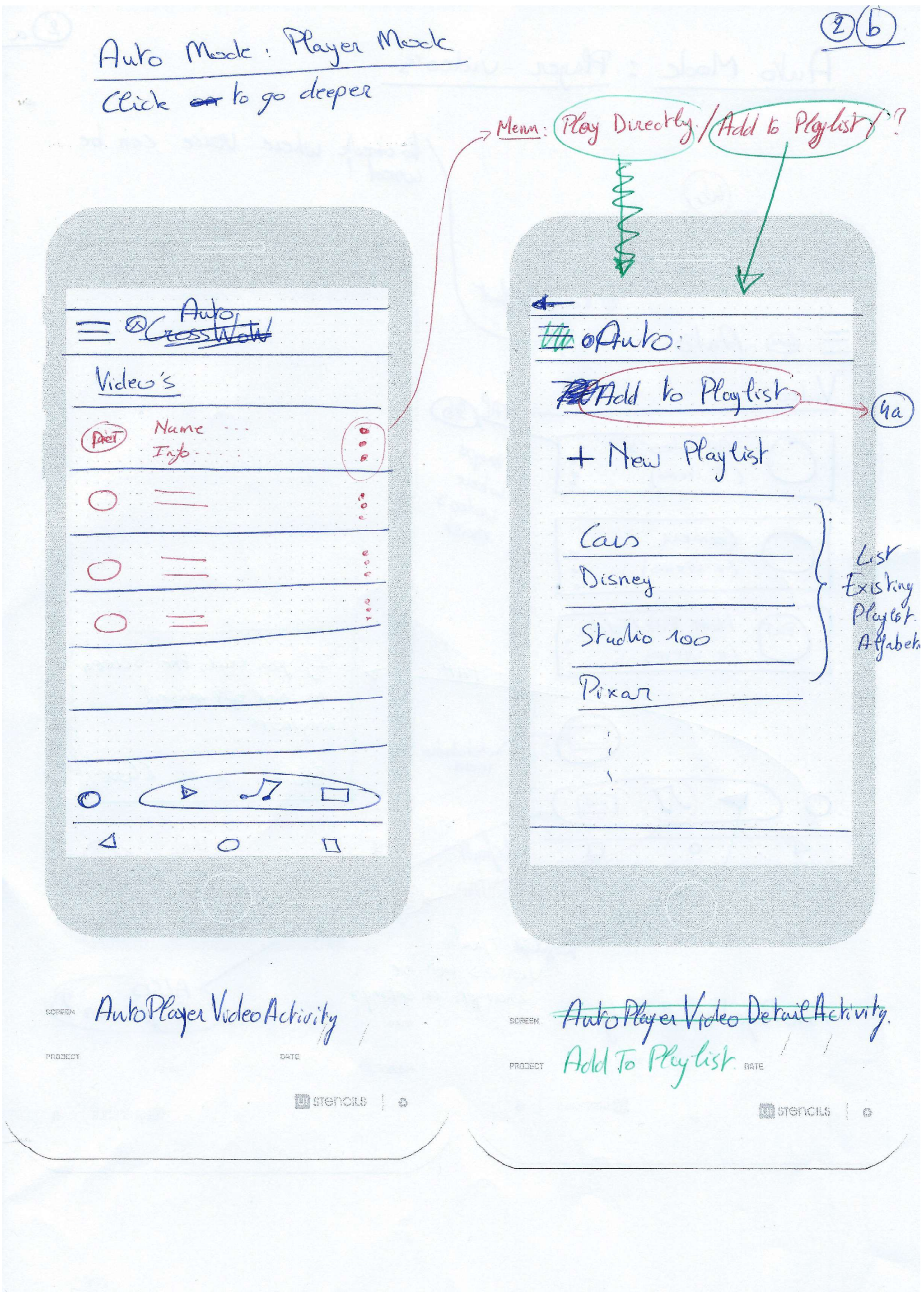


Figure A.3: AUTO: Possible actions for video items

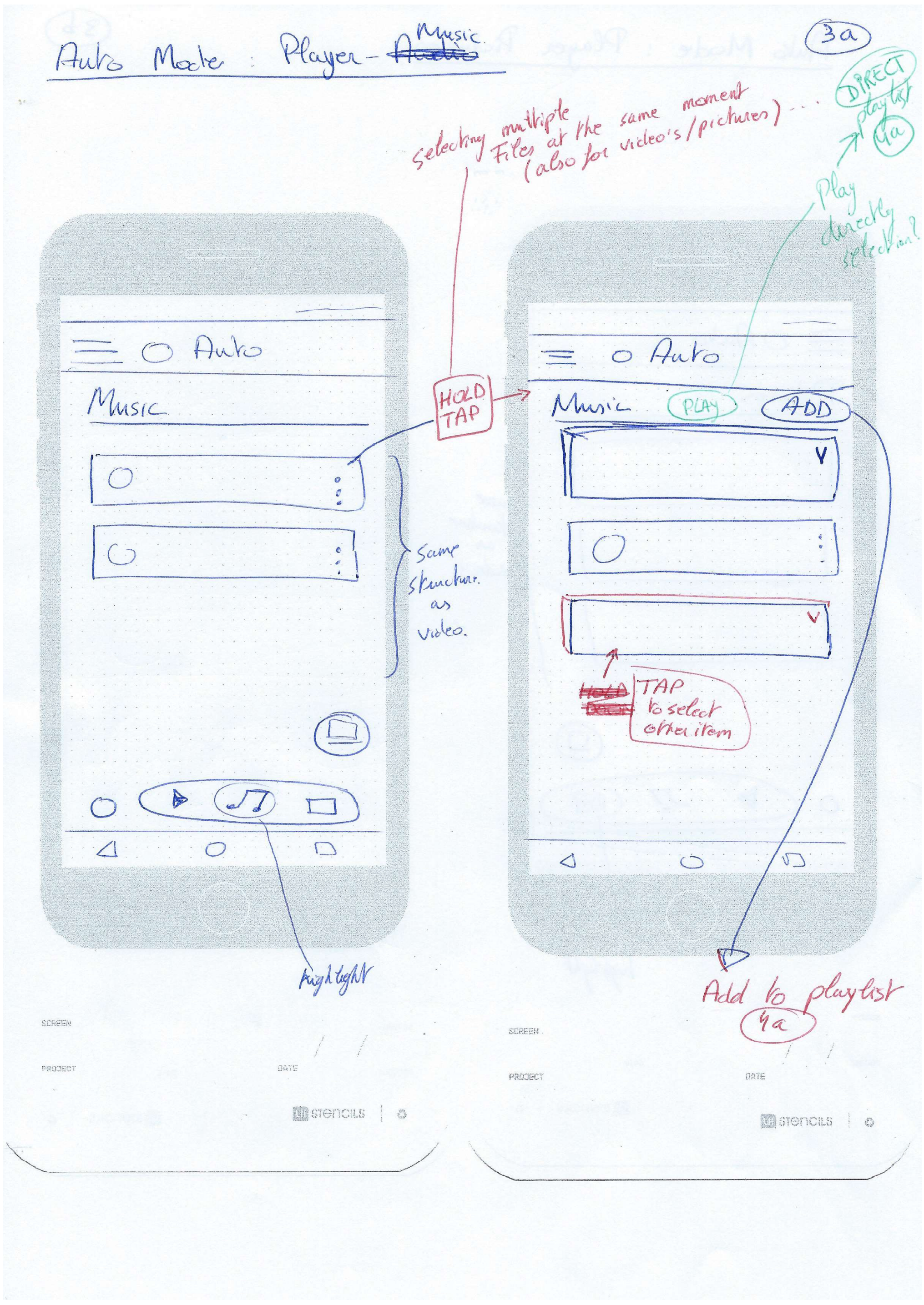


Figure A.4: AUTO: Sceen for music items - same as for video items

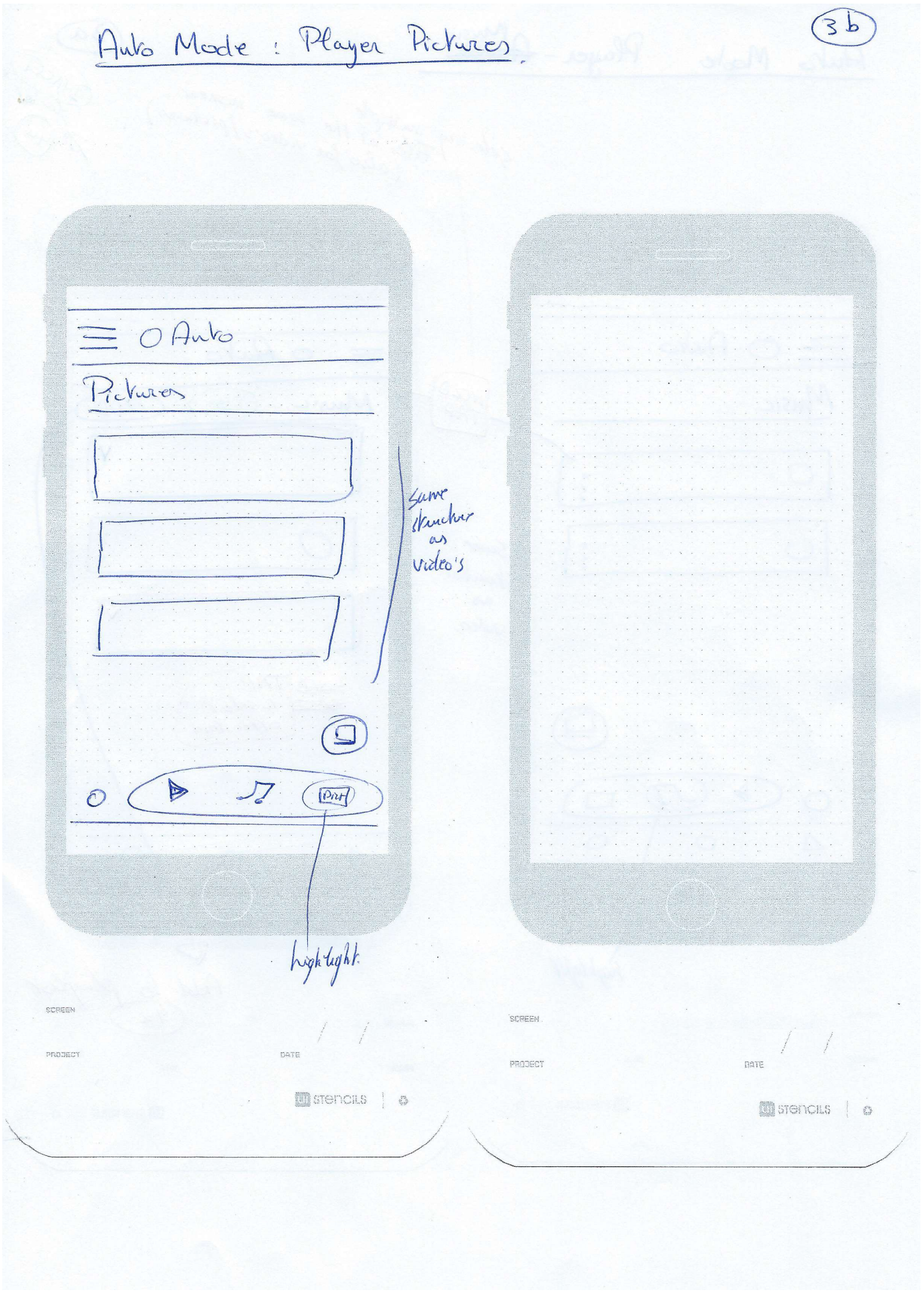


Figure A.5: AUTO: Screen for images - same as for video items

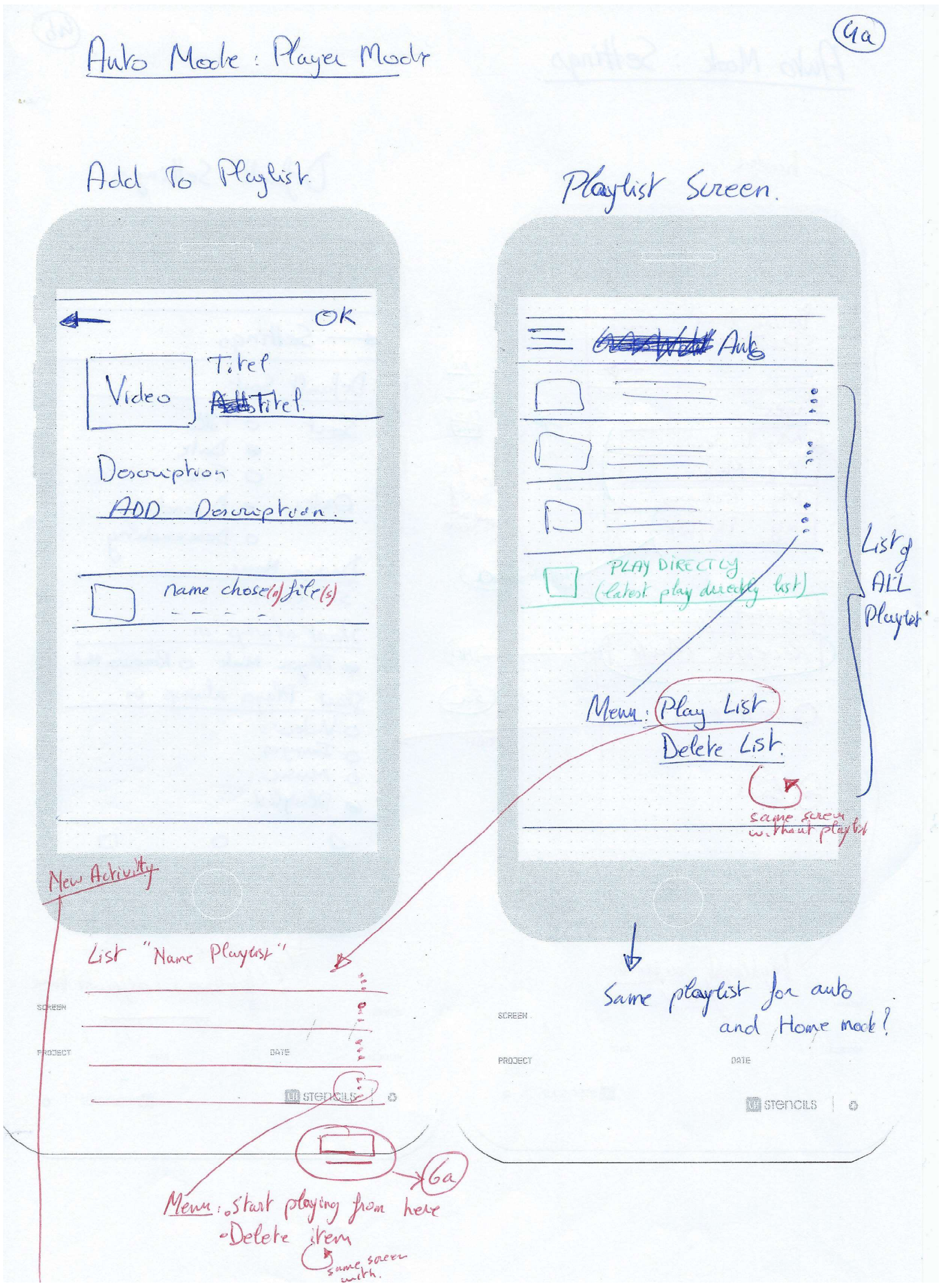


Figure A.6: AUTO: Playlist screens

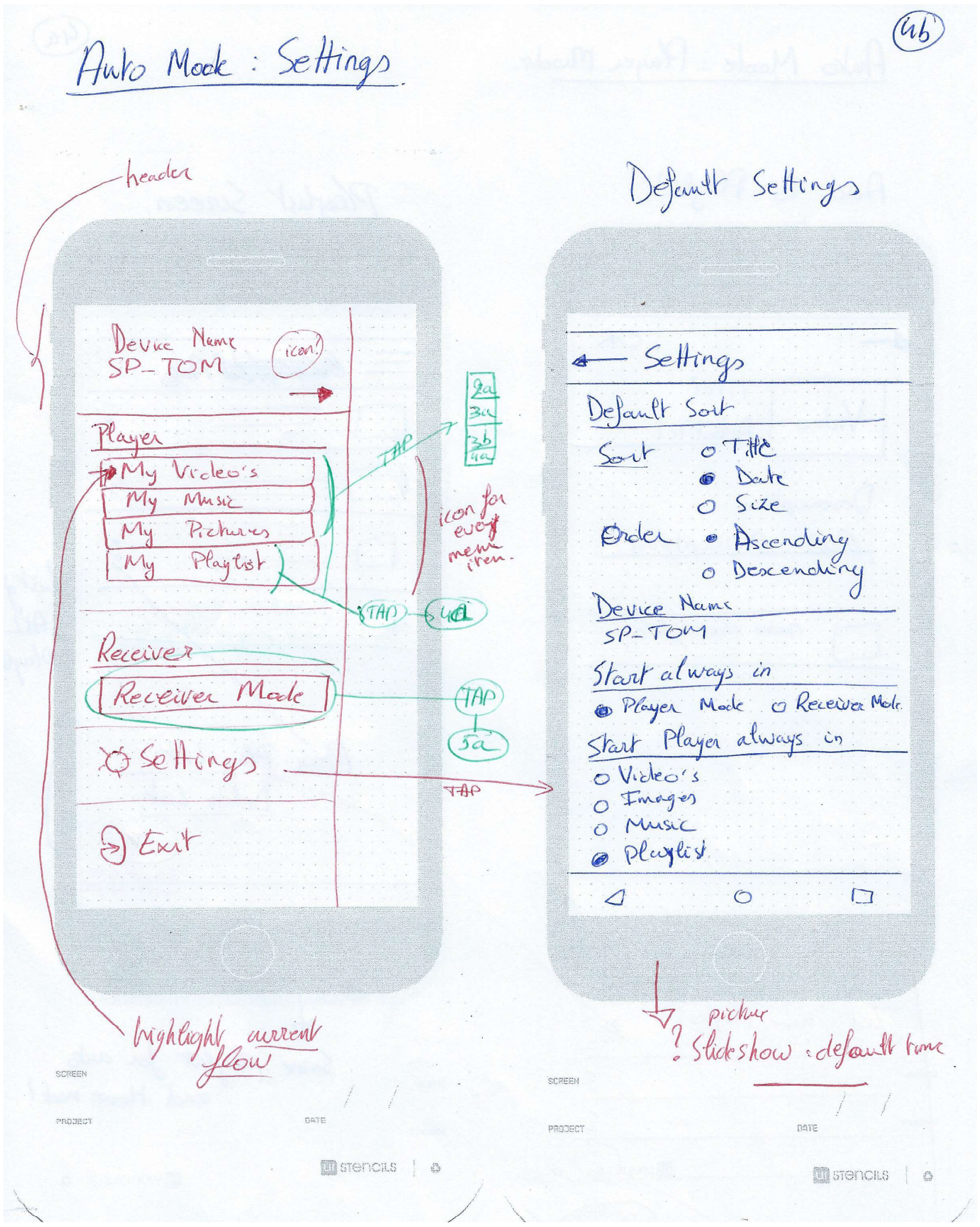


Figure A.7: AUTO: Menu and settings screen

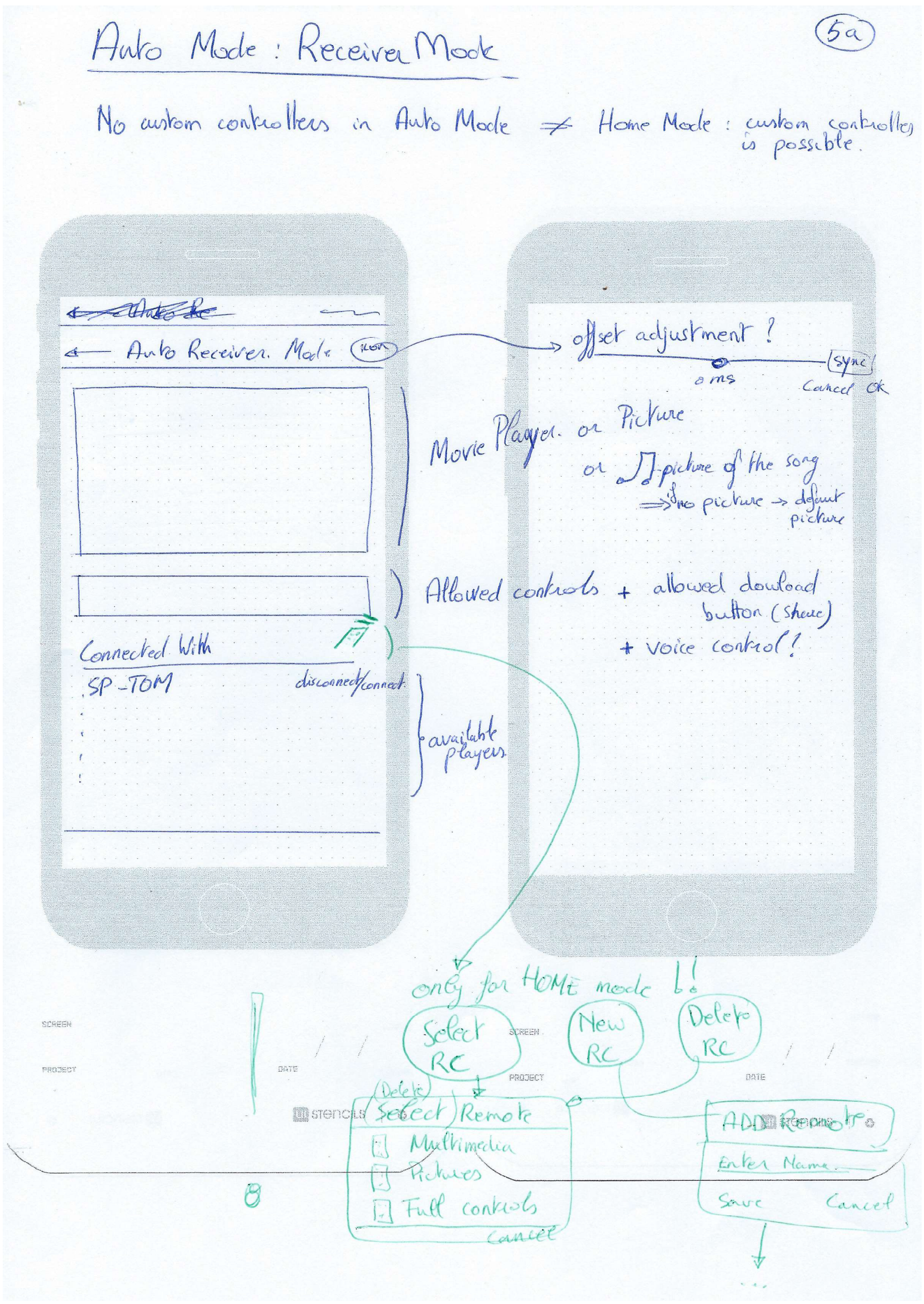


Figure A.8: AUTO: First screen in Receiver Mode

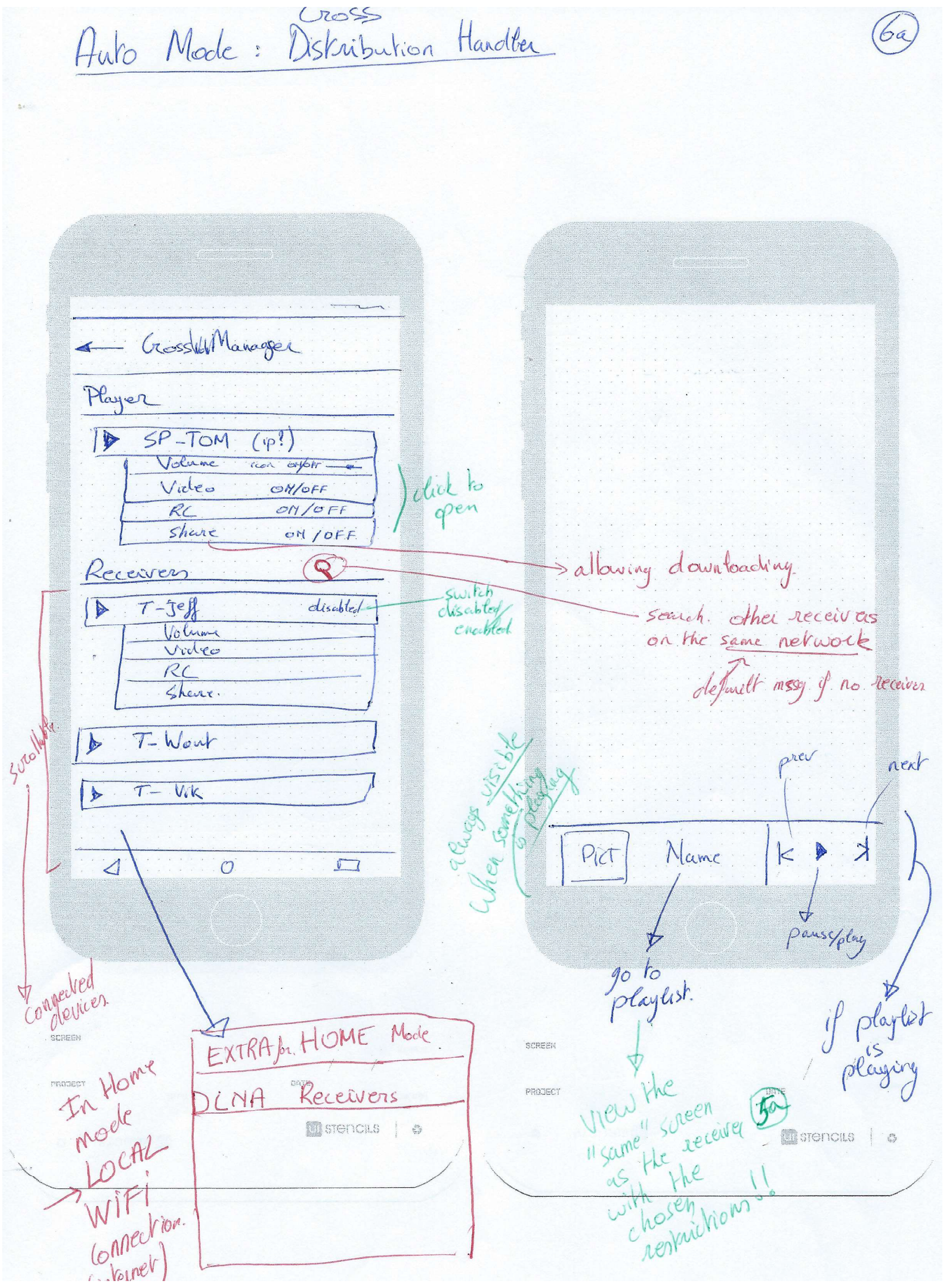


Figure A.9: AUTO: Distribution Manager in Player Mode

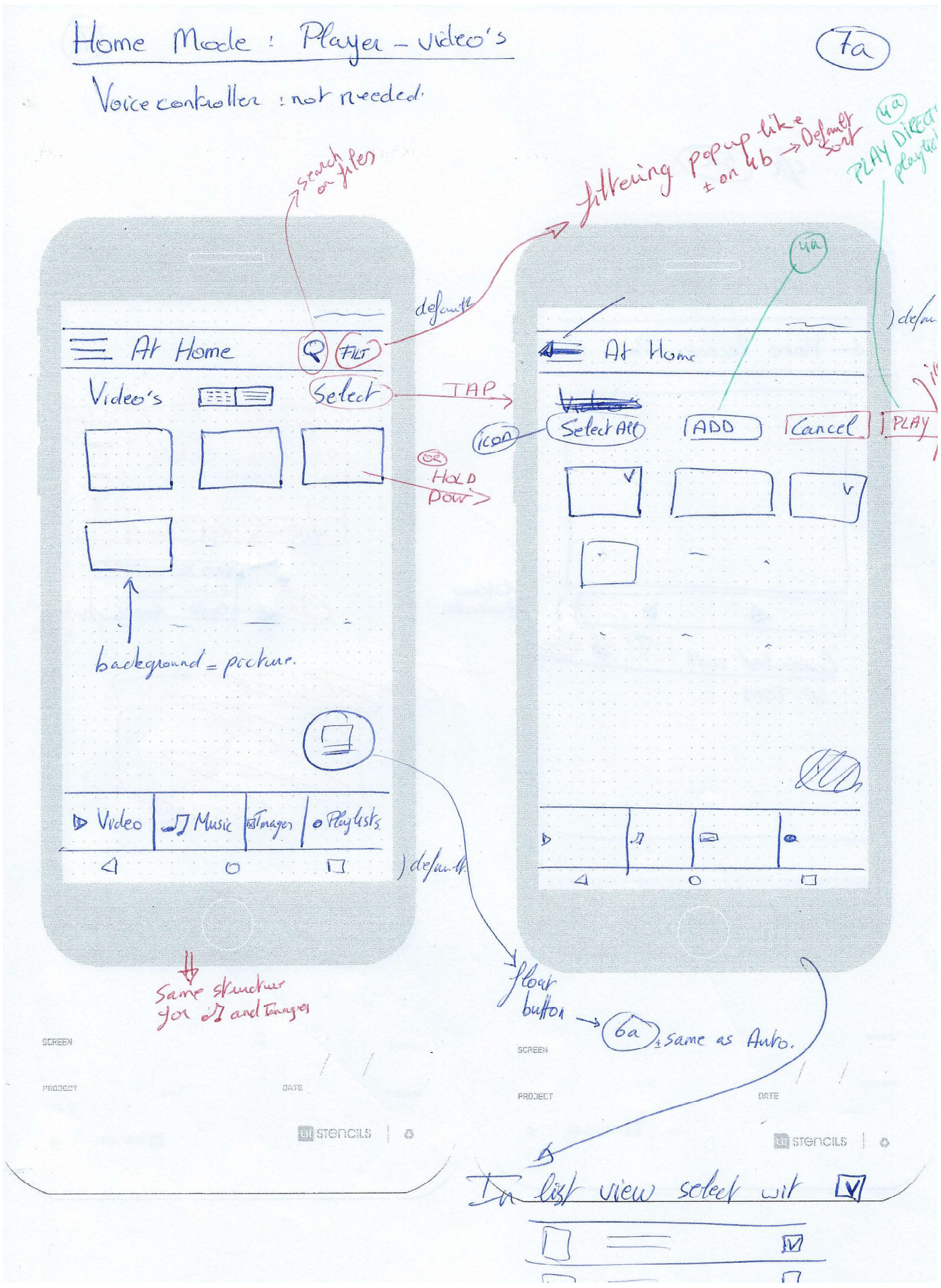


Figure A.10: HOME: First screen in Player Mode

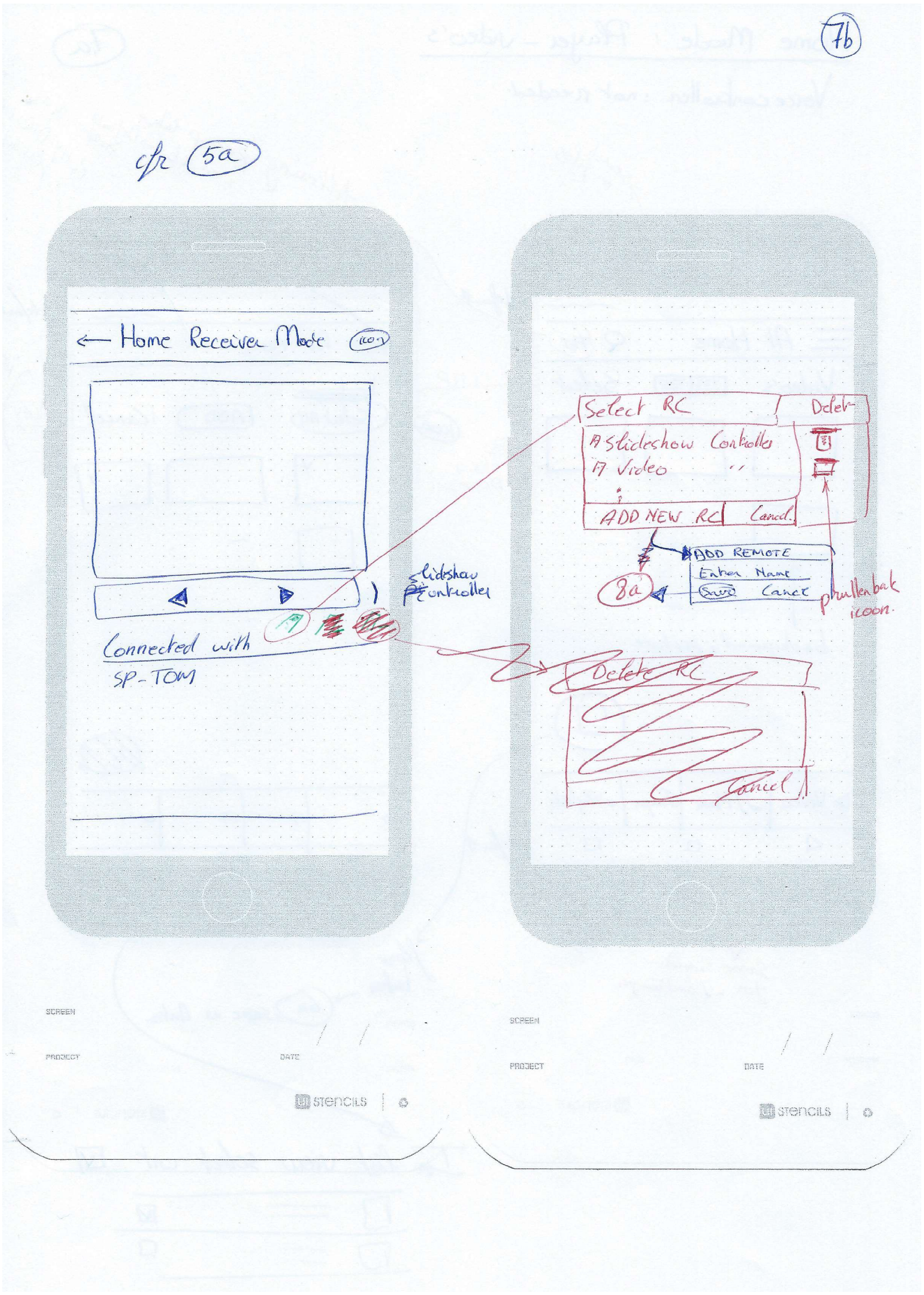


Figure A.11: HOME: First screen in Receiver Mode

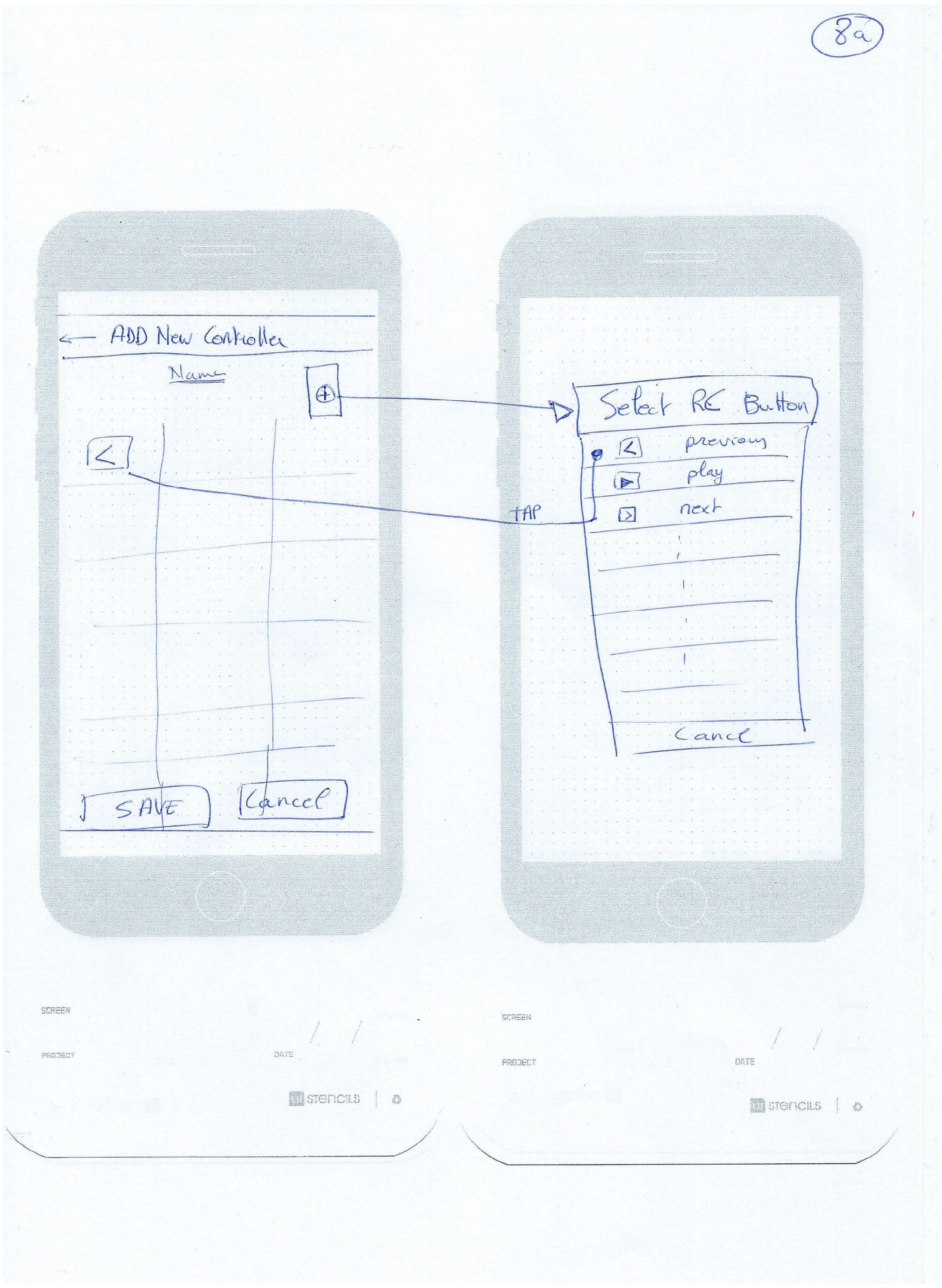


Figure A.12: HOME: prototype for making a custom controller - NOT implemented

```
//P starts Player Mode and start the CrossEngineService
//R1 & R2 starts Receiver Mode and connect to P with the logical name of P
//P send the media list to all the clients
private void DistributeMedia() {
Log.i(TAG, "Distribute Media");
CrossDevMedia media = new CrossDevMedia();
DeviceName name = new DeviceName();
name.displayName = localEngineService.storedDevice.displayName;
name.uniqueName = localEngineService.storedDevice.uniqueName;
media.engineName = name;
media.engineDomain = ipdevice;
media.currentWindowPosition=mediaPosition;
media.setMediaItemsCount(mediaURI_List.size());
int j = 0;
if (mediaURI_List.size() == mediaType_List.size()) {
for (String item : mediaURI_List) {
media.mediaItems[j].name = "name";
media.mediaItems[j].URI = ipdevice + mediaURI_List.get(j).replace(" ","+");
media.mediaItems[j].type = mediaType_List.get(j);
j++;
}
}
localEngineService.sendMedia(media);
}

//CrossEngineService
public boolean sendMedia(CrossDevMedia media) {
return mEndService.getAjCommMgr().sendMedia(media);
}

//EngineEndService: sendMedia
public boolean sendMedia(CrossDevMedia media) {
if (null == mDevI) {
Log.e(LOG_TAG, "mDevI is null");
return false;
}
Log.i(LOG_TAG, "sendMedia media ");
Message msg = mBackgroundHandler.obtainMessage(SIGNAL_SEND_MEDIA);
msg.obj = media;
mBackgroundHandler.sendMessage(msg);
return true;
}
...
case SIGNAL_SEND_MEDIA:
doSignalSendMedia((CrossDevMedia) msg.obj);
break;
...
private void doSignalSendMedia(CrossDevMedia media) {
Log.i(LOG_TAG, "doSignalSendMedia media ");
if (mDevI != null)
{
try {
mDevI.sendMediaOnSignal(media); // <---
}
catch (BusException e) {
e.printStackTrace();
}
}
}
```

Listing A.1: Signal: send media list to Alljoyn Bus

```
//ClientCommMgr - register signal handler
registerSignalHandlersHelper("handleSendMediaOnSignal", RecFromEngineHandler.iFaceName,
RecFromEngineHandler.sendMediaOnSignal, new Class<?>[]{CrossDevMedia.class});

//handler
@BusSignalHandler(iface = "be.vub.crossdeviceservice.Device.Interfaces.IDeviceMediaInterface",
      signal = "sendMediaOnSignal")
public void handleSendMediaOnSignal(CrossDevMedia media) {
Log.i(TAG, "handleSendMediaOnSignal");
synchronized (RecFromEngineHandler.this) {
 if (null == mIBusDataListener) {
Log.e(TAG, "mIBusDataListener is null, data lost");
return;
}

boolean bRet = mIBusDataListener.RecvMediaBusData(media); //<-- send message ...
 if (!bRet) {
Log.e(TAG, "mIBusDataListener.RecvMediaBusData fail!");
}
}
}

//RecvMediaBusData --> IBusDataClientListener must be implemented in application
//create your own listeners - region **** listeners methods ****
//1. IClientActionListener is splitted because we have two activities
private IClientActionListener mSourceActionListener = null;
public void setSourceActionListener(IClientActionListener listener ) {
Log.i(TAG, "Set Source Listener");
mSourceActionListener = listener;
}
private IClientMediaListener mSourceMediaListener = null;
public void setSourceMediaListener(IClientMediaListener listener) {
Log.i(TAG, "Set Source Listener");
mSourceMediaListener = listener;
}

//CrossClientService
private IClientMediaListener mSourceMediaListener = null;
private class BusDataClientListener implements IBusDataClientListener {
 ...
@Override
public boolean RecvMediaBusData(CrossDevMedia crossDevMedia) {
Log.i(TAG, "RecvMediaBusData data: " + crossDevMedia);
 if (null != mSourceMediaListener) {
mSourceMediaListener.setRecvMedia(crossDevMedia); <---
}
return true;
}
 ...
}

//AutoReceiverMainActivity
private class ClientMediaListener implements IClientMediaListener {
@Override
public void setRecvMedia(CrossDevMedia media) {
// ... coding to go through the media list and start an specific activity
startActivity(intent);
}
}
```

Listing A.2: Signal: Receive signal and send it to all the clients

```
//R1 has received a controller and stops the video
//R1 send a method to P and P broadcast this message to all their clients
//1. R1 application
case R.id.exo_mypause:
action.clientAction = ActionPlayer.PAUSE.toString();
localClientService.sendClientAction(action);
break;
//2. CrossClientService R1
public boolean sendClientAction(DeviceAction action) {
return mEndService.getAjCommMgr().sendClientAction(action);
}
//3. ClientCommMgr R1
mDevI.sendDevAction(deviceAction);

//4. EngineCommMgr P
@Override
@BusMethod
public void sendDevAction(DeviceAction deviceAction) throws BusException {
Log.i(LOG_TAG, "sendDevAction string: " + deviceAction.clientAction);
if (null == mIBusDataListener) {
Log.e(LOG_TAG, "mIBusDataListener is null, data lost");
}
Log.i(LOG_TAG, "mIBusDataListener - start RecvDevConfig");
mIBusDataListener.RecvDevAction(deviceAction);
};

//5. CrossEngineService P
@Override
public boolean RecvDevAction(DeviceAction deviceAction) {
Log.i(TAG, "RecvDevActionOnSignal data: " + deviceAction.clientName.displayName);
if (null != mSourceListener) {
mSourceListener.setRecvDevAction(deviceAction);
}
return true;
}

//6. P application
@Override
public void setRecvDevAction(DeviceAction deviceAction) {
Log.i(TAG, "Engine Player setRecvDevAction: " + deviceAction.clientAction);
...
if (deviceAction.clientAction.equals(ActionPlayer.PAUSE.toString()) ) {
View vw = findViewById(R.id.exo_mypause);
ActionClick(vw, false);
}
}
....
case R.id.exo_mypause:
localEngineService.sendEngineAction(ActionPlayer.PAUSE.toString());
player.setPlayWhenReady(false);
```

Listing A.3: Method: R1 send PAUSE action to P

```
//7. CrossEngineService P
public boolean sendEngineAction(String action){
return mEndService.getAjCommMgr().sendEngineAction(action);
}

//9. ClientCommMgr R1 + R2
registerSignalHandlersHelper("handleSendEngineActionOnSignal", RecFromEngineHandler.iFaceName,
RecFromEngineHandler.sendEngineActionOnSignal, new Class<?>[]{String.class});
 ...
@BusSignalHandler(iface = "be.vub.crossdeviceservice.Device.Interfaces.IDeviceMediaInterface",
       signal = "sendEngineActionOnSignal")
public void handleSendEngineActionOnSignal(String action) {
Log.i(TAG, "handleSendEngineActionOnSignal action: " + action);
synchronized (RecFromEngineHandler.this) {
 if (null == mIBusDataListener) {
Log.e(TAG, "mIBusDataListener is null, data lost");
return;
}
boolean bRet = mIBusDataListener.RecvEngineActionBusData(action);
 if (!bRet) {
Log.e(TAG, "mIBusDataListener.RecvEngineActionBusData fail!");
}
}
}

//11. CrossClientService R1 + R2
@Override
public boolean RecvEngineActionBusData(String s) {
Log.i(TAG, "RecvEngineActionBusData data: " + s);
 if (null != mSourceActionListener) {
mSourceActionListener.setRecvEngineAction(s);
}
return true;
}

//12. AutoReceiver activity R1 + R2
@Override
public void setRecvEngineAction(String action) {
boolean isPlaying = false;
 ....
 if (action.equals(ActionPlayer.PAUSE.toString())) {
player.setPlayWhenReady(false);
}
}
```

Listing A.4: Signal: P broadcast the action from R1 to all the clients

Bibliography

[1] Mathias Baglioni, Eric Lecolinet, and Yves Guiard. JerkTilts: Using Accelerometers for Eight-Choice Selection on Mobile Devices. In *Proceedings of ICMI 2011, Conference on Multimodal Interfaces*, pages 121–128, Alicante, Spain, 2011.

[2] Rajiv D Banker, Srikant M Datar, Chris F Kemerer, and Dani Zweig. Software Complexity and Maintenance Costs. *Communications of the ACM*, 36(11):81–95, 1993.

[3] Rajiv D Banker, Gordon B Davis, and Sandra A Slaughter. Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study. *Management Science*, 44(4):433–450, 1998.

[4] Jakob Bardram, Sofiane Gueddana, Steven Houben, and Søren Nielsen. ReticularSpaces: Activity-Based Computing Support for Physically Distributed and Collaborative Smart Spaces. In *Proceedings of CHI 2011, ACM Conference on Human Factors in Computing Systems*, pages 2845–2854, Austin, USA, 2012.

[5] Michel Beaudouin-Lafon and Wendy Mackay. Prototyping Tools and Techniques. *Human Computer Interaction-Development Process*, pages 122–142, 2003.

[6] Jacob T. Biehl and Brian P. Bailey. ARIS: An Interface for Application Relocation in an Interactive Space. In *Proceedings of GI 2004, Conference on Graphics Interface*, pages 107–116, London, Canada, 2004.

[7] Marco Blumendorf, Dirk Roscher, and Sahin Albayrak. Dynamic User Interface Distribution for Flexible Multimodal Interaction. In *Proceedings of ICMI-MLMI 2010, Conference on Multimodal Interfaces and the Workshop on Machine Learning for Multimodal Interaction*, pages 20:1–20:8, Beijing, China, 2010.

[8] Barry W Boehm. Understanding and Controlling Software Costs. *Journal of Parametrics*, 8(1):32–68, 1988.

[9] Tsung-Hsiang Chang and Yang Li. Deep Shot: A Framework for Migrating Tasks Across Devices Using Mobile Phone Cameras. In *Proceedings of CHI 2011, ACM Conference on Human Factors in Computing Systems*, pages 2163–2172, Vancouver, Canada, 2011.

[10] Keith Cheverst, Alan Dix, Daniel Fitton, Chris Kray, Mark Rouncefield, Corina Sas, George Saslis-Lagoudakis, and Jennifer G Sheridan. Exploring Bluetooth Based Mobile Phone Interaction with the Hermes Photo Display. In *Proceedings of CHI 2005, ACM Conference on Human Computer Interaction with Mobile Devices & Services*, pages 47–54, 2005.

[11] Pei-Yu (Peggy) Chi and Yang Li. Weave: Scripting Cross-Device Wearable Interaction. In *Proceedings of CHI 2015, ACM Conference on Human Factors in Computing Systems*, pages 3923–3932, Seoul, Republic of Korea, 2015.

[12] Nigel Davies, Daniel P Siewiorek, and Rahul Sukthankar. Activity-Based Computing. *IEEE Pervasive Computing*, 7:20–21, 2008.

[13] Olga De Troyer and Sven Casteleyn. Modeling Complex Processes for Web Applications Using WSDM. In *Proceedings of the 3rd International Workshop on Web-Oriented Software Technologies*, pages 27–50, 2003.

[14] Olga De Troyer, Sven Casteleyn, and Peter Plessers. *WSDM: Web Semantics Design Method*, pages 303–351. Springer London, London, 2008.

[15] David Dearman and Jeffery S. Pierce. It's on My Other Computer!: Computing with Multiple Devices. In *Proceedings of CHI 2008, ACM Conference on Human Factors in Computing Systems*, pages 767–776, Florence, Italy, 2008.

[16] A. Demeure, J. S. Sottet, G. Calvary, J. Coutaz, V. Ganneau, and J. Vanderdonckt. The 4C Reference Model for Distributed User Interfaces. In *Proceedings of ICAS 2008,Conference on Autonomic and Autonomous Systems*, pages 61–69, March 2008.

[17] Linda Di Geronimo, Ersan Aras, and Moira C. Norrie. *Tilt-and-Tap: Framework to Support Motion-Based Web Interaction Techniques*, pages 565–582. Springer International Publishing, Cham, 2015.

[18] Linda Di Geronimo, Maria Husmann, Abhimanyu Patel, Can Tuerk, and Moira C. Norrie. Ctat: Tilt-and-tap Across Devices. In *International Conference on Web Engineering*, pages 96–113, 2016.

[19] Niklas Elmqvist. *Distributed User Interfaces: State of the Art*, pages 1–12. Springer London, London, 2011.

[20] Facebook. Facebook for Business. `https://www.facebook.com/business/news/Finding-simplicity-in-a-multi-device-world`. Retrieved on November 30, 2016.

[21] Eli Raymond Fisher, Sriram Karthik Badam, and Niklas Elmqvist. Designing Peer-to-peer Distributed User Interfaces: Case Studies on Building Distributed Applications. *International Journal of Human-Computer Studies*, 72(1):100–110, 2014.

[22] Luca Frosini, Marco Manca, and Fabio Paternò. A Framework for the Development of Distributed Interactive Applications. In *Proceedings of EICS 2013, ACM Symposium on Engineering Interactive Computing Systems*, pages 249–254, London, United Kingdom, 2013.

[23] Luca Frosini and Fabio Paternò. A Framework for Improving the Multi-Device User Experience in Smart Cities. *Smart Cities*, page 14, 2014.

[24] Mayank Goel, Brendan Lee, Md. Tanvir Islam Aumi, Shwetak Patel, Gaetano Borriello, Stacie Hibino, and Bo Begole. SurfaceLink: Using Inertial and Acoustic Sensing to Enable Multi-Device Interaction on a Surface. In *Proceedings of CHI 2014, ACM Conference on Human Factors in Computing Systems*, pages 1387–1396, Toronto, Canada, 2014.

[25] Google. The New Multi-Screen World Study - Think with Google. `https://www.thinkwithgoogle.com/research-studies/the-new-multi-screen-world-study.html`, 8 2012. Retrieved on November 18, 2016.

[26] Peter Hamilton and Daniel J. Wigdor. Conductor: Enabling and Understanding Cross-Device Interaction. In *Proceedings of CHI 2014, ACM Conference on Human Factors in Computing Systems*, pages 2773–2782, Toronto, Canada, 2014.

[27] Ken Hinckley, Gonzalo Ramos, Francois Guimbretiere, Patrick Baudisch, and Marc Smith. Stitching: Pen Gestures That Span Multiple Displays. In *Proceedings of AVI 2004, ACM Conference on Advanced Visual Interfaces*, pages 23–31, Gallipoli, Italy, 2004.

[28] Steven Houben and Nicolai Marquardt. WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications. In *Proceedings of CHI 2015, ACM Conference on Human Factors in Computing Systems*, pages 1247–1256, Seoul, Republic of Korea, 2015.

[29] Da-Yuan Huang, Chien-Pang Lin, Yi-Ping Hung, Tzu-Wen Chang, Neng-Hao Yu, Min-Lun Tsai, and Mike Y. Chen. MagMobile: Enhancing Social Interactions with Rapid View-Stitching Games of Mobile Devices. In *Proceedings of MUM 2012, ACM Conference on Mobile and Ubiquitous Multimedia*, pages 61:1–61:4, Ulm, Germany, 2012.

[30] Dugald Ralph Hutchings, John Stasko, and Mary Czerwinski. Distributed Display Environments. *interactions*, 12(6):50–53, November 2005.

[31] Djilali Idought and Aicha Azoui. SOA Based Ubiquitous Computing System Design Framework. In *Proceedings of MobiWac 2014, ACM Symposium on Mobility Management and Wireless Access*, pages 71–75, Montreal, Canada, 2014.

[32] Tero Jokela, Jarno Ojala, and Thomas Olsson. A Diary Study on Combining Multiple Information Devices in Everyday Activities and Tasks. In *Proceedings of CHI 2015, ACM Conference on Human Factors in Computing Systems*, pages 3903–3912, Seoul, Republic of Korea, 2015.

[33] Ali K Kamrani and Emad Abouel Nasr. *Rapid Prototyping: Theory and Practice*, volume 6. Springer Science & Business Media, 2006.

[34] Dejan Kovachev, Dominik Renzel, Petru Nicolaescu, and Ralf Klamma. DireWolf - Distributing and Migrating User Interfaces for Widget-Based Web Applications. In Florian Daniel, Peter Dolog, and Qing Li, editors, *Proceedings of ICWE 2013, ACM Conference on Web Engineering*, pages 99–113, Aalborg, Denmark, 2013.

[35] Sang-won Leigh, Philipp Schoessler, Felix Heibeck, Pattie Maes, and Hiroshi Ishii. THAW: Tangible Interaction with See-Through Augmentation for Smartphones on Computer Screens. In *Proceedings of TEI 2015, ACM Conference on Tangible, Embedded, and Embodied Interaction*, pages 89–96, Stanford, USA, 2015.

[36] JJ Lòpez-Espin, JA Gallud, E Lazcorreta, A Peñalver, and F Botella. A Formal View of Distributed User Interfaces. In *Proceedings of CHI 2011, ACM Workshop on Distributed User Interfaces*, pages 97–100, Castilla-La Mancha, Spain, 2011.

[37] Andrés Lucero, Jussi Holopainen, and Tero Jokela. Pass-them-around: Collaborative Use of Mobile Phones for Photo Sharing. In *Proceedings of CHI 2011, ACM Conference on Human Factors in Computing Systems*, pages 1787–1796, Vancouver, Canada, 2011.

[38] Nicolai Marquardt, Till Ballendat, Sebastian Boring, Saul Greenberg, and Ken Hinckley. Gradual Engagement: Facilitating Information Exchange Between Digital Devices as a Function of Proximity. In *Proceedings of ITS 2012, ACM Conference on Interactive Tabletops and Surfaces*, pages 31–40, Cambridge, USA, 2012.

[39] Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proceedings of UIST 2011, ACM Symposium on User Interface Software and Technology*, pages 315–326, Santa Barbara, USA, 2011.

[40] David Merrill, Jeevan Kalanithi, and Pattie Maes. Siftables: Towards Sensor Network User Interfaces. In *Proceedings of TEI 2007, ACM Conference on Tangible and Embedded Interaction*, pages 75–78, Baton Rouge, Louisiana, 2007.

[41] Satoshi Nakamoto. Bitcoin: A Peer-to-peer Electronic Cash System, 2008, 2012.

[42] Michael Nebeling, Elena Teunissen, Maria Husmann, and Moira C. Norrie. XDKinect: Development Framework for Cross-Device Interaction Using Kinect. In *Proceedings of EICS 2014, ACM Symposium on Engineering Interactive Computing Systems*, pages 65–74, Rome, Italy, 2014.

[43] Takashi Ohta and Jun Tanaka. *Pinch: An Interface That Relates Applications on Multiple Touch-Screen by 'Pinching' Gesture*, pages 320–335. Kathmandu, Nepal, 2012.

[44] Fabio Paternò and Carmen Santoro. A Logical Framework for Multi-Device User Interfaces. In *Proceedings of EICS 2012, ACM Symposium on Engineering Interactive Computing Systems*, pages 45–50, Copenhagen, Denmark, 2012.

[45] Sarah Perez. Majority of Digital Media Consumption Now Takes Place in Mobile Apps. https://techcrunch.com/2014/08/21/majority-of-digital-media-consumption-now-takes-place-in-mobile-apps/, 2014. Retrieved on October 23, 2016.

[46] Stefan Poslad. *Ubiquitous Computing: Smart Devices, Environments and Interactions*. Wiley Publishing, 1st edition, 2009.

[47] Roman Rädle, Hans-Christian Jetter, Nicolai Marquardt, Harald Reiterer, and Yvonne Rogers. HuddleLamp: Spatially-Aware Mobile Displays for Ad-hoc Around-the-Table Collaboration. In *Proceedings of ITS 2014, ACM Conference on Interactive Tabletops and Surfaces*, pages 45–54, Dresden, Germany, 2014.

[48] Jun Rekimoto. Pick-And-Drop: A Direct Manipulation Technique for Multiple Computer Environments. In *Proceedings of UIST 1997, ACM Symposium on User Interface Software and Technology*, pages 31–39, Banff, Canada, 1997.

[49] Scott Robertson, Cathleen Wharton, Catherine Ashworth, and Marita Franzke. Dual Device User Interface Design: PDAs and Interactive Television. In *Proceedings of CHI 1996, ACM Conference on Human Factors in Computing Systems*, pages 79–86, Vancouver, Canada, 1996.

[50] D. Saha and A. Mukherjee. Pervasive Computing: A Paradigm for the 21st Century. *Computer*, 36(3):25–31, Mar 2003.

[51] Audrey Sanctorum and Beat Signer. Towards User-Defined Cross-Device Interaction. In *Proceedings of DUI 2016, Workshops on Distributed User Interfaces*, pages 179–187, Lugano, Switzerland, 2016.

[52] Stephanie Santosa and Daniel Wigdor. A Field Study of Multi-Device Workflows in Distributed Workspaces. In *Proceedings of UbiComp 2013, ACM Conference on Pervasive and Ubiquitous Computing*, pages 63–72, Zurich, Switzerland, 2013.

[53] Florian Scharf, Christian Wolters, Michael Herczeg, and Jörg Cassens. Cross-Device Interaction : Definition, Taxonomy and Application. In *Proceedings of AMBIENT 2013, Conference on Ambient Computing, Applications, Services and Technologies*, pages 35–41, Porto, Portugal, 2013.

[54] Mario Schreiner, Roman Rädle, Hans-Christian Jetter, and Harald Reiterer. Connichiwa: A Framework for Cross-Device Web Applications. In *Proceedings of CHI EA 2015, ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2163–2168, Seoul, Republic of Korea, 2015.

[55] Mike Shaw. Cross Channel Measurement - Understanding Consumer Behaviour Across Multiple Devices - comScore, Inc. `http://www.comscore.com/Insights/Presentations-and-Whitepapers/`

`2014/Cross-Channel-Measurement`, october 2014. Retrieved on October 23,2016.

[56] Kathy Sierra and Bert Bates. *Head First Java.* O'Reilly Media, Inc., 2005.

[57] B. Signer and M. C. Norrie. A Framework for Developing Pervasive Cross-Media Applications Based on Physical Hypermedia and Active Components. In *Proceedings of ICPCA 2008, Conference on Pervasive Computing and Applications*, pages 564–569, Oct 2008.

[58] Beat Signer and Moira C. Norrie. PaperPoint: A Paper-Based Presentation and Interactive Paper Prototyping Tool. In *Proceedings of TEI 2007, ACM Conference on Tangible and Embedded Interaction*, pages 57–64, Baton Rouge, Louisiana, 2007.

[59] Beat Signer and Moira C. Norrie. *Active Components as a Method for Coupling Data and Services – A Database-Driven Application Development Process*, pages 59–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[60] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2nd edition, 2002.

[61] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31, 2010.

[62] Lucia Terrenghi, Aaron Quigley, and Alan Dix. A Taxonomy for and Analysis of Multi-Person-Display Ecosystems. *Personal and Ubiquitous Computing*, 13(8):583, 2009.

[63] Bradley van Tonder and Janet Wesson. *IntelliTilt: An Enhanced Tilt Interaction Technique for Mobile Map-Based Applications*, pages 505–523. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[64] Chris Vandervelpen, Geert Vanderhulst, Kris Luyten, and Karin Coninx. *Light-Weight Distributed Web Interfaces: Preparing the Web for Heterogeneous Environments*, pages 197–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[65] Mark Weiser. The Computer for the 21st Century. *Scientific american*, 265(3):94–104, 1991.

[66] Jishuo Yang and Daniel Wigdor. Panelrama: Enabling Easy Specification of Cross-Device Web Applications. In *Proceedings of CHI 2014, ACM Conference on Human Factors in Computing Systems*, pages 2783–2792, Toronto, Ontario, Canada, 2014.

[67] John A Zachman. A Framework for Information Systems Architecture. *IBM systems journal*, 26(3):276–292, 1987.