Graduation thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

A Framework for the Rapid Prototyping of Cross-Device Interaction

KEVIN VAN RYCKEGEM

Academic year 2017 - 2018

Promoter: Prof. Dr. Beat Signer
Advisor: Audrey Sanctorum

Faculty of Science

Abstract

The purpose of this research, is to investigate and build a framework which would enable the rapid prototyping of cross-device applications. There are many different cross-device frameworks which focus on different aspects. However, very few of them are used in production. There are multiple probable causes for this. Some of which could include the accessibility, learning curve, user-friendliness or developer-friendliness of the framework. In this research, focus is put on the ease of use for the developer as well as the user. The framework should allow developers to easily and rapidly develop cross-device applications or websites and thus attract more developers to the cross-device applications field.

Acknowledgements

I would like to thank all the people that have contributed to this thesis. Their support and contributions are very much appreciated.

I have received valuable support during this thesis from my promoter, Prof. Dr. Beat Signer and my advisor, Audrey Sanctorum. I would like to thank them for supporting me throughout this thesis by providing helpful ideas and feedback.

I would also like to thank my family and girlfriend who have been great sources of inspiration and support.

Furthermore, I am thankful to the participants of the evaluation of the framework: Joren De Bot, Tim Leenaers, Nicolas Paradis, David Van Ryckegem and Ken Vernaillen.

Contents

1

Introduction

Nowadays people use many different devices on a daily basis. A typical household in the U.S. contains at least 3 internet-connected devices [8]. While devices can be linked and some applications can synchronise themselves across devices, very few enable cross-device user interfaces. As there are more and more devices inside and outside households, the possibilities inside the cross-device interactions domain increase. However, to our knowledge, no rapid cross-device application frameworks which offer a complete set of distribution possibilities exist at this point of time. In this type of interaction, one or many user(s) can use many different devices with different platforms together in order to create interesting applications and user interfaces [28]. This is different from the current way for example, web applications are used, since these usually assume that there is only one single user and one single device used [12]. However, in a cross-device application a situation with a single or multi-user and multi-device are assumed. Data from devices can influence other devices, or simply be reproduced on other devices (distributed and synchronised user interfaces). A distributed user interface can have one or multiple elements which are distributed, but does not require the entire interface to be distributed.

There are many different cross-device application frameworks which focus

on different aspects. For example, Panelrama [32] is focused on the ease of implementing cross-device applications by introducing the concept of distributable *panels*. Weave [5] focuses on cross-device interactions such as actions triggered by cross-device gestures. Proxywork [26] and Polychrome [2] introduce the use of proxies in cross-device application frameworks. XD-Session [20] implemented testing functionalities for cross-device application frameworks. XDBrowser [19] introduced the aspect of letting the user control the way a web page is distributed across the different devices, while XDBrowser 2.0 [18] introduced the semi-automatic distribution of web pages across devices, where a list of possibly useful defaults were given to the user and automatically applied. Frosini et al. [9] proposed and implemented a way for multiple screens to be combined into one display. Reticularspaces [3] shows us that not only web pages can be distributed, but entire environments depending on the physical location of the user can be as well. These frameworks all discussed in detail in Section 2.2. Unfortunately, very few of these cross-device application frameworks are used in production environments. There are multiple possible causes for this. Some of which could include the accessibility of the framework to developers, the learning curve for developers, the user-friendliness or developer-friendliness of the framework. In this thesis, focus is set on the ease of use for the developer as well as the user. The framework should allow developers to easily and rapidly develop cross-device applications or websites. By making this development easier to access and develop, more interest could be sparked into the cross-device applications field.

In order to solve the problems in the current domain, we introduce new classifications of cross-device frameworks, introduce new concepts such as distribution methods, in order to organise and facilitate the cross-device development and finally, we present a hybrid cross-device application framework called the HDUI Framework. This HDUI framework stands for *Hybrid Distributed User Interfaces Framework* and is our solution to solve the aforementioned problems.

1.1 Scenarios

There are many different ways cross-device (user) interactions could be used. One obvious example would be games. Many variations are possible depending on the amount of users and devices. A clear example of this would be a computer game which connects to a secondary mobile device (mobile phone

or tablet) in order to display more information (health, scores and more). While secondary devices can be used to display information, they could also be used as an input. This means that by selecting options on the mobile phone, another weapon, armour or spell could be chosen in the game. A diagram of this example can be seen in Figure 1.1.

Figure 1.1: A diagram of the game use case for cross-device interaction. The primary display would be the computer, while the tablet and mobile phone could provide additional inputs (game controls) and outputs (display of information).

A second use case would be in stores. Stores could let buyers use their phones to connect to the store's TVs and change the channel to their pleasing (although restrictions would need to be made in order to prevent abuse in such a scenario).

As a third use case, a hallway could be filled with connected displays. Once the user connects their phone to the displays, preferences from the user's phone could be used for displaying user-specific data on the displays. While the user walks down the hallway of displays towards their destination, flight statuses or directions can be shown on the display nearest to the user. Displays could be screens or even LED arrows (with clear identification of the user the arrow is related to).

In a fourth use case, teachers and students both have connected devices. Some elements of the application (questions or topics chosen by the teacher) could be automatically distributed across devices, while other elements (student inputs and corrections) would not. This would allow for teachers to change settings or questions while students have individual working parts inside the application. Students could send their answers to the teacher, by distributing these to the teacher's device and/or by saving these into a database, while automatically getting an answer sheet. A variation to this use case would be where a teacher changes the topic shown to students, while students can then individually click sub-topics for more information during

lectures.

Nebeling et al. [19] categorise some possible uses of cross-device applications into 3 categories.

1. The usage of multiple displays in order to create one bigger screen.
2. Allowing input from a device to output something onto, or manipulate another device.
3. The use of one particular device in order to manipulate others remotely.

As a final concept, cross-device applications can provide a way of input on display-only devices (such as non-touch screen displays). Users would then be able to use their own devices as a way of input.

1.2 Justification

We have chosen to research, prototype and develop a rapid hybrid cross-device application framework for this thesis. This section explains why we have chosen to set the focus on a hybrid cross-device framework and developer-friendliness, such as rapid development. There are many other aspects in this thesis, such as the newly introduced distribution methods concept and protocol-independence, but these will be discussed in further sections.

Most cross-device frameworks are developed using a web-based or native technology. However, these are not the only possible solutions. To our knowledge, no research articles can be found about the usage of hybrid frameworks in order to develop a cross-device application or application framework. Continuing in this direction is interesting since hybrid frameworks are device agnostic, while they can still provide true native features, which web frameworks cannot (some, but not all, may be possible through web APIs). Device agnosticism, the ability of an application to work with multiple platforms using the same code base, without requiring extra changes, is especially interesting in the cross-device user interaction domain, as this means that there would be less work for the developers compared to a native technology. Compared to a web-based technology, advantages such as accessibility to the devices' capabilities emerge.

As previously mentioned, many current cross-device user interaction applications are developed using a web platform such as HTML [32, 26, 19]. By

doing so, native access to the device's functionality is lost. While HTML5 does contain many APIs in order to access device functionality, native apps still have better access to a device's features. For example, a mobile phone's contact list cannot be accessed using HTML5 (work on this feature has been discontinued due to the amount of obstacles) [27], while this is possible using native apps.

In contrast, native cross-device user interaction frameworks have some advantages, but the disadvantages outweigh the advantages. While these do have access to all of the features of a device, they require a different codebase per platform. Since cross-device user interactions need to support multiple platforms, this means that the developer needs to develop another native application per platform. This makes the development of cross-device user interaction applications slower, more difficult and more costly.

The aim of this thesis is to research the advantages and capabilities of hybrid frameworks in the context of cross-device applications and doing so while improving elements that are lacking in existing cross-device application frameworks. For instance, the focus is set on both user- and developer-friendliness. The framework should facilitate developers into rapidly creating cross-device applications. In order to do so, possibilities and concepts will need to be researched in more detail. Some frameworks do focus on ease of use for developers [5], but do so while introducing new tools for the developer to learn [5, 21], or have too minimalistic functionality [32]. However, this research is aimed at not introducing new tools, in order to keep the learning curve low for developers while still introducing an adequate amount of functionality into the framework, enabling the developers to implement exactly what they intended to.

1.3 Problem Statement

At this point of time, cross-device development is not popular due to a variety of reasons, such as the accessibility, difficulty and the amount of time it takes to develop one. We want to spark more interest and enable developers to take a deeper dive into this development area. The problem is that cross-device development is usually too complex, not only for the development itself, but for the debugging and testing also. Solutions need to be found to these problems in order to allow this research field to grow and become more mature so that it could eventually reach end-users. This is especially true since there are so few cross-device applications actually in use, nowadays.

Many different cross-device frameworks are emerging, but none of these are taking the ease of development into account while also providing enough functionality to the developers. It is not only the ease of development, but the achieved functionality that matters. Concepts need to be introduced in order to avoid the continuing problem of time-consuming and difficult to develop cross-device application frameworks that do not make use of a structured manner for developers to implement cross-device functionalities.

To our knowledge, there are no existing cross-device application frameworks that were developed using hybrid frameworks. For the reasons explained in Section 1.2, the usage and further research of hybrid frameworks for cross-device applications would be advantageous to the area of cross-device applications. Our goal is to create a cross-device application framework using a hybrid framework and re-evaluate the advantages or disadvantages related to the usage of a hybrid framework in this research area. A second goal is to make this framework as simple and easy to develop as possible, while keeping important cross-device functionalities. We want to analyse, research and introduce concepts on how a cross-device application framework can be made easier and faster to develop.

Another problem is that most of the frameworks are focused on the making of a functional system, that not enough time is spent on the user- and developer- friendliness. While it is important that the system is functional, user- and developer- friendliness cannot be left out of the equation, as these are important. When a framework is not user- or developer-friendly, few people will want to use it, causing current cross-device application frameworks to not be used by either developers or end-users. In turn, this causes other problems such as the need for better debugging tools for developers.

There are two main goals in this research.

1. Find new ways to improve developer-friendliness of cross-device application frameworks. 2. Experiment with hybrid frameworks for developing a cross-device application framework and re-evaluate the possibilities of hybrid frameworks for cross-device development.

While working on these goals, some other innovative features are introduced and tested as well. These include, but are not limited to, protocol-independence, distribution methods and device capability distribution.

1.4 Thesis Structure

The thesis is structured into the following chapters. First, an introduction to the subject and research is given, while introducing the reader to the different terminology and technologies. Afterwards a literature review is available in the background chapter, discussing the state of the art and its current limitations. Research is then conducted in order to introduce new concepts and prototype a potential solution. The implementation of the framework and its details are then discussed in the implementation chapter. A separate chapter demonstrates how the framework could be used along with some example applications that were implemented. An evaluation experiment is also conducted and discussed in the evaluation chapter. We then report our results and conclusions in the conclusion chapter, while also explaining the possibilities of future works.

2

Background

Cross-device applications work by connecting and using multiple devices' inputs and/or outputs for one or multiple users. Since there are many different types of devices and screens, having cross-device applications allow users to benefit from the advantages of the different devices combined in a seamless manner, with their respective capabilities. Furthermore, the availability of devices is growing [26], while the amount of connected devices per US household is rapidly and steadily increasing [8]. This allows for more opportunities within the cross-device interaction area.

Currently existing systems and research is further discussed in this chapter, to understand where the current cross-device application technology is at and improve on the existing systems. Before discussing the existing systems, some basic cross-device application concepts are explained.

2.1 Concepts

Gallud et al. [10] describe some of the important concepts and issues concerning distributed user interfaces by referring to the DUI concepts made by Terrenghi et al [25]. Some of these concepts which will also be used in

this thesis to describe systems include: *multi-device usage* (using multiple devices for one purpose), *multi-platform usage* (the use of different computing platforms), *multi-monitor usage* (usage of multiple monitors across one device), *multi-display usage* (the usage of multiple monitors from different devices; otherwise called displays) and the *multi-user usage* (multiple users interacting with one or more devices). Monitors are said to be bound to one device, such that a device could have multiple monitors. A combination of two devices with one monitor each is said to be a two displays set-up. A cross-device application is typically able to be run on multiple monitors, multiple devices and be cross-platform, as can be noted from the following State of The Art section in 2.2.

The previously explained multi-user aspect could be extended, since it is not only interesting to be able to support multiple users, but there are also many different ways for implementing this. For example, one UI component could be only shown to the device of a user that is authenticated for that component, introducing the new aspect of multi-user *-security*, *-authentication* and even *-privacy* into the context of cross-device applications [32]. It is possible to make a distributed user interface in a manner that it is user-centered and only shares specific screens to specific users. Other multi-user DUI concepts are the distribution of UI elements without interfering with other user's activities, or by delegating tasks to other users, as stated by Gallud et al.

2.2 State of The Art

In this section, different cross-device application frameworks are discussed and compared in order to demonstrate the different ways these frameworks are currently implemented, along with their features, advantages, limitations and possible improvements.

Frosini et al. [9] created a framework for the distribution of user interfaces which allows developers to build applications with specific UI elements that can be distributed and shown on different displays. This framework consists of two parts, a library and runtime engine support. The runtime engine support allows devices to link and maintain a connection with each other, for distributing UI elements. This can be done peer to peer or by using an extra standalone server in between. The library contains the necessary methods for requesting actions between devices. The framework then facilitates connection and communication between the devices in a session. The framework was implemented using Java and is able to run on desktop platforms and An-

Figure 2.1: The museum application implemented using the Frosini et al. [9] framework.

droid devices. The authors made an example application for museums using the DUI framework, as can be seen in Figure 2.1. This example application allows users to scan a QR code using their phone. This QR code identifies the device to connect to and is unique per device. This QR code is available for scanning on the display of the device itself. After scanning the QR code, the phone has an interface allowing the user to control the scanned device. The user can then for example, select an item on their phone, resulting in the interactive item being shown on the scanned display. While the the manner for connecting devices one to another (via QR code) may be great for a museum application, it can be problematic in different settings. Since the QR code is shown on the screen of the device a user wants to connect to, the user of the device to connect to will need to be interrupted in order to show the QR code to new users wishing to join the session (in a multi-user setting). However, in the case of a museum application, the QR could be printed next to the displays, solving this problem. The way a cross-device application provides the session connectivity thus depends on the setting the application is used in. An architectural advantage of this framework is that it works by sending a *UI state* instead of *UIDL* (UI Description Language). This means that the UI state received is not bound to a specific UI format or UIDL.

For instance, instead of sending pure HTML for distribution, the framework sends a message (UI state) that a client can understand and interpret in order to distribute the correct item, without the use of any UI description language. UI update commands in this framework are structured and sent using an XML syntax. More research is done concerning the use of UI states or UIDL as a way of solving DUI communication in Section 3.4.1.

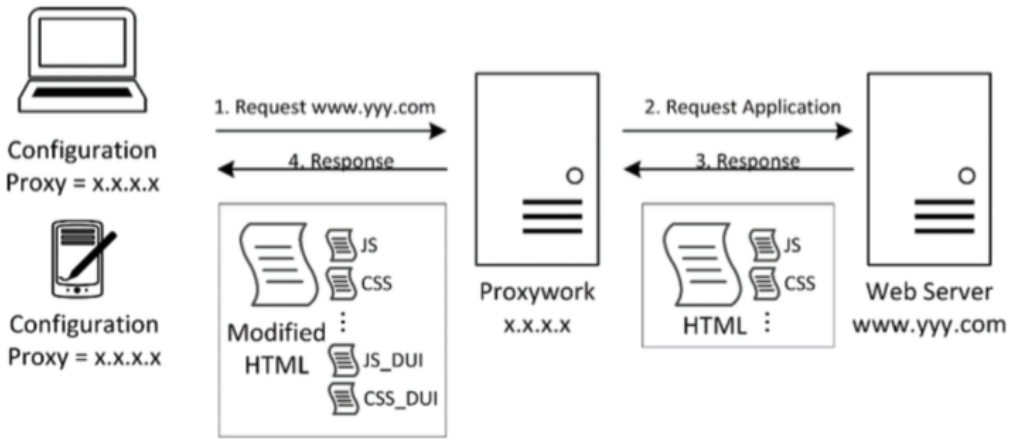

Figure 2.2: Web page processing using the Proxywork [26] cross-device system.

Another cross-device application framework called the *Proxywork system* [26] works differently than the previously mentioned Frosini et al. framework [9]. One major difference is the fact that the Proxywork distributes and modifies UIDL instead of communicating to other devices using UI state as in the Frosini et al. framework. New cross-device terms are also introduced in the research in order to distinguish the different user interface distribution states and capabilities. The Proxywork system enables users to distribute any web component of any web page over multiple devices. It is doing so by extracting and processing a specific piece of HTML from the webpage and distributing it to other devices. As shown in Figure 2.2, the Proxywork system needs to be run on a standalone server located between the clients and the actual server to be reached. This server is called a proxy (all traffic has to go through this server, for the DUI framework to work correctly), hence the name of the system. The proxy modifies the HTML pages before they are sent back to the clients, in order to add the DUI abilities to the website (such as a menu and features). The system implements some primitive methods for DUIs: the connect/disconnect, rename, copy, clone and migrate methods. These methods allow the user to distribute an application at a very general level.

However, this system does have limitations as well. The HTML needs to be perfectly structured and the system currently does not support the HTTPS protocol. The authors also introduced an authoring tool for manipulating the interface distribution models, since the Proxywork system is based on a metamodel introduced by Tesoriero. Tesoriero also asked an interesting question: *Are we really sharing the interface?*. This question is further discussed and distribution methods are proposed in Section 3.4.1.

In another research, Villanueva et al. [29] evaluated the Proxywork system. The evaluation consisted of 5 people, who were asked to perform a specific task with and without the use of the Proxywork system, after a training session. The task consisted of opening a specific Youtube video on multiple devices. With the Proxywork system, one could use the clone primitive method in order to solve this task. The result of the evaluation shows 95% confidence that Proxywork makes the assigned task faster to solve than without using it. However, the evaluation mostly consists of a repetitive tasks over multiple devices. Without this repetition, the authors would not be able to generate such a result.

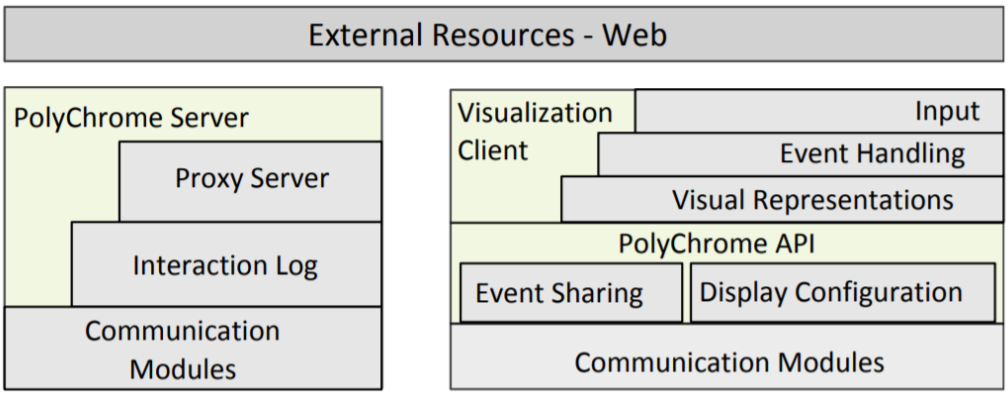

Figure 2.3: The architecture of the Polychrome framework [2]

Proxywork is not the only cross-device application framework which uses a proxy in the system's architecture. Polychrome [2] is another cross-device application framework which utilises a proxy-based architecture. It is a web-based cross-device application framework with the objective to allow users to collaboratively create visualisations that span over multiple devices. Polychrome is based on a similar architecture as the Proxywork framework, meaning that the contents are filtered and modified through a proxy server, which takes care of the distributing of the different parts of an application

towards the different client devices. This can also be seen from the architecture diagram in Figure 2.3. Badam et al. [2] also define two interesting concepts concerning distribution and synchronisation, implicit and explicit sharing. *Explicit sharing* happens when a user decides that a particular piece of content should be distributed over multiple devices and performs an action in order to make this happen. *Implicit sharing* is said to be the case when for example multiple devices are linked in a synchronised web page, meaning that doing anything on one device will automatically be done on the other, implicitly. That is, the user does not have to decide or explicitly perform an action for this synchronisation of distribution to occur. Polychrome is publicly available to developers[1].

Melchior et al. [17] however focus on the platform-independence of UIDL (user interface description language) and develop a model-based DUI framework where the different UIDLs for different platforms are automatically translated in order to make the UIDLs platform-independent. They explain this using the concept of CUIs (Concrete User Interfaces), which enable a DUI framework to be developed using a platform-independent UI language. A CUI is a model that is independent from its platform and the respective UIDL. In this framework, UI components are distributed between different platforms (Microsoft Windows, Mac OS X, Linux and Android). Therefore, in order to distribute the UI components between the different frameworks, a UI parser would need to be written to parse the UIDL for the specific platform. Every platform that uses a different UI language will require such a parser (or CUI model) that can understand and render the given UIDL. For the implementation of this model-based DUI framework, the application is divided into two parts: a proxy for keeping the state of the application and a renderer for the displaying of the UI. A common problem in the development of cross-device applications is that UIDL can often not be shared from one platform to others. Therefore, Melchior et al. introduced the model of CUIs in order to solve this exact problem. The translators of UIDL are thus able to translate and display the UIDL on the different systems. Yang et al. (authors of another cross-device application framework: Panelrama) however state that while such a model-based approach does have advantages such as the programming language-independence, it does have its disadvantages as well. Yang et al. state that such an approach is based on the idea of the system having a lot of platform-dependent code, which would cost some of the flexibility of the framework. It also means that the framework cannot be device agnostic (meaning that the application will require changes for it to work on the different platforms, as opposed to being able to be run

[1] `https://github.com/karthikbadam/PolyChrome`

cross-platform without requiring any changes).

XD-MVC, introduced by Husmann et al. [13] is another interesting cross-device application framework. It is web-based and focuses on being compatible with the current ways of programming web applications. The authors state that the reason few cross-device applications are available to end-users is due to the fact that few tools are available for developing cross-device applications for developers. XD-MVC is a light-weight system that does not require any installation, but can simply be run from the web browser of a device. Distribution works using specific queries, where developers or users can choose to distribute items locally or towards other devices. XD-Browser introduces user roles which are considered to be tags of information attached to specific devices. Since it is based on a peer-to-peer communication (when available), information such as user roles need to be propagated towards all other devices when changes occur. XD-MVC is publicly available to developers[2] and could technically be used in order to develop cross-device applications.

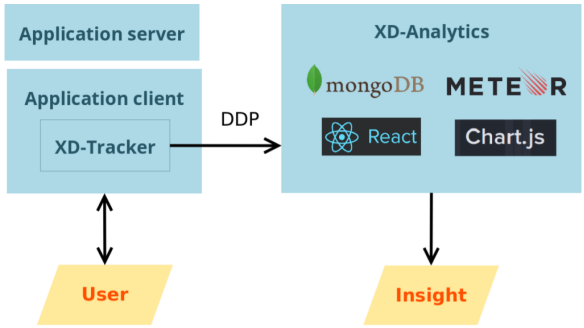

Figure 2.4: The information flow when tracking user behaviour using XD-Analytics [14].

In the research of XD-MVC, Husmann et al. state that too few tools are available for developers interested in the cross-device applications field and users are not yet fully understood [13]. In order to understand more and solve this problem, the authors also introduce a new tool that analyses the way users use the cross-device applications, called XDAnalytics [14]. This tool would allow us to better understand what users need and how cross-device application frameworks could be improved accordingly. The XDAnalytics tool allows an analyst to build charts from actions or from the specified

[2] `https://github.com/mhusm/XD-MVC`

variables available in the framework. In order to allow the clients' data to be tracked, a tracker has to be integrated into the client application. This forwards the tracking data to the XDAnalytics server, which in turn provides insight to an analyst. This flow of information is demonstrated in Figure 2.4. This tool is also publicly released and available to be used by developers or researchers[3].

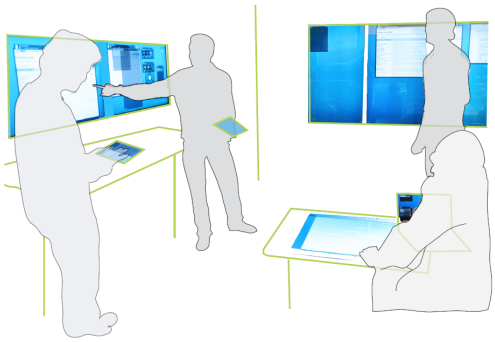

Figure 2.5: Distributing applications across devices using the ReticularSpaces [3] system.

While previously discussed frameworks can distribute parts of the user interface, they cannot distribute entire applications. Bardram et al. [3] developed an early (2012) cross-device interaction solution called ReticularSpaces which does have the ability to distribute applications and documents as well. This can be seen in Figure 2.5. The system allows for combining different displays into one display, moving transferring information, messages across a session and tracking location of users in order to split or merge the user into a different session. The users' locations are tracked using RFID tags and readers. However, this does add an extra requirement. An RFID reader must be present in the different locations where the cross-device sessions are to be created, otherwise this location-tracking session feature would not function. The authors do not mention the cross-platform compatibility of the framework, however by analysing the technologies used, a conclusion can be made. ReticularSpaces' architecture exists out of two parts. An activity manager, which manages the data and logic, and the ReticUI, which runs on the client devices. The activity manager is implemented in Java (using the Aexo distributed event system [4]). However, since the activity manager is not used on the client side, this is generally not a problem, unless the activity manager were to be created inside one of the client devices. Eliminating the

[3] `https://github.com/nmaro/xd-analytics`

requirement of an extra activity manager device could be interesting, as it would remove the need of having an extra device in the network in order for the cross-device application to function. However, this could impact performance depending on the specifics of the client devices involved and was not implemented as such in the ReticularSpaces system. Another part of the system, the ReticUI, which is run on client devices is implemented in Adobe Flex Air. Adobe Flex Air is compatible with Windows, Mac OS, Android, iOS and Blackberry Tablet OS[4]. This allows the framework to run on a multitude of devices. However, compared to other frameworks which can be run from a web browser [26, 32, 20], ReticularSpaces does require an installation of the system before being able to be used by its users.

Figure 2.6: Combined screens of multiple devices using the Connichiwa framework [24].

The previously discussed frameworks provide basic DUI functionality for distributing UI components onto multiple screens using different ways of implementation. The Connichiwa framework [24] builds on top of the existing frameworks in order to provide more functionality and cross-device interaction possibilities. An example of the added functionality is the feature to combine multiple displays into one using the spatial relations of the different devices, as seen in Figure 2.6. For example, a video player application made using the Connichiwa cross-device web applications framework allows the user to use multiple devices' displays in order to form one bigger display for playing videos across the different devices. In order to synchronize the device's positions for forming an entire screen out of different devices, Connichiwa introduces the concept of *stitching*. The user performs a specific gesture (such as *pinching*) on two devices at the same time in order for the devices to recognise their relative positions. However, the Connichiwa frame-

[4] `http://www.adobe.com/products/air/faq.html`

work also has limitations. There are memory limits which need to be solved in the future. Besides this problem, the framework only fully supports Mac OS X and iOS devices. Other devices can also use the framework, but not at its full potential. The connection setup, ad hoc connectivity and automatic device detection does not work on other platforms and needs to be ported. Connichiwa is publicly available to developers[5].

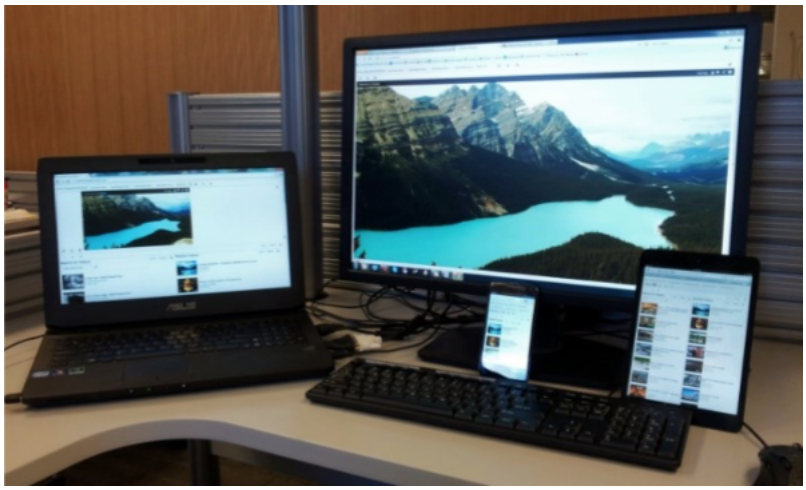

Figure 2.7: A YouTube application supported across multiple devices, by using the Panelrama framework [32].

While previously discussed frameworks did not mention anything about developer-friendliness, some cross-device researchers do mention the need of developing a cross-device application framework which is easy to implement for developers. Panelrama [32] is a cross-device application framework which is focused on being easy to implement for developers into existing web applications. The specific feature of Panelrama consists of the ability to divide parts of a web application into *panels* which can then be distributed across devices. Each of these panels can be distributed across devices, as shown in Figure 2.7. Just like the Proxywork system [26], Panelrama is an HTML5 framework, which means that it is platform-independent. Contrarily to the model-based approach such as the one presented by Melchior et al., Yang et al. choose not to implement the cross-device framework using a model-based approach. Yang et al. do state that a model-based approach has advantages such as the independence of programming language. However, they conclude that the implementations of such an approach are relying on a lot of platform-specific code (for rendering the platform-specific UIDL), which they do not

[5] `https://github.com/BlackWolf/Connichiwa`

want to use for Panelrama in order not to lose any of the flexibility of the framework. One of the applications made using Panelrama is a video streaming application which allows the user to distribute the different elements of the application over the different devices. In this case, it is possible to put the video player on one device, while distributing both the controls and the suggested videos onto other devices. Panelrama works by allowing developers to implement extra Panelrama HTML tags (such as the <panel> tag) inside their current web applications. Each panel section can then be distributed onto other devices. The developer however, does need to provide panel definitions in order to keep the state of the panels synchronised across devices. Such a panel definition consists of a selection of state information to be synchronized, a rating for automatic panel distribution across device screens and business logic for interaction with state information. The state is kept on a standalone server using MongoDB[6] (since Panelrama uses a client-server architecture). It is an advantage that Panelrama works using HTML5, since most devices already include some sort of web browser, which makes HTML5 a platform that can easily be used for cross-device applications, with a single codebase. Hybrid app frameworks such as the Ionic Framework also make use of HTML5 for the development of cross-platform applications. A limitation of Panelrama is that it is not possible to detect a device's characteristics yet. Because of this, it is not possible to automatically allocate a specific panel to the most convenient device for that panel. In order to make this work, Panelrama requires the user to input what kind of device it is at start-up of the application. Yang et al. also state that there are no features for multi-user user-specific panels with security. For example, one panel with a user logged in, should only belong to that specific user, while another panel with credentials should only belong to another user. The authors state that they will research more about direct connections between devices in order to improve latency, since the current Panelrama implementation makes use of a standalone server. A limitation that appears with the use of HTML5 such as in Panelrama, is that there is more limited access to the device's hardware capabilities and data [31]. While some HTML5 APIs are trying to solve this demand, the limitation still exists. Using a framework for Hybrid apps (such as Cordova, using a combination of both HTML and Javascript[7]) can solve this problem by introducing an extra layer in-between the app and the web browser, called a *native container*, which handles requests to these device-specific sensors. On the other hand, web apps do not require an installation, while hybrid apps do generally require an app to be installed.

[6] `https://mongodb.com`

[7] `https://cordva.apache.org`

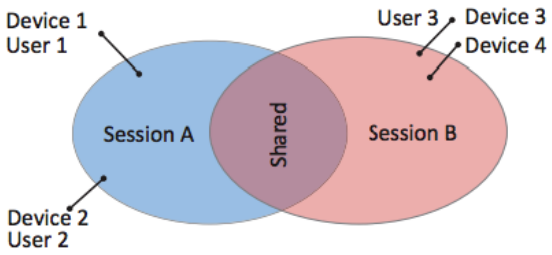

Figure 2.8: The session concept used in XDSession [20].

Panelrama is not the only HTML5 cross-device framework. XDSession, presented by Nebeling et al. [20] is a cross-device application framework focusing on the testing and debugging of cross-device session. The authors noticed that there was little research about cross-device application development. Furthermore, there were few testing frameworks for cross-device applications. XDSession was developed in order to solve this problem. A particularity about XDSession, is that it is based on the idea of synchronising *sessions* across devices. Nebeling et al. did some previous research on the development of a session-based cross-device framework in 2013 [22]. They explain that a session should contain three elements. A user, a device and information. This concept is used for creating operations (such as duplicate, join and fork). An example of how cross-device sessions works in XDSessions can be seen in Figure 2.8. Next to the XDSession framework, two visual tools are provided. The first one being a *session controller*, which allows the developer to visualise and manage the different sessions in a cross-device application. The second tool is a *session inspector*, which visualises the data and metadata linked to a session, as seen in Figure 2.9.

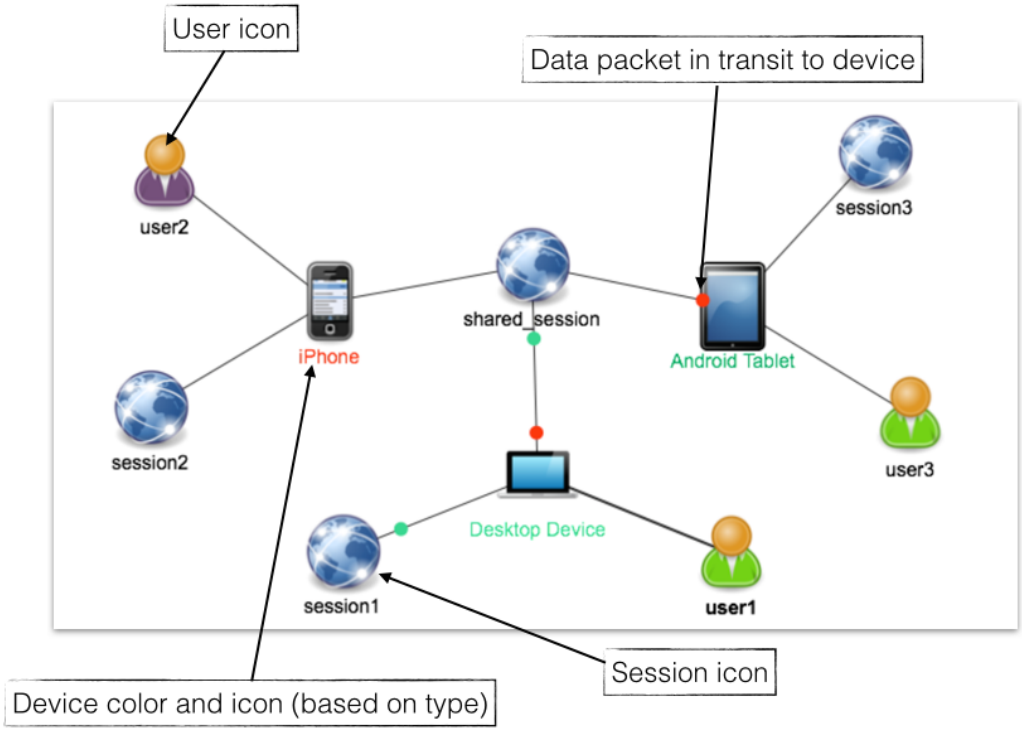

Figure 2.9: Session Inspector visual tool in XDSession [20] framework.

The combination of the framework and all these tools allow for a controlled manner for developing cross-device session applications. XDSession consists of one or more clients and a server. The server is developed in Java for managing the data, the information and UI distribution. The client application is developed using HTML5 and Typescript. SocketIO is used in order to maintain an active connection between the server and the clients.

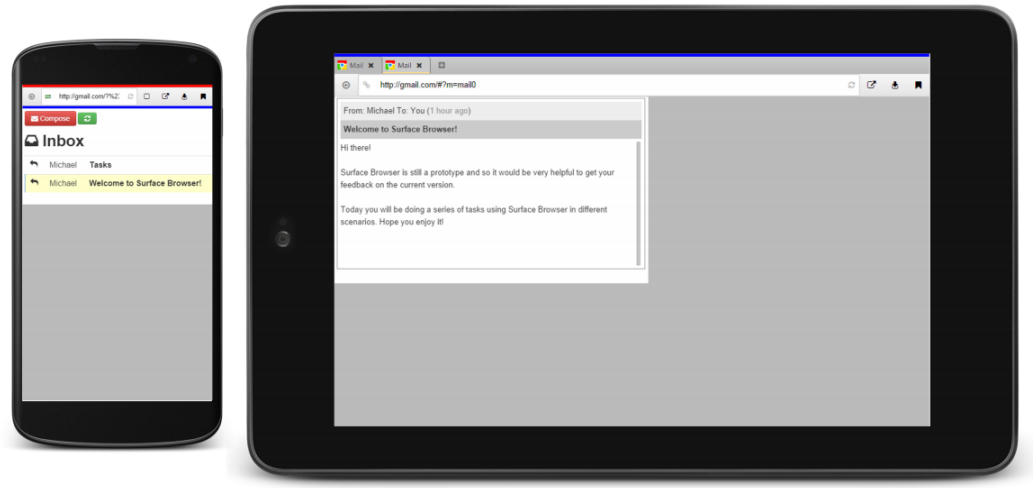

Figure 2.10: XDBrowser [19] dividing the user interface on a phone and a tablet.

In 2016, Nebeling et al. continued development in the area of cross-device applications and developed XDBrowser. XDBrowser is a cross-device web browser application that runs on multiple platforms [19]. XDBrowser is built to run on any default web browser of devices. It is developed using HTML5, CSS3, Javascript, Socket.IO and Node.JS. It enables users to distribute any part of a website as they wish, between the different available browser tabs on different devices, as seen in Figure 2.10. While Panelrama requires developers to add panel tags into the HTML, XDBrowser does not require any modification to the webpage and can distribute any regular website. It works by using iframes to display content, acting as a web application that can be run in the default browser of the operating system. The client side of XDBrowser consists of two parts. The first part consists of browser windows. The browser windows have the ability to navigate the web browser (using iframes). The second part consists of selection boxes. These are used in order to let the user choose what to display on other devices (for dividing and distributing the user interface across multiple devices). There are some limitations linked to using iframes. While iframes work well for local websites, Nebeling et al. note that some websites block the embedding of the website in iframes. Next to this problem, browsers block cross-site scripting for security measures. The authors proposed to solve both these problems by using a proxy server. Another limitation is the authentication. Users currently need to sign in on every device when using the cross-device browser, as authentication sessions are not kept across devices. Depending on the

application, this might not be the intended behaviour, with further research required about security of multi-user cross-device applications. The results of a user study of XDBrowser reveal the top preferred design patterns of use with XDBrowser. There were 15 participants for the user study. The top 1 chosen pattern was the remote control pattern, in which one device can act as a remote control of another. The top 2 pattern was the overview + detail pattern wherein one device provides an overview while another provides the detail view of the application, controlled by the overview device. One user stated that the controls for synchronising different devices could have been more sophisticated.

In 2017, Nebeling et al. introduced XDBrowser 2.0, which is an improvement on XDBrowser [18]. It adds support for semi-automatic cross-device design, i.e., the semi-automatic distribution of UI across devices (or tabs across devices in XDBrowser's case). It is not fully automatic, as the user is still required to choose a specific cross-device design layout (however, one could be set as default for new webpages). For example, *overview+detail* or *remote-control*. This would remove the need of the user having to manually distribute a UI across the tabs before being able to use the website. Previously, cross-device applications require the user to distribute the UI, or the developer to annotate and implement the distributed UI design.

Nebeling et al. also developed a cross-device user interfaces GUI builder for the development of cross-device applications, before XDBrowser was developed. This tool is called XDStudio [21]. A cross-device applications framework is used behind the scenes of the GUI builder. It is built using NodeJS, JQuery and HTML5. XDStudio is focused on the development aspect of the cross-device applications domain.

Another cross-device interaction framework, Weave, presented by Chi et al. [5] decided to also build an HTML cross-device framework, while focusing on providing features that were missing in the current cross-device appplication frameworks. The authors noticed that the existing frameworks do not have high-level abstractions for cross-device interactions. An example of such a cross-device interaction is the ability to trigger specific actions upon the completion of a cross-device gesture (such as two or more devices shaking at the same time). The authors propose to solve this problem by introducing the Weave framework. Weave [5] is a cross-device interaction framework focusing on providing a high-level abstraction for cross-device gestures and interactions. Weave is based on a combination of NodeJS, JavaScript and HTML. The Weave framework works by keeping state in a central server. Devices then implement logic for the given state in the central server. One

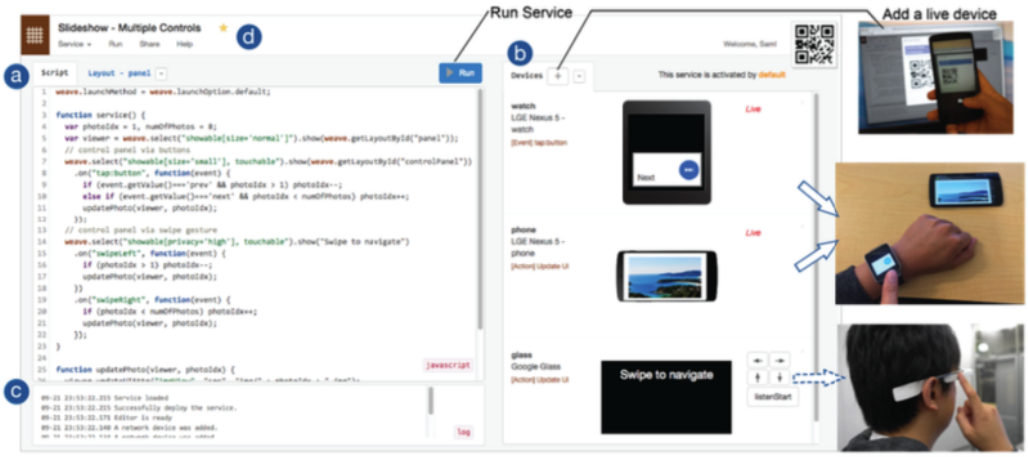

Figure 2.11: The Weave [5] development tool and a demonstration of an application implemented using Weave.

possible use of the Weave application, is the implementation of a cross-device photo gallery. One device can change the image shown on another device. This is done by keeping a variable with the photo (index) to be shown on the other device(s). The other devices are subscribed to a service that will change the image on the device whenever the previous or next actions are triggered. Similar to the Frosini et al. framework [9], UI components are not communicated directly between devices. Instead, the UI state is communicated (such as photo index of the photo to be shown on the device). This method avoids problems such as relying on the syntax of a specific UI platform (for example relying on HTML UIDL, when other devices use another UIDL) and the inflexibility of relying on platform-specific code as described by Yang et al [32]. Weave particularly focuses on allowing the synchronisation of cross-device interactions and events. The high-level abstraction provided by Weave allows developers to easily intercept events such as shaking or pinching on multiple devices at the same time. Weave is one of the only cross-device application frameworks that features a web-based development environment. The framework also collects the devices' capabilities using a Weave service, which is running on the specific devices. This service handles the task of detecting a device's capabilities. By collecting the capabilities of each device, it is possible to find and select devices with specific capabilities for running a specific task. For example, one specific information could be distributed only to devices that have a Bluetooth or Wi-Fi capability. This is made possible using the abstractions introduced by Weave. There are some limitations and future work as well for the Weave framework. For instance,

Weave does not implement any way to track the spatial relations between devices, while other frameworks such as Connichiwa [24] do provide such a feature (which they use in order to create combined screens). Since Weave does not support spatial relations between devices, it is also not possible to provide combined screens (putting devices next to each other in order to create one bigger screen). Weave is also only shown on Android devices and does not provide any platform continuity features, which can be found in frameworks such as the Ionic framework. The Ionic framework and its platform continuity are further discussed in Section 3.3.5. The authors of Weave do not explain details about the linking of different devices, for the devices to work under the same session. Privacy and security issues are not explained as well. However, they state that gestures such as shaking devices at the same time could be implemented and used to add the devices into an active session.

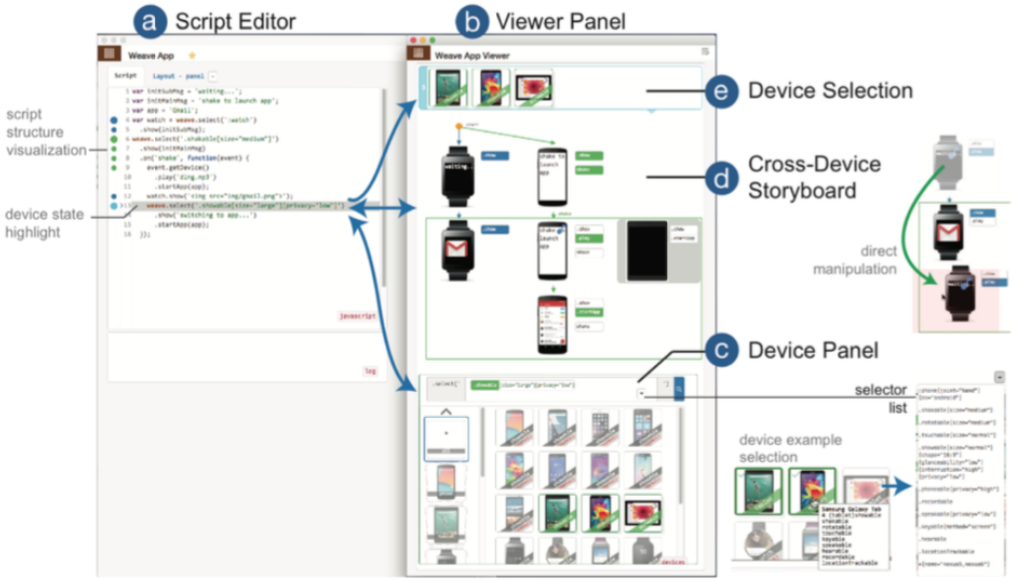

Figure 2.12: The Demoscript [6] editor, for the Weave framework.

DemoScript [6] is an addition to the Weave framework, developed in 2016. DemoScript adds the ability to use and make storyboards for developing cross-device applications to the Weave framework. This can be seen in Figure 2.12. Developers can then *storyboard* the intended behaviour of the cross-device application. Chi et al. state that developing applications across multiple devices can be difficult and that DemoScript is their solution to improve the development efficiency of cross-device applications.

2.3 Testing and Debugging

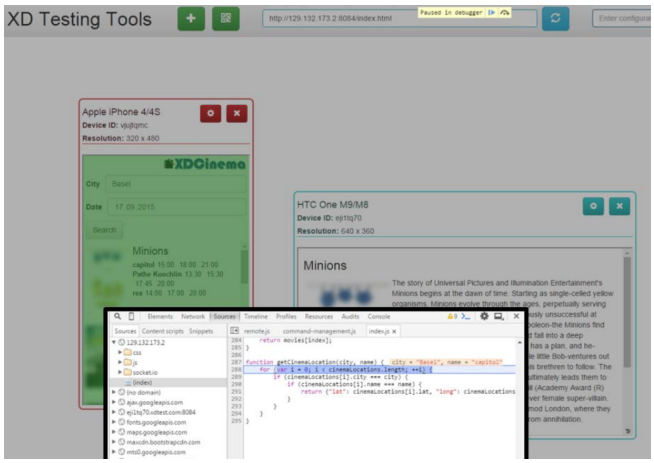

Figure 2.13: XDTools [11] emulating virtual devices within Chrome, while debugging an application with breakpoints.

While there are many different frameworks, Husmann et al. explain that there are only a few of these frameworks that have applications implemented [11]. Next to this, most of the implemented cross-device applications were implemented as part of a demonstration of the framework. Some of these applications are not released to the public either. The authors explain that there is a lack of testing and debugging tools for cross-device applications. This is why XDTools was developed. XDTools is a debugging and testing tool for cross-device applications. XDTools is a web-based application (as can be seen in Figure 2.13), but in order to use all of its features, Google Chrome has to be used because Google Chrome offers extensive support for emulating devices. In order to test or debug the cross-device applications, XDTools provide a way to create virtual devices inside the browser. XDTools also extends the emulation support in Google Chrome. By default, it is only possible to emulate one device in one browser tab, while XDTools allows the emulation of multiple devices in one tab with the use of iframes. The iframes have the ability to communicate with each other if needed for the cross-device application (in case a standalone server is not used and the cross-device application would function in a direct connection manner).

Husmann et al. conclude that some of the current problems for cross-device frameworks are that a framework itself is not enough. An environment for testing and debugging was part of the missing components for cross-device

development and their solution, XDTools, solves that problem. Some earlier cross-device application frameworks do provide some debugging features, such as the XDSession framework by Nebeling et al [20]. However, the XDTools package is more complete [11]. The authors state that one question is still left. Will XDTools be enough to get more developers into cross-device application development? XDTools is publicly available to developers[8].

A previously discussed framework, XDSession, can also be useful in the testing and debugging area. XDsession provides tools for controlling and visualising sessions, which can also be used for debugging and testing [20]. It is possible to replay events that occurred in the session inspector and propagate them to all the related devices. When replaying sessions, it is still possible to use the session controller and inspector for having a detailed view at what is actually happening within the specific sessions and understanding why the specific behaviour is occurring.

2.4 Classifications

Sanctorum et al. [23] propose a classification for cross-device interaction frameworks based on different criteria. The classification can be seen in Figure 2.14. This classification shows the different existing approaches for the creation of cross-device applications by the development of cross-device application frameworks. The proposed classification categorises the frameworks based on location constraints, granularity and support for distribution of the state of an application. The frameworks which support the distribution of state are highlighted in bold and are blue. As can be seen from Figure 2.14, some frameworks can be used anywhere without a network connection, while others do require a network connection, which could be a local or remote connection, while a local connection would add further limitations to the framework. On the Y-axis of the diagram, we can find the granularity of the frameworks. Some frameworks are only able to distribute entire applications, while some others can distribute elements within the application, or even raw screen pixels. This is illustrated using the granularity dimension of the diagram. As we do not want to create unnecessary constraints, this research is focused on the frameworks which have a location constraint of *Network connection to the server* or *Anywhere*.

[8] `https://github.com/mhusm/XDTools`

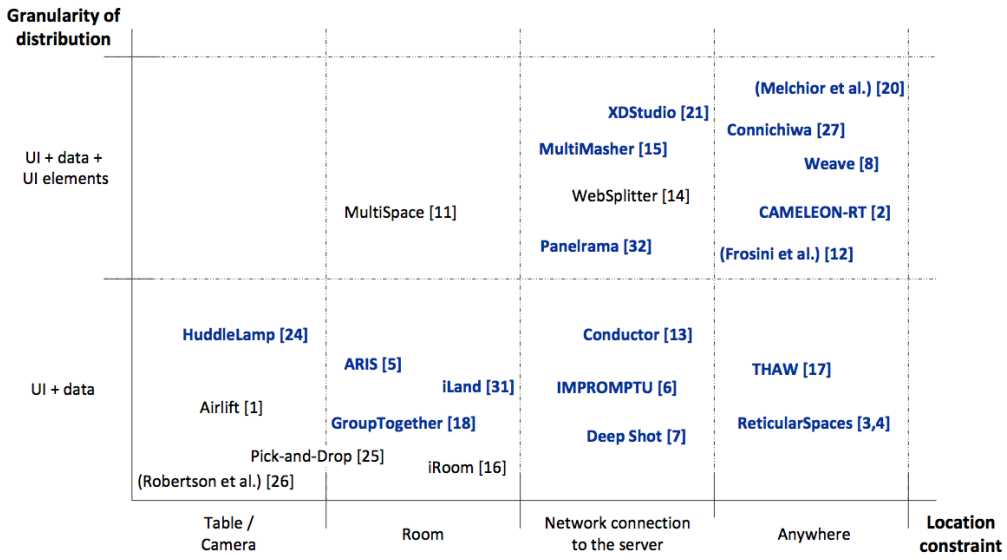

Figure 2.14: The classification of existing DUIs, by Sanctorum et al. [23].

Since this thesis is focused on the experimentation of the use of a hybrid framework in order to implement a cross-device framework, another cross-device application frameworks classification is proposed in order to illustrate the different technologies primarily used and the lack of experimentation with hybrid framework-based cross-device application frameworks in this area. The resulting classification can be found in Table 2.1. The classification contains some of the well-known cross-device application frameworks and research. In order to illustrate that relatively few cross-device application frameworks are available publicly for developers to use on their projects, frameworks that were publicly available and could potentially be used by developers are highlighted (as seen in the legend). *Non-developer accessible frameworks* could not be publicly found at the time of this research and thus could not easily be used by a developer searching for a cross-device application framework. Frameworks that are available on demand (by contacting the authors) are considered to be *non-developer accessible frameworks*, since these are not easy to find.

Technology	Frameworks, Applications or Tools
Web-Based	Weave [5] Demoscript [6] Panelrama [32] Proxywork [26] XDBrowser [19] XDBrowser 2.0 [18] XDSession [20] **XD-MVC[1] [13]** **XDAnalytics[2] (tool) [14]** **XDTools[3] (tool) [11]** **Polychrome[4] [2]**
Java-Based	Frosini et al. framework [9] ReticularSpaces [3] **Connichiwa[5]* [24]**
Platform-Dependent Technologies	Model-Based Approach by Melchior et al. [17]
Hybrid	*None to our knowledge*

Developer Accessible Framework
Non-Developer Accessible Framework

* In development

[1] `https://github.com/mhusm/XD-MVC/`
[2] `https://github.com/nmaro/xd-analytics`
[3] `https://github.com/mhusm/XDTools/`
[4] `https://github.com/karthikbadam/PolyChrome`
[5] `https://github.com/BlackWolf/Connichiwa`

Table 2.1: Classification of Cross-Device Applications by Primary Implementation Technology

2.5 Conclusion

We can conclude that the way current cross-device applications and frameworks are implemented varies a lot. Some are web-based [32, 5, 26, 12, 20, 19, 18, 6]. Others are implemented in a cross-platform compatible programming language such as Java [9, 3]. Some still lack full cross-platform compatibility [24]. It is interesting to notice that all of these current cross-device application frameworks have been using programming languages such as Web or Java in order to render the framework cross-platform using the same codebase. However, none of these have attempted to use a specialised *hybrid framework* as a base. Some frameworks are focused on the developer-simplicity, but as a result contain few features to really take control of the entire cross-device interaction space [32]. Other research such as a model-based approach for cross-device applications have been criticised to be not flexible enough, since they would still require the different codebases to be implemented for each platform and thus be less efficient for rapid development [17]. Other frameworks introduced the concept of distributing data using UI state rather than UIDL, such as the Frosini et al framework [9]. While these cross-device frameworks have introduced good concepts into this research field, the advantages and functionalities are still scattered around numerous different frameworks. Using the existing knowledge about cross-device applications to combine these different features and advantages (the use of UI state, developer-friendliness, functionality, device agnosticism and different approaches) into one powerful framework would give the developers the ability and tools to really control the cross-device interaction space. While we can find an abundance of different cross-device application frameworks, few of them are actually used in live applications. Some frameworks lack platform-compatibility and device agnosticism[24], user-friendliness (for example, platform continuity) or programmable functionality [32]. Other frameworks are simply inaccessible to the public, and hence cannot be used by developers [5, 32, 9, 3, 26, 12].

3

Research and Prototyping

We introduce the HDUI framework (Hybrid DUI framework), a cross-device application framework based on a hybrid framework, Ionic.

3.1 Framework Goals

Some problems became clear from the background and related works in Section 2.2. For instance, the research area is scattered with various different frameworks which all have their own specific advantages. This makes it difficult for developers to choose a framework, since the combined advantages are what make it interesting. Another issue is that relatively few cross-device application frameworks are actually freely and publicly available to developers, as seen in Table 2.1. In order to solve this problem, we are striving for a publicly available framework with a minimum of available documentation, allowing developers to effectively use the cross-device framework. In order to facilitate the developers' needs, the framework will need to implement a maximum of features so the developers will have less work on their hands. Next to having enough features, it should be relatively quick for developers to make a cross-device application. Since there are already cross-device application

frameworks that focus on the speed and ease of development [5, 32], this is a must-have. While these frameworks do focus on the development speed, some still lose developers' control and flexibility over the framework's functionality, such as Panelrama [32]. Developing a speedy way for developers to implement cross-device applications should not affect the functionality or control over the cross-device aspects of the framework and its applications. The development of a developer-friendly cross-device application framework should not harm the user-friendliness of its developed applications neither.

Some frameworks had to develop extra tools to allow developers to easily debug, test or develop cross-device application frameworks [20, 19, 14, 11]. In this framework, we do not want to have to rely on extra tools that need to be developed. Instead, the framework will need to be built on top of an existing technology with enough existing debugging tools and development tools, effectively removing this extra need of developer tools for cross-device applications. This would also make it easier for developers, to be able to use the tools they are already familiar with, instead of introducing many new tools which negatively impact the learning curve of developers new to the cross-device interaction framework.

Some requirements that resulted from these findings are discussed in Section 3.2.

3.2 Requirements

Requirements need to be defined which effectively comply to the goals mentioned in Section 3.1, in order to ensure that all required features and improvements are researched and implemented into the HDUI framework. After researching previously existing cross-device application frameworks, we decided to have the following main requirements for the reasons listed below in a pursuit of solving the problems described in 3.1. These functional and non-functional requirements have been adapted and improved as the design and purposes of the framework evolved. The requirements are shown in no particular order.

R1. Based on Hybrid Framework

R2. Easy Access to Native Functionality and Data

R3. Developer-Friendly (and rapid)

R4. Device Agnostic and Universal

R5. Protocol-Independent

R6. Detect Device Capabilities

R7. Extensible

R8. Minimal Set-up and Run Requirements

R9. User-Friendliness

R10. Platform Continuity

R11. Sufficient Developer Tools

R12. Sufficient Cross-Device Functionality

R13. Cross-Platform

R14. Public Availability

R1. Based on a hybrid framework. The framework should be implemented using a hybrid app framework. There are no DUI frameworks implemented using Hybrid frameworks to our knowledge and further research should be conducted in this area. As seen from our research into existing cross-device systems and the summary of DUI approaches by Sanctorum et al. [23], most frameworks contain web-based or Java implementation technologies without any use of hybrid frameworks. This can be seen from the overview and classification of cross-device application frameworks by technology in Table 2.1. As research advances, some advantages become clear such as access to the device's native capabilities (by using the native container which the hybrid framework provides), integrated platform continuity, access to existing developer tools and device agnosticism.

R2. Easy access to native functionality and data of the device. The framework should not prevent the developer from accessing a device's native capabilities such as a mobile phone's contacts or camera. Unfortunately, some frequently used technologies for cross-device application frameworks such as HTML5 do not have access to all native features of a device, such as the contacts (the W3 HTML5 feature implementation has been discontinued [27]).

R3. Rapid development. The framework should allow the developers to rapidly develop cross-device applications. This means that efforts should be made in order to ensure that the framework has a minimum of built-in and automatic features, so the developers do not need to implement these themselves. Instead, the developers can simply re-use these features. For

example, instead of requiring the developer to develop a UI in order to allow users to distribute items, develop a built-in UI for users to distribute specific items to specific devices into the cross-device application framework. This requirement will be tested with an experiment available in the evaluation Section 6. Since very few public cross-device applications can be found, the development of cross-device applications must be easy enough for developers in order to let these adopt the relatively new technology of cross-device interaction, hence the requirement of rapid development. While some current frameworks already contain such features [32], these should not harm the functionality and control a developer has over the framework. Secondly, the developer-friendliness should not harm the user-friendliness of the framework as well.

R4. Device agnosticism and universality. The framework should be device agnostic (be able to run on multiple environments using the same code base without requiring any modifications at all) in order to avoid more work for the developer of the cross-device application. The framework should also allow for universality. This means that currently existing websites should be able to be used within the framework for distribution of parts between devices, without requiring any changes to these websites. Another part of universality is the ability to provide a simple template application which works on a multitude of platforms and contains a web container, for the chosen website to be used within the cross-device framework. The developer will then only need to fill in the website address in order to create a minimal yet complete cross-device application which works on a multitude of devices.

R5. Protocol-independence. The DUI framework should be protocol-independent. It should be relatively easy to change the communication protocol that devices use within the framework to communicate to each other from one to another. For instance, this would allow the programmer to rapidly change the communication protocol from WebSockets or WebRTC to Bluetooth by changing a setting inside the configuration file of the framework. To our knowledge, existing frameworks lack such a feature which would enable the framework to be used in multiple manners for multiple different purposes. It would enable the framework to be truly flexible in terms of protocol and platform compatibility. It should also be relatively easy to implement a new protocol to be supported by the framework.

R6. Device Capability Detection. The framework should be able to automatically detect devices' capabilities. These capabilities can be used within the cross-device application framework for various purposes. For example, a client could request to distribute a video to a device with the *large-screen* and

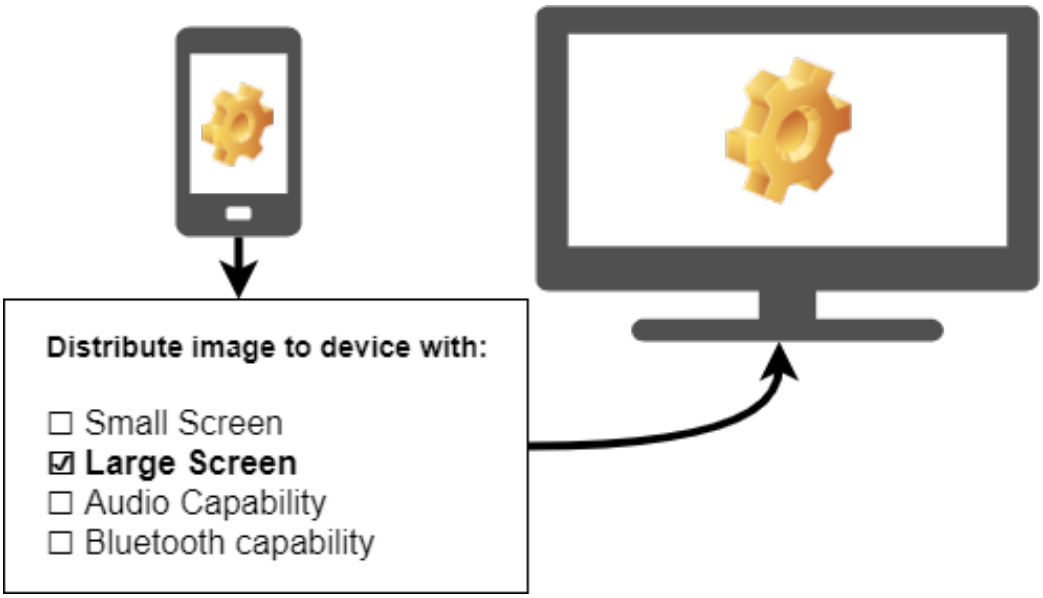

Figure 3.1: A mock-up showing a potential use case for the device capabilities information obtained from requirement *R6. Device Capabilities*.

audio capabilities, while the video controls element could be automatically opened up on the device with the *small-screen* and *touch-screen* capabilities (such as a mobile phone). In Figure 3.1, an example can be found of a user distributing an image from a mobile device to all devices with the *large-screen* capability. Some cross-device frameworks [5] were built on the idea of device capabilities and the filtering of distributed content using these capabilities. We believe this feature cannot be missed as it adds a lot of cross-device interaction opportunities for the developers using the framework. Developers could easily target different kinds of devices with different abilities when showing a screen, or distributing items, whether it is manually distributed by a user, or automatically distributed by the cross-device application.

R7. Extensible. The framework should be easily extensible. If the developers want to change parts of the core of the cross-device application framework, this should be possible in a flexible manner. Doing this without extensions would cause developers to lose their changes upon upgrading the core of the framework to a newer version (in case there is a new release of the framework available). By introducing the extensions, developers can copy the extensions to the newer version of the framework, without losing them and without having to manually copy all transformations made to the core of the framework towards the new core of the newer version of the framework, which would cause a considerable amount of time to be lost. Extensions allow the devel-

oper to maintain the changes made to the core of the framework, but in a portable format that can be re-used by a community of developers. With a potentially big community and a lot of extensions by developers, this could also increase the rapid development aspect of the framework. The framework could then act as a skeleton for the development of cross-device applications, which can then be configured and set-up according to the application's needs by using these extensions.

R8. Minimal set-up and run requirements. The set-up and run requirements of the framework should be as simple as possible in order to make it simple for both the user and developer to use the framework. As a best case scenario, the framework does not require any third party tools or servers in order for it to be run by both the developer and the user. The set-up should be as straightforward as possible. The user should be able to easily connect various devices in a session, while the developer will have a strict minimum of actions and set-up required in order to publicly release a working copy of the cross-device application. This requirement also contributes to the *Rapid Development (R3)* requirement.

R9. User-friendliness. The framework needs to comply to a minimum of user-friendliness. User-friendliness will need to be evaluated (for example, expert or user evaluations) and improved from the feedback gathered. The user-friendliness should not be harmed due to the *Rapid Development (R3)* requirement. Another important point is that the speed of an app should not be influenced in a manner that considerably negatively impacts the user experience. Since the framework will be based of a hybrid-framework, this will need to be considered during the user-friendliness evaluations as well.

R10. Platform continuity. There should be as much platform continuity as possible. If an Android device opens the app, the app should look and act like an Android app. If the app is opened by an iOS device, it should look and act like an iOS app. An example of platform continuity ccan be seen in Figure 3.2. Since we are not dealing with native apps, we cannot guarantee a native user experience for the user. However, a native user experience does form a positive contribution to the user-friendliness of an app. According to Xanthopoulos et al., native apps provide the richest user experience [31]. Compared to native apps, hybrid frameworks rarely have a similar the look and feel. For this reason, this requirement is added in order to find a way to overcome the disadvantage introduced by using a hybrid framework. This is also part of the user-friendliness (requirement R9) of the framework.

R11. Sufficient developer tools. As stated by multiple cross-device applica-

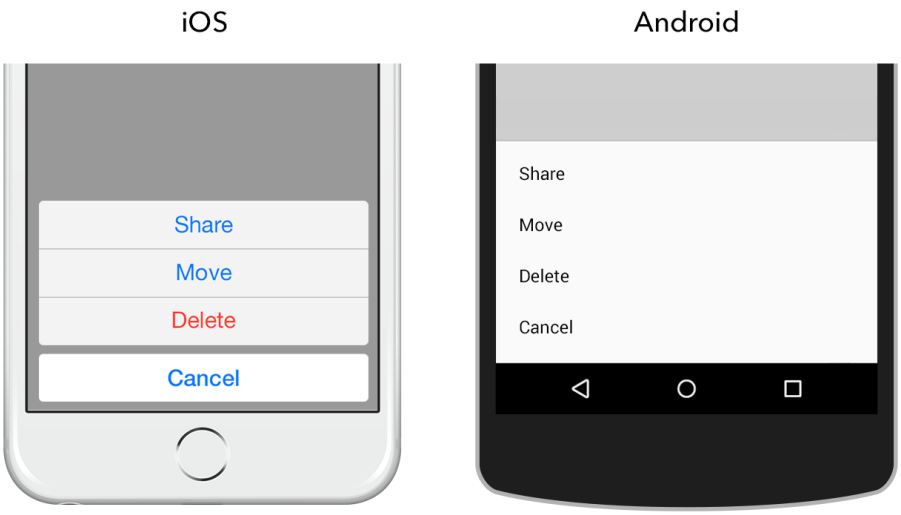

Figure 3.2: An example of platform continuity, *requirement R10*, in the Ionic framework, as demonstrated in an Ionic blog article [7].

tion framework authors [20, 14, 11], the cross-device interaction field does not contain enough tools for cross-device application developers. In order to solve this problem, researchers have developed and implemented new tools for cross-device application developers such as XDTools, XDSession and XD-Analytics [20, 14, 11]. The goal of our new framework is not to contribute to this problem by creating yet another framework which will require yet new developer tools. Instead, we want to eliminate the need for new external cross-device application tools. The framework should be compatible with existing developer tools in a way that a developer has a sufficient amount of tools for developing cross-device applications with ease.

R12. Sufficient cross-device functionality. After all, this research is about cross-device frameworks and thus the framework will need to contain more than basic functionality concerning cross-device interactions. For instance, new functionalities and concepts should be researched, discussed, developed and evaluated as part of the functionality of this researched cross-device application framework. The framework should at least be able to let the user select a part of the UI and let it be distributed to a specific device chosen by the user (whether the device is chosen by name or capability). Next to letting end-users distribute their chosen elements towards other devices, developers should also be able to link specific actions to the automatic distribution of content. Extra features such as automatically combined displays could also be interesting to add to the framework, however, this could be done later on

using plug-ins for the framework.

R13. Cross-platform. Since it is a cross-device application framework we are developing, the framework should enable us to easily develop cross-platform apps. The apps should be able to run on mobile environments, desktop environments or even TVs.

R14. Availability of the Framework. There are very few cross-device applications in the wild. In order to contribute and encourage developers to develop cross-device applications, the framework should be made available publicly and with a minimum of documentation.

3.3 Hybrid Framework Selection

There are many different hybrid frameworks available nowadays. In order to find one that suits ours needs and requirements (see Chapter 3.2), we will first have a look at a few well-known hybrid frameworks.

3.3.1 Xamarin

Xamarin[1] is a famous hybrid framework which allows developers to develop apps in C#, while the result is a set of apps that work on Android, iOS, Windows and Mac. Xamarin's generated apps are actually native apps for the respective platforms. Since these are native apps, these have native API access as well. Apps made using Xamarin can then be submitted to the respective application stores of the different platforms and apps still need to be installed on end-user devices before being able to be run. Debugging apps does require the use of a specific simulator, which needs to be installed for every platform. As an extra, Xamarin also has a component store called NuGet, which contains different extensions for Xamarin apps. Xamarin is open-source and free, however, when developers want access to a more complete IDE, a fee must be paid for access to *Visual Studio Professional*.

[1] `https://xamarin.com`

3.3.2 React Native

Unlike many available hybrid app frameworks, using React Native[2] does not involve the development of a mobile web app or a hybrid app. Instead, the result is a truly native app, developed using two popular web programming languages, Javascript and React[3]. This can be positive, since this means that a genuine native look and feel is provided to the user, when using React Native apps. It also means we do not need to think about the speed impact of using a hybrid framework, since the use of this framework actually results in native apps. Similar to Xamarin, React Native requires a simulator or actual device in order to test and debug the developed apps. While React Native is a free hybrid framework, no IDE is provided and the developers will be able to set up their own favourite development environment to be used with the framework.

3.3.3 PhoneGap

Adobe PhoneGap[4] is another popular hybrid app development framework. The framework allows developers to implement mobile applications using HTML, CSS and Javascript. A recent evaluation of the framework [1] explains that while it is easy to use and develop hybrid apps using PhoneGap, the resulting applications' usability and design are mediocre. Requirement *R9* of the HDUI framework states that the framework's resulting apps should be user-friendly. Due to this requirement, PhoneGap will not be used for the HDUI framework and no further research into this framework needs to be conducted.

3.3.4 Mobile Angular UI

Another hybrid framework, Mobile Angular UI[5] is known for the fact it uses AngularJS[6] as a base for the framework. This framework can work on both mobile devices and desktops, as required. However, the design and usability on the desktop platform is mediocre as well due to the resulting web page still containing UI elements which are mostly optimised for mobile

[2] `https://reactnative.com`
[3] `https://reactjs.org`
[4] `https://phonegap.com`
[5] `https://moileangularui.com`
[6] `https://angularjs.org`

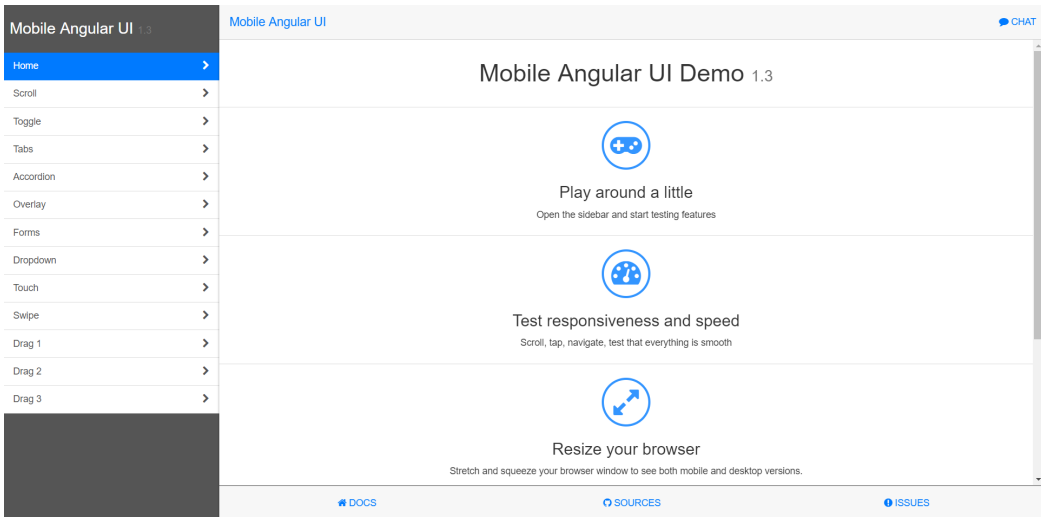

Figure 3.3: A Mobile Angular UI example app running on a desktop computer.

devices such as tablets and mobile phones. This can be seen from Figure 3.3. A hybrid framework with good user-friendliness on all platforms would be more advantageous for the HDUI framework. This framework does however provide a possibility of debugging its apps inside a web browser, since the framework is web-based, unlike the previously discussed Xamarin and React Native.

3.3.5 Ionic Framework

The Ionic framework[7] is based on an architecture of Cordova[8] and AngularJS[9]. Cordova enables Ionic to develop apps for multiple platforms such as Android and iOS, while AngularJS is used for the developing and programming the framework. Huynh et al. [15] describe the Ionic framework as the most popular framework of hybrid frameworks and state that it is usually a programmer's first choice for a hybrid framework. The authors even describe the rise of the Ionic framework as being the end of the native apps domination and expect many more hybrid apps to be developed using this framework. The authors say that Ionic has a big potential and a lot of features while still being highly mobile-optimised. The Ionic framework also

[7] `https://ionicframwork.com`
[8] `https://cordova.apache.org/`
[9] `https://angularjs.org`

contains a feature that none of the other discussed frameworks have, which is platform continuity. The Ionic framework will automatically change the UI appropriate to the device's OS look and feel. An example can be shown in Figure 3.2 and platform continuity is discussed in detail in *Requirement R10*, Section 3.2.

3.3.6 Conclusion

We have chosen to use the Ionic framework[10] to implement and develop the HDUI framework.

PhoneGap was not considered because of its lack of usability and design, which would fail requirement *R9*.

Ionic provides platform continuity and thus automatically fulfills Requirement *R10*. While some hybrid frameworks like Xamarin and React Native make use of native apps, and thus have true platform continuity, Ionic still maintains the flexibility of web-based apps. While native apps required simulators or real devices, web-based hybrid apps by Ionic could be run inside any web browser, which works in the advantage of Requirement *R3 rapid development*. Using native apps UIDL across platforms may still be tougher to distribute across devices compared to using web UIDL which is compatible across all devices the app was generated for. For these reasons, React Native was left out of the equation. Xamarin was also eliminated as a potential candidate because of similar reasons to React Native, since it uses native apps technology underlyingly, and due to the fact that the advanced IDE is not free. Mobile Angular UI's UI elements are mostly optimised for mobile devices such as tablets and mobile phones. However, the framework should be user-friendly as well across all platforms, including desktop environments. For that reason, Mobile Angular UI was not used for the development of the HDUI framework.

This left us with one possible framework: the Ionic framework. The Ionic framework fulfils the platform continuity, as discussed in Requirement *R10* in Section 3.2. Debugging and testing is easy, considering that break points can be set using the *debugger* keyword, and apps can be tested from within any web browser tab. Since Ionic has platform continuity, Ionic even lets developers debug their apps for different platforms in the exact same web browser. Developers simply need to add a parameter to the URL they

[10] `https://ionicframework.com`

are visiting, for example, for Android apps it would look like the following */?ionicplatform=android*. While Ionic apps can be tested inside a web browser, they can be tested just as well inside an emulator or actual device. Apps can be left running, while Ionic automatically builds the app and reloads it once changes are made to the code. Furthermore, an app can be installed as a native app and left running on a mobile device, while Ionic will still automatically reload it once code is changed by the developer. This flexibility is very interesting for the rapid aspect of the HDUI framework. Another interesting feature of Ionic is the fact that, since the apps are developed using AngularJS and Typescript, developers can use their favourite IDEs in combination with the framework. This automatically removes the need for extra tools for developers and fulfils Requirement *R11*, as developers already have sufficient tools for debugging, testing and development of Ionic hybrid apps. While the UI elements did still seem a bit mobile-oriented like in Mobile Angular UI, it is still acceptable for a desktop experience. There are multiple starter templates available with Ionic. The sidebar template does seem more mobile oriented, however this template can easily be customised by the developers, to fulfil their needs. The sidebar starter Ionic template can be seen in Figure 3.4. Since the Ionic hybrid framework is web-based, it is ideal for the distribution of UIDL between devices, since all devices will share the same UIDL technology, which is the web's UIDL.

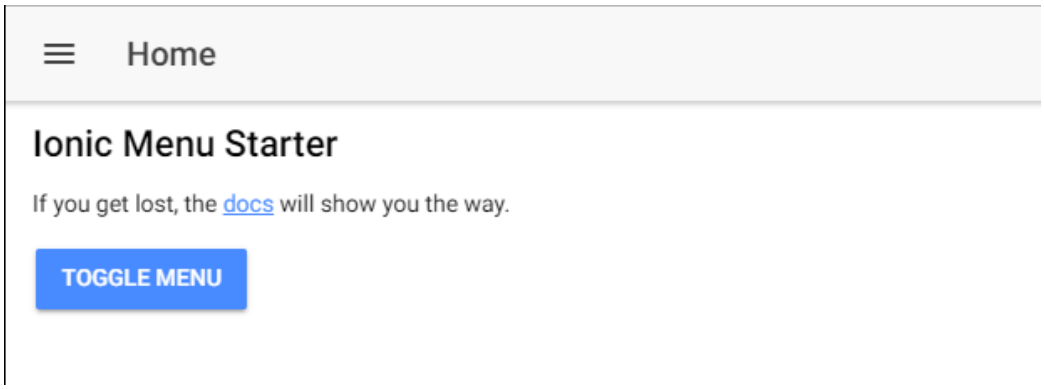

Figure 3.4: A default Ionic application, run on a desktop environment.

3.4 Framework Concepts

3.4.1 Distribution Methods

Before developing the framework, it is important to understand the different possibilities. Tesoriero et al. [26] have the following question, however, "Are we really sharing the user interface?". This leads us to think about what we are actually sharing between the devices in a cross-device application. What do we need to share and what are the advantages and disadvantages of that specific method? In order to clear this up, the following three structured distribution methods are proposed. Having distribution methods also allows us to clearly define what can or cannot be automated within the framework, in order to avoid having the developer implement features which could be automated by the framework, effectively enhancing the rapidness of the framework. Some items which require more business logic and purely depend on the application cannot always be automated.

- General Information Distribution (Automatic)
- UI Components Distribution (Automatic)
- UI State Distribution (Semi-Automatic)

General Information Distribution Method

The *General Information Distribution* method allows pure or primary information to be shared. The information also needs to be able to be recognised as a specific type of data, as both the data type and the specific data itself are key for the distribution on this level, implementation-wise. Examples of data types for this distribution method are: text, file and hyperlink. This distribution method is very general. It can be an advantage to have a general level, since it would allow the use of very general structures which are widely compatible, giving the users more freedom in what they want to distribute. The following is a concept to illustrate the use of the general information distribution method. A *distributing service* could be running in the background of an app, providing general distribution functions. When *we do not know what the user will want to distribute*, we often cannot guess the underlying data or UI representation. The user could for example select any item or data shown on the screen, right click or long press the item and choose to distribute it. The distributing service which in this case is already in an existing cross-device session, could then send the information to other devices, in terms of one of the general data types supported by the service. This

could be a text, a file or a hyperlink. The receiving device can then display the received data in a way that fits the general data type. For instance, when a hyperlink is received, the link could be opened if allowed by the receiving user. When text is received, this could be simply shown to the user of the device and so on.

The general information distribution method solves the problem of distributing unknown items of any type or structure, without direct access to the item's underlying representation, onto multiple devices. By implementing these general categories, a wide range of items can be automatically distributed. Developers could even provide plug-ins or more standard data types, allowing for an extensible manner to distribute general information across devices.

This type of distribution can be fully automated within the framework.

UI Components Distribution Method

The *UI Components Distribution* method is still a very general distribution method, since any selection of UI components could be distributed. However, the way these are distributed depends on the UIDL of the UI elements, making it more difficult to secure and distribute it in a cross-device manner. In this distribution method, the entire underlying UI components description language is distributed. The devices then need to render the component represented by the UIDL on the user interface displayed at runtime. This level provides a lot of flexibility for the user, since every single UI component can be individually distributed with a fine granularity, as categorised by the Sanctorum et al. [23] DUI classification. However, it can become expensive if the UIDL used in the application is platform-specific. This would require a parser to translate the UI syntax for every different platform and such a parser would need to be developed for different UIDL. This is essentially what Melchior et al. [17] have attempted in their model-based approach for cross-device applications. The authors introduced the concept of CUIs, Concrete User Interfaces, which could translate the platform-specific UIDL to another platform-specific UIDL, effectively allowing the UIDL to be *translated* for use with other platforms, for compatibility. Another example of this, is the concept of CUIs implemented by UsiXML [16]. Having a platform-independent UIDL could eliminate the need for such a translator or CUI models. Since Ionic is mostly web-based, the UIDL on one platform will work just as well on another platform. Panelrama [32] also benefits from a platform-independent UIDL, since this is an HTML5 DUI framework, and HTML can run on any browser and most devices with an available web browser. Another example

of a framework using the UI components distribution method, is the Proxywork framework [26]. The Proxywork framework distributes HTML and as a result requires perfectly structured HTML to be written.

This type of distribution can be fully automated within the framework.

UI State Distribution Method

The *UI State Distribution* method works in a more structured manner using specific messages send forth and back between the different devices. Let's assume the cross-device framework consists of a server and a client to illustrate this distribution method. The server could be running standalone or on one of the cross-device application clients. Both the clients and the server must be aware of the specific structure of the UI states which are communicated. This means that the developer must implement the specific actions linked to the different messages sent across the devices. Specific actions must be taken when specific UI states (messages) are communicated and the server and clients must be aware of these actions (as they should be programmed by the developer). This allows for fine-grained control of the distributed items, but requires the developer to expect and develop the functionality for the specific actions ahead of time. The framework by Frosini et al. is distributing the user interface using this type of distribution method [9]. A more concrete example made by Chi et al. that shows how this distribution method could be useful, is an image viewer application implemented using the Weave framework [5]. The application contains two types of interfaces: a controller and a viewer. One client could have the possibility to change an image, while the other client could display the image. Instead of sending the entire file (information level) or the entire UI syntax (UI syntax level), the application can send a message such as *imageNext*, which is understood by all clients, since it is implemented by the developer in the application. The user is limited to what the developer has provided within this distribution method, which can be both an advantage or a disadvantage, depending on the situation. Considering that the developer has to write client-code for the UI state messages means that the actual distribution will happen in a more controlled way and could be made more user-friendly by the developers, since the context of the action can be taken into account. This method allows developers flexibility in order to let them develop exactly what they have in mind.

This type of distribution cannot be fully automatic, but is semi-automatic in this case. This is due to the fact that developers are still required in order to implement their chosen business logic and expected application behaviours.

While the framework can automate the synchronisation of variables and the triggering of remote functions, the developer still needs to define how these variables are used and what happens inside these remote functions.

While the distribution methods may seem like individual methods, these can actually be combined in order to form a new type of distribution. For example, we could add a new data type, *HTML* for the *General Information Distribution* method, resulting in *UI Components Distribution* inside a *General Information Distribution* system. This is easily possible since the General Information Distribution Method allows for an easy way to add more data types.

Next to explaining some of the different distribution possibilities and their advantages or disadvantages, the different methods now have names which can be useful for referring to these methods and technologies in the future.

Distribution Methods in the HDUI Framework

Since the terminology and ideas behind the distribution methods have been explained, we have chosen to adopt the following types of distribution methods for the HDUI framework. First of all, in case the developers do not know what the users want to distribute, a modified version of the *General Information Distribution* method will need to be provided. Not only will it accept any data such as hyperlinks, images or text and know how to handle these in a specialised manner, but it will also be able to handle and distribute mixed content such as HTML. Like previously referred, this would be a combination of the General Distribution method and the UI Components Distribution method, which in this case is the mixed content in the form of HTML.

While the General Information Distribution method is good for explicit distribution of UI elements by the user, it does not let the developer have enough control on the framework for fine-grained cross-device interactions. Instead, the *UI State Distribution* method will be utilised for letting developers implement specific actions across devices when a user performs a particular action. Combining this with the previously discussed and chosen general information distribution method allows both the user and the developer to take control of the app.

The UI State Distribution method's implementation can still be a bit vague. In order to solve that, the following high-level concept was developed and is proposed. This concept works on the basis that a *UI State Database or Cache* is kept within the devices. Developers can add triggers to this database, with

the respective function callbacks. Whenever this trigger is triggered, the callback will be executed. Developers need to implement their triggers and callbacks in the code of the application. Triggers can be remotely triggered by any device in the cross-device session. For example, when clicking on a button, the *button-triggered* trigger is executed on all devices in that cross-device session. It would be more interesting if not only triggers could be run, but values could be synchronised as well. Therefore we added another list into the UI State DB, called the synchronised variables list. For instance, a synchronised variable with the name *brightness* could be created. The value of this synchronised variable would automatically be distributed across devices, triggering their respective callback functions. This means that changing the value on one device would change the value on other devices. Instead of calling this second feature triggers, which could be confusing for new developers, we decided to call it a synchronised variables database, which is easier to understand for new developers in the cross-device interaction space. Thus we end up with a UI State DB containing two different lists. Triggers, which do not contain any value, and synchronised variables. A mockup of the UI State DB is shown in Figure 3.5.

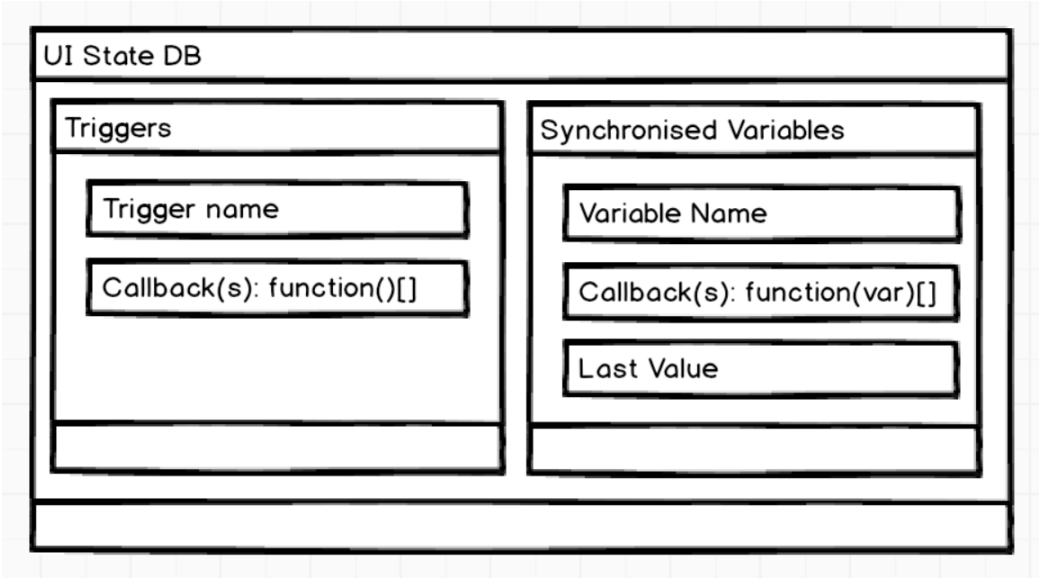

Figure 3.5: A mockup of the UI State DB, made using Balsamiq

With these distribution methods, the HDUI framework is ready for both implicit and explicit distribution. For implicit distribution, the UI State Distribution method can be used, while the developer can implement the implicit actions. As for the explicit distribution, users can select UI elements

and share them across devices by pressing the distribute button, once the user is in an active cross-device session.

In the context of fully distributed user interfaces, the HDUI framework would be a semi-automatic DUI framework. With the term *fully distributed user interfaces*, we mean an application where every single action of a user is reflected on all connected devices. While users can distribute items themselves, an application cannot be automatically synchronised between screens using the HDUI framework. Instead, developers can make use of remote triggers and synchronised variables, which are further discussed in Section 5.2. Since remote triggers and synchronised variables are automated, the developer could use these in order to implement a synchronised application. This means that for synchronised applications, the framework is semi-automatic. While it does require a bit more work from the developer, when developing fully synchronised applications, it is useful in the way that the developers have full control over the behaviour of the applications and every detail can be modified to the wishes of the developers, instead of forcing the application to be entirely synchronised. For example, in the case of a cross-device application for teachers and students, the teacher could choose not to synchronise answers to students' devices until a specific time. This is possible to implement using these concepts of synchronised variables and remote triggers, however fully DUI application frameworks would generally not be able to implement such a detailed behaviour, as this requires more control over the framework's features.

3.4.2 Framework UI Prototype

During the process of development, some mockups were made on paper. We have reproduced these mockups using a mockup tool called Balsamiq[11]. The first mockup at the left side of Figure 3.6 shows the process of connecting a device to a cross-device session. When clicking on the connection icon, the framework then checks whether the name of this device is already known from previous session and uses this in case it is available. Otherwise, the framework will show a dialog requesting the user to input a name for this device. Leaving the field blank will allow the device to be randomly labeled by the framework. While it would have been ideal to let the framework find for example, the hostname of the device and use it as device name, our findings conclude that this is not possible yet with the use of Ionic, without

[11] `https://balsamiq.com/`

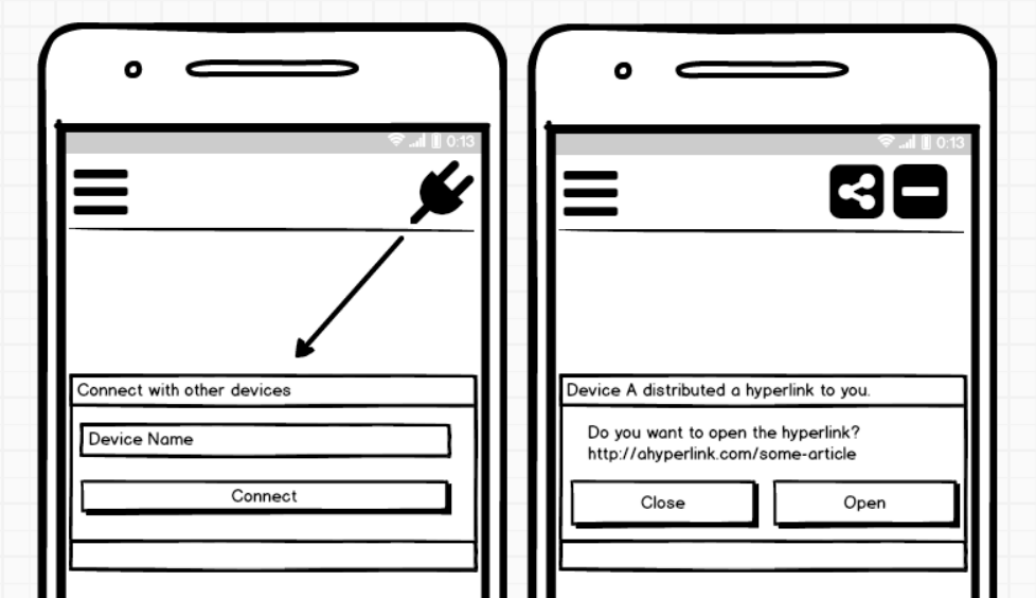

Figure 3.6: A mockup of connecting to a session and distributing a hyperlink to a device, made with Balsamiq.

requiring extra plugins which would cause the app to stop working on desktop environments.

The right part of Figure 3.6 shows other concepts within the HDUI framework. When a user decides to distribute a hyperlink to specific devices, the receiving devices will then receive a confirmation alert, for security reasons, letting them choose whether to open the hyperlink or not. Another thing to notice about this mockup is that the connection icon has disappeared, since this particular device is already in an existing cross-device session. Instead, two different icons appear, a distribute icon and a disconnect icon. The distribute icon leads to multiple choices for selecting devices to distribute to, as seen in a later prototype shown in Figure 3.7. The disconnect icon allows the device to disconnect itself from the session and alerts devices in that cross-device session that this specific device has disconnected from the session.

Distribution can happen in two different manners as described by [2], *explicit* or *implicit*. These concepts are further discussed in Section 3.4.1. Explicit distribution is when a user explicitly selects a part of the application and requests to distribute it to specific devices with criteria selected by the user. The selected UI elements are then shown in a new page or dialog on the selected devices. These devices can also see which device distributed these

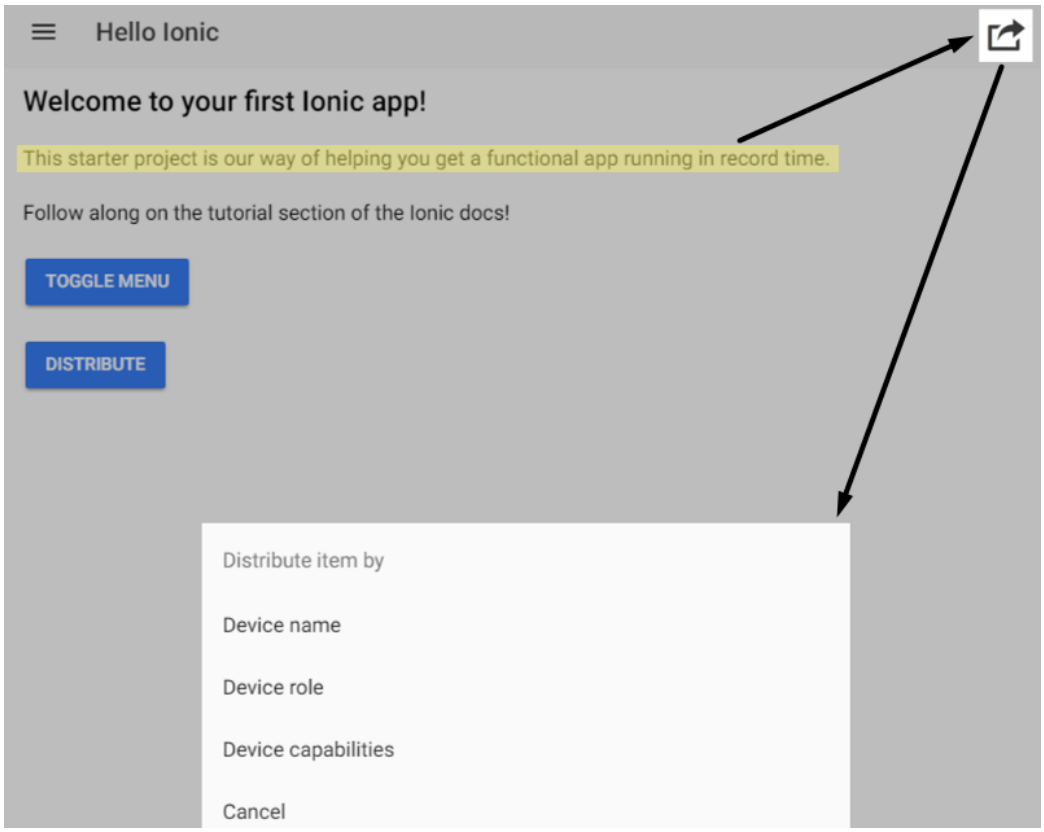

Figure 3.7: A UI prototype of the explicit distribution feature, shown on top of a starter Ionic template app.

items, by device name entered during the creation of the cross-device session. Implicit distribution happens when the developer chooses a trigger for a certain action to be distributed across devices. In order to let users of cross-device applications explicitly distribute any part of an app, we have chosen to add an explicit distribution button at the top right of the app, as shown in the prototype in Figure 3.7. The user first selects a part of the UI to distribute across other devices, which in this case is a piece of text. By clicking the share button on the top right corner, a menu is opened allowing the user to select specific criteria for distributing the item to (device name, role or capabilities). The device role could either be *display* or *controller* in the case of a video application.

Implicit distribution however, needs to be configured and used by the developers when specific actions occur and cannot be seen explicitly from the UI. The prototype in Figure 3.7 is a bit simplistic in the way that, devices need a way to connect to each other as well, before being able to distribute items.

In order to solve this, an extra icon could be added on the top right of the app, while the sharing icon would be hidden until a cross-device session is created with other devices.

3.5 Framework Architecture

The framework is based on the Ionic framework, which means that some patterns used in the Ionic framework have to be re-used in the HDUI framework. For instance, Ionic makes use of AngularJS[12], which is based on the *MVC/MVVM/MVW* architectural patterns. The architecture of the HDUI framework is mostly based on the concept of a page being a controller of the application which requires access to a number of other internal components. These components then provide the required functionality to the controller. This means that all components and features are made in separate classes and parts of the framework and are then all integrated into the HDUIController for easy usage by the developer. This controller is then inherited by every page in the HDUI application and the developer is able to access and use the most important functionalities from this class. During the initialisation of an HDUI page, the inherited HDUIController ensures the cross-device functionality is automatically available to the end-users.

The HDUI Controller class presents a central entrance point for cross-device functionality that the developer can use. It communicates with other components for connecting to sessions, distributing elements, synchronising variables or triggering remote functions on other devices in the cross-device session. It also takes care of the basic menus presented to the user.

There are a number of different components that the HDUI Controller communicates with. These components are further explained and listed below. These components can also be seen in the HDUI architecture diagram in Figure 3.8.

Closely related components are closely together in the architecture diagram as well. An arrow with an empty arrowhead indicates inheritance, while arrows without arrowheads indicate an association, in this case the association is the usage of a component. The direction of the usage of components is all towards the framework's core, the HDUIController.

- HDUIController

[12] `https://angularjs.org`

- General Communication Component
- Communication Implementation Component
- Communication Factory Component
- Configuration Component
- Capabilities Component
- General Distribution Component
- UI State Distribution Component
- Session-Pool Management Component (optionally third-party)
- Cross-Device Sessions Management Component
- Plug-in Management Component (see remark below)
- HDUI Pages

HDUIController

While the HDUIController is considered as the root component which makes use of the components discussed below, we will also discuss it here. The HDUIController integrates all the different components to work with one another. It is the brain of the HDUI framework. It provides UI feedback, sets up the communications with the use of modules and detects the elements selected for distribution to other devices in the cross-device session. It also contains the instance of the SyncDB discussed in the UI State Distribution component. The HDUIController is also responsible for anything that has to be done with a generic HDUI page. That means that the logic behind the buttons on the HDUI menu, as further discussed in Section 4.3, needs to be implemented here. For example, when connecting to a session a loading dialog is shown and the connect button is replaced with a disconnect button. This is the responsibility of the HDUIController, implementing the base logic of an HDUI page.

General Communication Component

- GeneralProtocol abstract class
- IProtocol
- IProtocolAddress
- ProtocolMessage

The general communication component defines the minimum behaviour of a protocol inside of an abstract GeneralProtocol class, in order to set a contract

which the developer can follow as well as provide as much functionality as possible so the developer would have less work for implementing a new protocol into the framework. Interfaces and a standard message encapsulation is also provided within this component.

Communication Implementation Component

- Protocol<*name*> (extends GeneralProtocol)
- Protocol<*name*>Address (implements IProtocolAddress)

This component contains the actual protocol implementations. The developers need to create a new folder with the shortened name of the protocol and implement the two required minimum protocol implementation classes as illustrated above this paragraph. One of them needs to extend the GeneralProtocol class and implement the abstract methods, while the other needs to implement the IProtocolAddress class in order to instruct what a device address looks like in that specific protocol.

Communication Factory Component

- CommunicationFactory

The Communication Factory is in charge of initialising the protocol-related classes of the correct protocol instance, as set in the HDUI configuration file. This component is implemented using the factory method design pattern.

Configuration Component

- HDUIConfig

This component consists of a single class which holds the main configuration of the framework. Developers should start by configuring the different options of the framework from this file. The configuration is further discussed in Section 4.9.

Capabilities Component

- HDUICapabilitiesIdentifier
- HDUICapability
- HDUINetIdentifier

The HDUICapabilitiesIdentifier identifies the capabilities of the device and allows them to be used for instance, for distributing elements only to devices with particular capabilities such as *large_screen* or *cellular*. These capabilities can also be easily extended by a developer. The HDUINetIdentifier

identifies the performance of the device on the network, including network speeds and RAM limits of the device. This is useful for determining which client device inside a new cross-device session should take care of the cross-device session management component, since this is currently only the case for one device per session.

General Distribution Component

- HDUIGeneralController
- GeneralDData
- GeneralDFile
- GeneralDType (enum)
- GeneralDUtils

In combination with the HDUIController, general distribution is made automatically functional for HDUI pages. It allows users to distribute any element of a page across devices in the cross-device session. The other classes in this component support the general controller for operation with separated concerns. The GeneralD prefix stands for General Distribution. A GeneralDType enum is used in order to differentiate the different kinds of general data which can be distributed, along with the display template for every different supported general data type.

UI State Distribution Component

- UIStateData
- SyncDB

The UIStateData class is simply used for structuring the incoming and outgoing UI State data. The SyncDB however, is more interesting. The SyncDB is available on every client in a cross-device session. It contains two databases. One database contains a pair of variables and the corresponding trigger function while the second only contains trigger functions for a specific identifier. This database is automatically kept in sync by the HDUI framework and the respective triggers are run every time a change is detected in one of the variables, or every time a remote function identifier was triggered. The SyncDB is the base structure for the entire UI State Distribution method. While being quite simplistic, it can be used in many different situations in order to fit the developer's needs while not over-complicating the task of developing cross-device applications.

Session-Pool Management Component

- HDUISessionConnector

While the client pools for creating new cross-device sessions are managed on the NodeJSRouter for the Socket.IO protocol, it would be very inconvenient for developers to have to re-implement this functionality over and over again for every protocol and it would render this framework to be quite the contrary of rapid. Our solution to this problem is to provide a generalised class which contains a simple session matchmaking algorithm which is quickly reused. While for the Socket.IO protocol, this class is used externally within a separate NodeJSRouter project, it can as well be used within the code used by clients of the HDUI framework. This means that it would work for both protocols where an intermediate server is required, or protocols where one would not use an intermediate server but where devices are directly connected to each other. Not creating a separate component for the session pool management would result in a more difficult process for integrating new protocols within the HDUI framework.

Cross-Device Sessions Management Component

- HDUIClient
- HDUIClientList
- HDUISimpleClient
- HDUIServer
- HDUIServerClient

One device per cross-device session is appointed to be the *server* of the session, meaning that it will keep the list of all clients inside the session and accept or reject new clients from entering the session. The choice of using one device as a server instead of multiple devices is discussed in further Section 3.6. While this makes the redundancy of the server within a cross-device session more complex, it has performance advantages that cannot be neglected. There are a few different reasons why we need the data that the server keeps within the framework. For instance, when a device distributes to another specific device, it needs to know the up-to-date names of the other devices in the cross-device session along with their protocol addresses.

Another reason why the server is very useful in the framework is that a broadcast distribution is possible within the framework. Performing a broadcast distribution works in the following manner. First of all a client will forward the broadcast request towards the server of the session - if the device is not already the server. The server of the session then receives the request and forwards it to every device connected in the session to process this distribution. Broadcast distributions are only accepted if they originate from the server of the session. This means that once a client initiates a broadcast, it is not

immediately broadcasted to that client itself, but it is only accepted once it has travelled to and from the session server. This is to ensure that the sender of the broadcast distribution only notices the distribution happen after it has effectively been accepted and performed by the session server.

In case of a pure peer-to-peer cross-device session management - i.e. without a server - the synchronisation of data is much more complex. Whenever a client joins the framework, it does not know who to contact for registering into the session and it is much harder to propagate all the information in an efficient manner across the devices. In case the protocol offers functionality for automatic session broadcast, it would be much easier, but we cannot make this assumption as this could harm the protocol-independence of the framework. The session management using a single entity does provide some issues for server redundancy during disconnects, but some solutions exist for this and this is further discussed in Section 3.6.

Plugin Management Component

- HDUIPluginManager

In order to make the system extensible, a plug-in system could be implemented. Specific parts in the code can tell to run the plug-ins that have subscribed to those specific hooks. For instance, assume we have a joinSession() method, inside this method, we can request to run all the hooks with the joinSession_before name, and afterwards run all the hooks with the joinSession_after name. Many popular applications or frameworks make use of this plug-in hooks system, such as Cordova itself. Cordova is the underlying hybrid system that the Ionic framework uses. However, during the research and development, more interesting areas of research were found such as distribution methods and protocol-independence. Due to this shift in focus, this particular component was not practically developed and is a future work of the HDUI framework. It would be useful for developers using the HDUI framework, but this would not be an enormous contribution to the cross-device research field, since plug-in and modules systems already exist in many different research fields. Such a system could be re-used and implemented into the HDUI framework as a future work.

HDUI Pages

- Page.ts class (extends HDUIController)
- page.html template
- page.scss template design

Finally, there are the actual HDUI pages which extend the HDUIController in order to inherit the cross-device functionality, as well as the necessary page templates and design using SCSS.

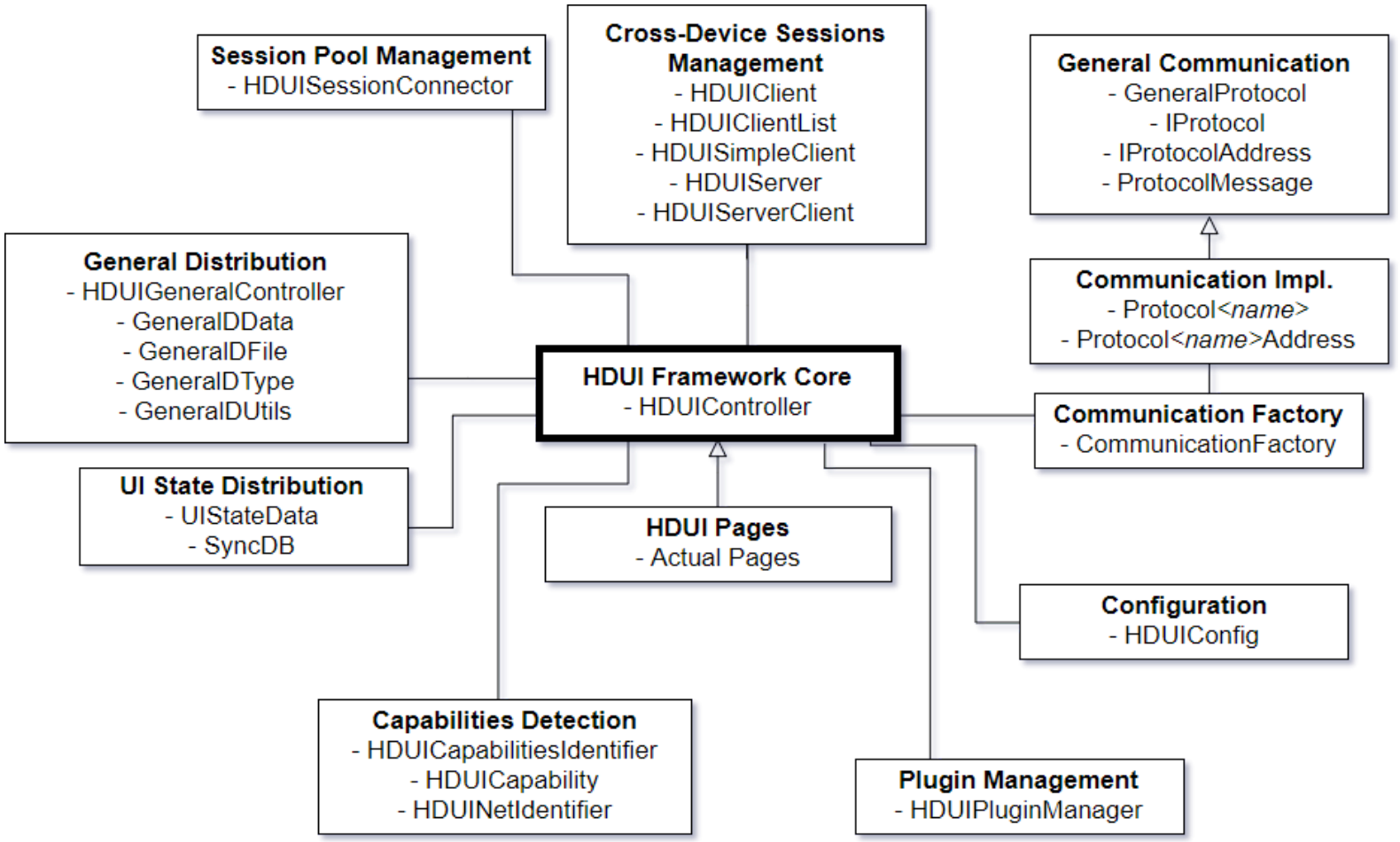

Figure 3.8: The architecture of the HDUI Framework.

3.6 Session Management Architecture

When clients do not have a session and want to join a new session, sessions are recognised that are attempting to join and matched together. However, in this case we are not discussing about managing the matchmaking of clients, but we are discussing the management of existing cross-device sessions. Clients can leave and join at any time and this needs to be managed by every cross-device session. We do not want to rely on the chosen protocols to do this, since the framework should be protocol-independent, as discussed in Requirement R5, Section 3.2. Note that in this case when a *server* is mentioned, it is actually a client of a cross-device session which is considered a server, as it is managing the cross-device session.

There are multiple ways to implement this. We haven chosen to implement this using a centralised server architecture with an addition for ensuring redundancy within the cross-device sessions. This choice is explained in the following sections. First of all, the data that should be kept within every cross-device session is discussed.

3.6.1 Managed Data

A session in the HDUI framework should keep some data which is useful for that session and its management by an entity or in a peer-to-peer manner. We defined a session to contain *at least* the following data and functions.

A server address - all UI state distribution requests are sent to this device and then propagated across the devices, if the request is valid.

A list of clients, which would contain the clients' names, capabilities and addresses for the chosen protocol as a minimum. This could easily be extended in the future.

A method for registering clients into the session. The client needs to be registered into the client list of that session. This is done using this method. Another aspect is controlled in this functionality. The names of devices in the cross-device session should be unique. This method ensures this. In case a device joins with a name which is not unique, a suffix could, for example, be added.

A method for receiving the entire client list of the session. Any clients in the cross-device session should be able to explicitly request the client list.

A method for receiving client list updates as soon as the client list is updated. Clients of the session need to keep an updated list of the clients. This is useful, for example to allow clients to distribute to other devices by the names from the client list.

This data list could be extended in the future by adding functionality such as session passwords in order to ensure no intruders can join the cross-device session.

3.6.2 Architecture

We have chosen to use an architecture with a centralised server for each cross-device session. However, at first, a peer-to-peer architecture was considered due to the advantages in terms of redundancy when devices disconnect. However, the framework is supposed to be rapid and we like keeping the implementation as simple as possible in order to solve complex problems. That way, when the framework needs to be extended, it can easily be understood by developers and this contributes to the rapid development part of the framework. A centralised server which contains all the logic for the cross-device session and which does not need to negotiate with other clients on what should happen is easier to implement and understand. For instance, when a client joins a current cross-device session, it needs an entrance point for registering into the session. With a centralised server, this entrance-point is clearly defined. As a bonus, the device with the best speed and performances could be detected and assigned the task of being the cross-device session server.

3.6.3 Redundancy

Choosing for a simpler client-server based architectures leaves us with the problem of redundancy. What if the server's connection fails? That would be problematic in a pure client-server implementation. However, the concept of *High Availability* is used for the HDUI framework. High availability means that in case the server disconnects in any way, another client in the cross-device session needs to pick up the task of being the server. An alternative to keeping the server's data in sync across the clients, which costs bandwidth, is to simply request the data again from the clients once a new server is assigned. This means that there is no need to keep sending the synchronised data across all clients of the cross-device session. Instead, once one client

notices that the server has disconnected from the session, it checks whether it is allowed to be the replacement of this server with the use of priorities, which are synchronised as part of the client-side functioning of the framework. If the client is the next device allowed to become the server, the client with the best speed and performance of the cross-device session, it takes over the task, notifies the other clients that the server has started and requests all details from all clients again. The process that happens during initial connection is being re-used behind the scenes. While this can be seen from the communication between the clients as a developer, users will not be able to notice this reconnection. This way, redundancy is achieved within the cross-device framework. However, since some technologies such as Socket.IO[13] do not notify other devices when a user abruptly disconnects, for instance, because of a connection issue or browser crash, a keep-alive system had to be implemented as a backup. The server sends a keep-alive message to all clients. If the message does not arrive on time, the clients start the procedure for changing the server of the cross-device session. Since some protocols might have something similar built-in, such as MQTT[14], an option was added in the configuration of the framework, HDUIConfig, to turn the keep-alive on or off. The configuration of the HDUI framework is further discussed in Section 4.9. The configuration of the keep-alive system is shown in the below Listing 3.1.

```
1  // Polls the server to make sure it is still there
2  // If there is no reply, a new server is assigned in the
   ↪  cross-device session
3  // Use this in case your chosen protocol does not notify about
   ↪  devices who got disconnected
4  // due to connection errors or a crash
5  public useKeepAlive = true;
6  public keepAliveIntervalMs = 2000;
7  // The maximum time the keep alive message can be delayed
   ↪  without assigning a new server
8  public maxKeepAliveDelay = 500;
9  // For the initial connection setup, the keepalive timeout will
   ↪  be multiplied by this number
10 // in order to avoid issues during connection setup with the
   ↪  keepalive.
11 // The keepalive will be reset to the original value after the
   ↪  first keepalive message is broadcasted.
```

[13] https://socket.io
[14] https://mqtt.org

```
12 public keepAliveInitialConnectionMultiply = 5;
```

Listing 3.1: The configuration of the keep-alive system implemented by default in the HDUI framework.

Due to the lower implementation complexity, taking into account the different technologies we are working with, we decided to use a client-server based architecture with an easy simple addition that ensures the redundancy of the server within the cross-device sessions.

3.6.4 Client-Server Architecture

Once a session ID and server ID is assigned to a client, the cross-device session is formed and it needs to be managed from within. In case the device notices it is the server, the *HDUIServer* class is used instead of the *HDUIClient* class. Since the HDUIServer class inherits the HDUIClient class' functionality, it is easy to replace without requiring changes in the framework's core, the HDUIController.

Once a session is assigned to all the clients, the clients must contact the server with their identity (ID, name and capabilities). In order to make sure that the server is set up, the clients await a message from the server before doing so. The server confirms the individual device registrations, allowing the individual devices to show a success confirmation to the users with their assigned names, which can be modified in case of duplicates. The *Session Set-Up Process* is shown in the following table. The server is one of the clients that was assigned as the server due to having a bigger priority, in order to give priority to devices with better connections and speeds.

Session Server	**Direction**	**Clients**
RequestDetails()	==>	Test and gather capabilities
Register device	<==	**requestRegistration (deviceId, name, capabilities)**
confirmRegistration()	==>	Success feedback to user

Table 3.1: Session Set-Up Process (Client-Server Architecture)

After this process, the devices are connected and can then distribute items to each other without needing to worry about redundancy.

4

Implementation

The HDUI framework was implemented using the Ionic framework[1], as discussed and chosen in Section 3.3.6. Some of the features and their implementation solutions are discussed in this section.

4.1 HDUI Page Template

The framework provides general distribution features without requiring the developer to program anything. Simply extending pages as shown in Listing 4.1, allows the general distribution features to be automatically enabled on that specific page, however there is a caveat.

```
1 export class MyPage extends HDUIController
```

Listing 4.1: Enabling cross-device functionality on an application page.

While letting a page simply extend a defined controller would be ideal for the rapid prototyping of the system, a few extra definitions are required in order to let the framework function as expected. While there are not many

[1] https://ionicframework.com

definitions to add, they can still be an obstacle that a developer would need to remember. In order to solve this problem, an empty page template was created and inserted into a project template, so that developers could copy and paste this in order to add new pages to the project. This empty page template is shown in the following Listing 4.2.

```
1  // Required superclass
2  export class HDUIPageTemplate extends HDUIController implements
   ↪  AfterViewInit {
3
4          // A. Required ViewChilds for HDUI
5          @ViewChild('duiBtnStart') duiButtonStart;
6          @ViewChild('duiBtnStop') duiButtonStop;
7          @ViewChild('duiBtnShare') duiButtonShare;
8          @ViewChild('duiBtnFile') duiButtonFile;
9          @ViewChild('duiMenu') duiMenuBase;
10
11         constructor(public navCtrl: NavController,
           ↪  actionSheetCtrl: ActionSheetController, toastCtrl:
           ↪  ToastController, alertCtrl: AlertController,
           ↪  loadingCtrl: LoadingController, modalCtrl:
           ↪  ModalController, diagnostic: Diagnostic) {
12                 // B. Required super call
13                 super(navCtrl, actionSheetCtrl, toastCtrl,
                   ↪  alertCtrl, loadingCtrl, modalCtrl,
                   ↪  diagnostic,
                   ↪  HDUIConfig.pageIdentifiers.template);
14         }
15
16         ngAfterViewInit(): any {
17                 // C. Required initialisation for HDUI buttons
18                 super.setupDUIButton(this.duiMenuBase,
                   ↪  this.duiButtonStart, this.duiButtonStop,
                   ↪  this.duiButtonShare, this.duiButtonFile);
19         }
20 }
```

Listing 4.2: A new HDUI page template.

The reason this extra code is required, is because it is not possible to add and manipulate the DUI menu buttons from within the extended controller class in Ionic. Code parts A and C in Listing 4.2 are required for retrieving

the DUI-related buttons on the page and initialising these. Code part B is required because we would otherwise have no access to these controllers which are automatically injected, while access to these controllers from within the HDUI framework is required. An attempt was made at automatically injecting the menu buttons into a child class, however, it is not possible due to security features in Ionic, creating the need for code parts A and C. This means that the developer also needs to use a template for the page's layout which includes the actual buttons. This can be seen in Listing 4.3. However, an advantage of not automatically injecting these menu buttons is that they can be easily customised within individual pages. While it would have been good to eliminate the need for all of these code parts, the template page is still *very minimal and rapid to use* for developers.

```
<ion-header>
        <ion-navbar>
                <button ion-button menuToggle>
                        <ion-icon name="menu"></ion-icon>
                </button>
                <ion-title>Ionic-HDUI Starter</ion-title>
                <!-- Required buttons for the HDUI framework,
                ↪  customisable -->
                <ion-buttons right duiMenu>
                        <button ion-button outline icon-only
                        ↪  color="primary" hidden
                        ↪  (click)="disconnect()" #duiBtnStop>
                            <ion-icon
                            ↪  name="remove"></ion-icon>
                        </button>
                        <button ion-button outline icon-only
                        ↪  color="primary" hidden
                        ↪  (click)="hduiPopup()" #duiBtnShare>
                            <ion-icon
                            ↪  name="share"></ion-icon>
                        </button>
                        <button ion-button outline icon-only
                        ↪  color="primary" hidden
                        ↪  (click)="distributeFilePopup()"
                        ↪  #duiBtnFile>
                            <ion-icon
                            ↪  name="attach"></ion-icon>
                        </button>
```

```
18                          <button ion-button outline icon-only
                             ↪  color="primary"
                             ↪  (click)="hduiJoinSession()"
                             ↪  #duiBtnStart>
19                                  <ion-icon
                                     ↪  name="wifi"></ion-icon>
20                          </button>
21                  </ion-buttons>
22          </ion-navbar>
23  </ion-header>
24
25
26  <ion-content padding class="getting-started">
27  <!-- Page content -->
28  </ion-content>
```

Listing 4.3: The HTML template of an HDUI page modified from an Ionic page template.

4.2 Communications and Protocols in the HDUI Framework

The chosen default communication protocol for the HDUI framework is WebSockets, for the following reasons. Specifically, the Socket.IO implementation of WebSockets is the default communication protocol in the HDUI framework. One reason is that Socket.IO has good cross-platform compatibility. It works on most devices that have a recent web browser and it allows for *bi-directional communication*. One downside of Socket.IO is that it requires a server to be present and cannot be used for true peer-to-peer communications, as a server is still required between the group of devices. One way to solve this problem would be to create a server on one of the devices, which other devices would find through broadcasting. However, the technology has is limited as it cannot create broadcasts over a network, due to security reasons and a Socket.IO server cannot be run on client devices. One solution we found was to create a Node.JS[2] server. Devices would all need to connect to this server, while the server would contain very few logic and mostly forward the correct messages to the correct devices. In order to do this, a

[2] `https://nodejs.org`

Node.JS server that we call the *NodeJS Router* was developed which keeps a table of cross-device sessions along with the device IDs, as seen in Figure 4.2. Along with some basic communication capabilities the NodeJS Router is able to forward the message to the correct device. Since its main purpose is to forward the correct message to the correct device, we called it the NodeJS Router. It also contains the code in order to match devices into one session. Its functionality is all created into one protocol-independent *HDUISession-Connector class*. In case another protocol would be implemented, it would allow the developers to re-use the code used in the NodeJS router without getting stuck by the protocol-dependencies. This is also important for the rapid prototyping aspect of the HDUI framework. Adding a server for forwarding all communications can have some safety and privacy repercussions, which is why the data sent through this server should be encrypted in the future and further security research needs to be done in the are of the HDUI framework and cross-device application frameworks in general. This concept can be seen in Figure 4.1.

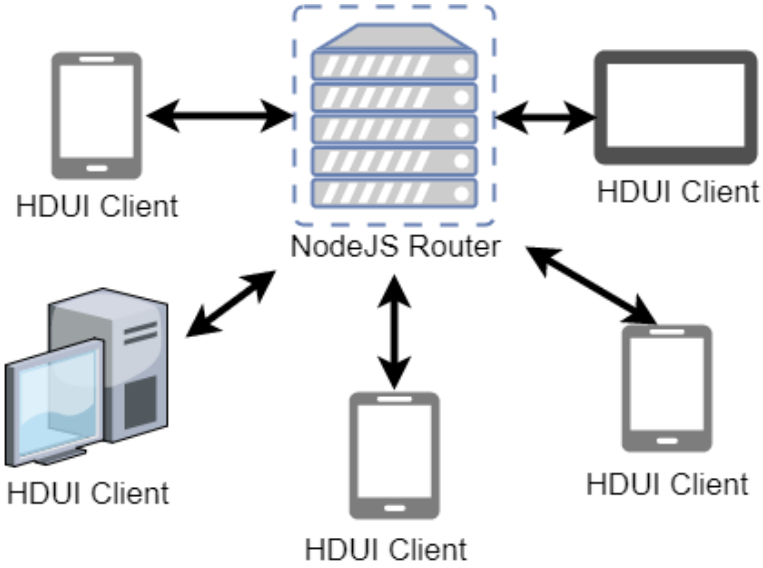

Figure 4.1: The NodeJS Router concept for use with the Socket.IO technology.

Looking back at Figure 4.2, we can see that every session also has one specific session server, which is a client device in the cross-device session. The reason for this is that we want to keep the logic implemented in the NodeJS Router as minimal as possible. The more logic implemented in this router, the harder it becomes for a developer to develop a new protocol for the

framework, as this logic will become part of the framework — if the developers do not make use of the *HDUISessionConnector* which ensures that this logic can be re-used without the need of re-implementing it. By assigning one device in a session as a server, we can keep the server-logic in the client-side of the framework. The client-side should never have to be modified for a protocol to work, which makes it wise to keep the most logic as possible in the client-side instead of the server-side. The protocol would just need to be added as a new class in the framework. We can see that a truly peer-to-peer communication protocol would be ideal for supporting the protocol-independence of the HDUI framework. Unfortunately, the devices cannot broadcast themselves over a network using WebSockets or even WebRTC, due to security constraints. Doing so might even be disadvantageous to the HDUI framework, as that would require the devices to be all in the same network, adding location constraints as illustrated by the classification of Sanctorum et al. [23]. With the current system, the devices can be on any network, anywhere, as long as there is a connection to the internet. A client list can be found in Figure 4.2, however, the NodeJS Router does not save the names of the different devices, but only temporary IDs provided by Socket.IO in order to reduce the risk of privacy-related issues.

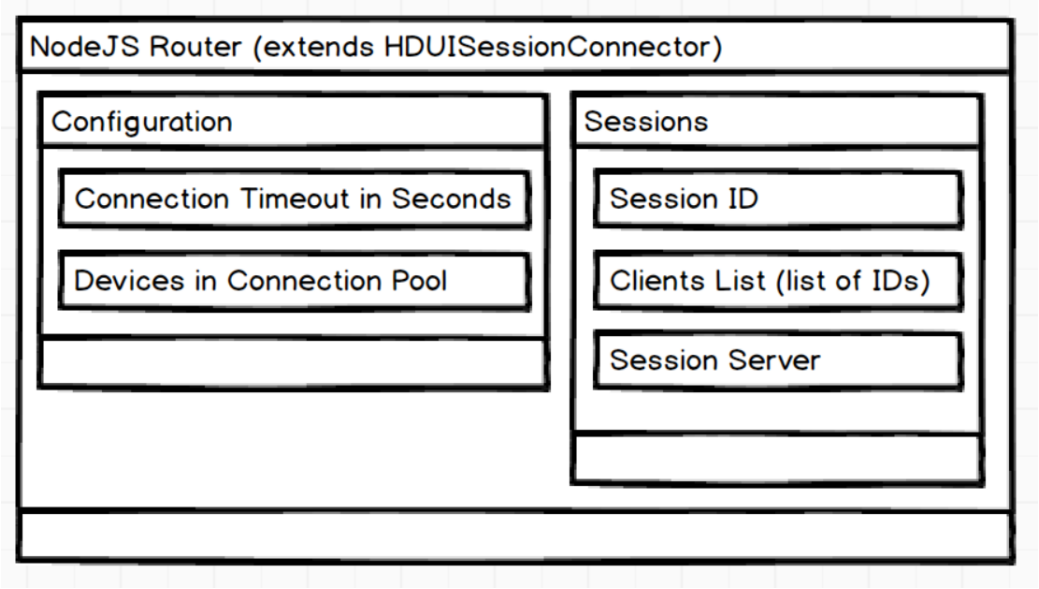

Figure 4.2: The data and configurations of an HDUISessionConnector class, extended by the NodeJS Router. Diagram made using Balsamiq

Another similar protocol, *WebRTC*[3] (Real-Time Communications) is also

[3] `https://webrtc.org`

worth researching. While Socket.IO requires a server, WebRTC allows for direct communications between clients once a connection has been established. This means that the first time connecting, it will still require a server, but future communications can pass directly to the client, effectively eliminating the server in between. However, at this time the browser compatibility might be an issue with this protocol and especially in a cross-platform cross-device context. WebRTC has some quite good compatibility with web browsers such as Chrome and Firefox, however the compatibility is not as good for Safari and Edge. This might be troublesome for devices with more primitive web browsers, which is why WebRTC was not implemented in the HDUI framework. However, since the HDUI framework is protocol-independent, a small plugin of about 100 lines of code would be able to add WebRTC communication capability to the HDUI framework. This protocol could be added to the framework once WebRTC has matured and compatibility has improved.

4.3 The HDUI Menu

Once a HDUI page is created in an HDUI application by using the template explained in Section 4.1, users can automatically start creating cross-device sessions as well as they can distribute UI elements or other data on that specific page.

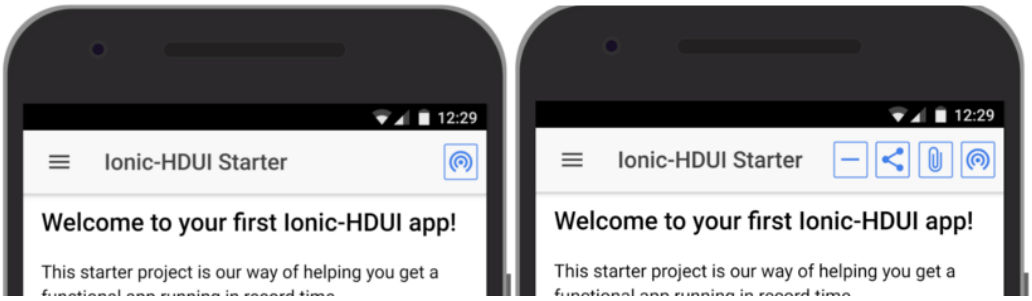

Figure 4.3: The HDUI menu for devices which are not connected to a cross-device session (left). The HDUI menu for devices in a current cross-device session (right).

An HDUI page consists of a menu for controlling the cross-device features and functionality. This menu is different for users which are currently in a cross-device session and users which are not in a cross-device session. This is shown in Figure 4.3. On the left side, we can see a device which is not

connected to a cross-device session. Only one icon appears, the connect to another HDUI session icon. On the right side however, we can see a device which is connected to a cross-device session. Four icons appear, these are discussed from left to right. The first icon is a *disconnect icon* which lets the user safely disconnect from the cross-device session and informs its peers that it has disconnected. A disconnection alert is sent to all devices in the cross-device session, as shown in Figure 4.4. The second icon is the *distribute icon*. It allows users to select a part of the UI, text or hyperlinks in order to distribute it to devices in the cross-device session. When clicked, it brings up a device selection menu, allowing the user to select which devices it is distributed to within that session. This is further explained in Section 4.6. The third icon is the *file distribution icon*, which effectively allows users to distribute files from one device to other devices in the active cross-device session. Similar to the distribution icon, it lets users specify criteria for selecting devices to distribute the file to, as further explained in Section 4.6. Finally, there is a *connection icon*, which allows new devices to join an active session.

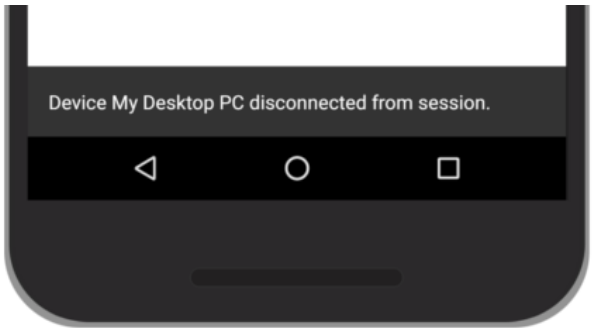

Figure 4.4: Alerting devices in a cross-device session that a device has disconnected.

4.4 Cross-Device Sessions

There are many different ways for cross-device frameworks to implement a way for connecting multiple devices into the same session. For instance, Chi et al. [5] introduce a shaking gesture that has to be done at the same time on different devices, in order to let these connect to the same session.

We decided to keep the initial manner for connecting devices as simple as possible, as according to *Requirement RX extensibility*, the framework should

be extensible and further gestures or complex interactions for connecting devices into the same session could be developed as extensions. In the case of the provided HDUI framework, sessions can be made between multiple devices by pressing the connect button at roughly the same time. That is, within a time-frame of about 10 seconds, which is a configurable setting in the HDUI framework. This is fine if only a limited amount of people are going to use the framework. However, in order to make this more scalable, an extension could be written requiring the devices to be in roughly the same location. This would mean that the intermediate server the HDUI framework uses, which is further discussed in Section 4.2, could be shared between multiple strangers that do not want to have an active session with each other. However, an alternative would be to use a different server for example, by organisation or group,

If a cross-device session is already existing between multiple devices, a new device can join them by clicking the connect button on the disconnected device and one of the connected devices. The connected device will re-open the session for new users by pressing the connection icon at roughly the same time as a disconnected device.

When devices press the connection button for joining or creating a new cross-device session, they are requested to fill in a device name which would be recognisable for the other devices and users in the cross-device session. Once filled in, the device will be connected to the cross-device session. The device name will be remembered on every device, by keeping it in the cookies of the device. If the cookie is present, the framework will not ask the device name anymore, in order to let the user have a fast user experience with the cross-device session framework. The following figure 4.5 illustrates how the framework requests the device name of a new device before connecting to a cross-device session. If the device's name already exists in the cross-device session, a number will be appended to the device's name. When the device connects, the name is shown to the user of the device using a toast message. For instance if a second device in a cross-device session is called *Device*, the device's name will be changed to *Device 2*. Alerting the cross-device session devices of the new device joining the session allows them to identify the name as well.

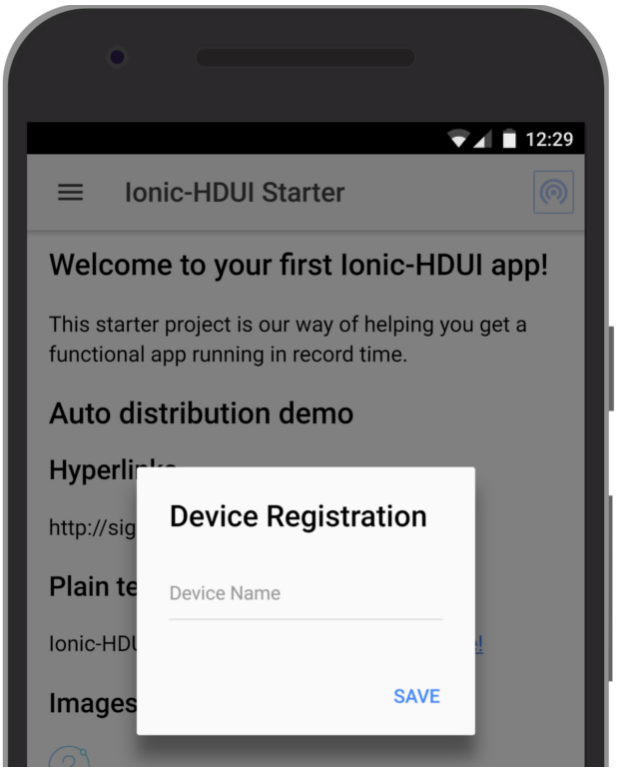

Figure 4.5: Requesting the name of a new device before connecting to a new or existing cross-device session.

4.5 General Distribution Method

The general distribution method enables users to select any UI elements and distribute them however they please across other devices in the cross-device session. Developers simply need to be using the provided templates and classes in order to ensure this happens automatically.

The previously discussed HDUI page template code in Section 4.1 automatically allows users to have an interface for interacting with the cross-device functionality of the framework sessions without requiring the developer to develop any code for cross-device functionality. It allows the user to make a selection of UI elements on the existing page and distribute it to the specifically selected devices. Devices can receive various preprogrammed data types such as text, hyperlinks, mixed data such as HTML and even files, which is a feature that was added later on.

Since any hyperlink could be distributed across devices, we added a confir-

mation dialog to the receiving devices in order to avoid abuse and security issues involved in the distribution of hyperlinks. A similar dialog was added for confirming the distribution of files across devices. This feature is shown in Figure 4.6. The feature is automatic and the developer does not need to implement anything for these features to work in the cross-device application.

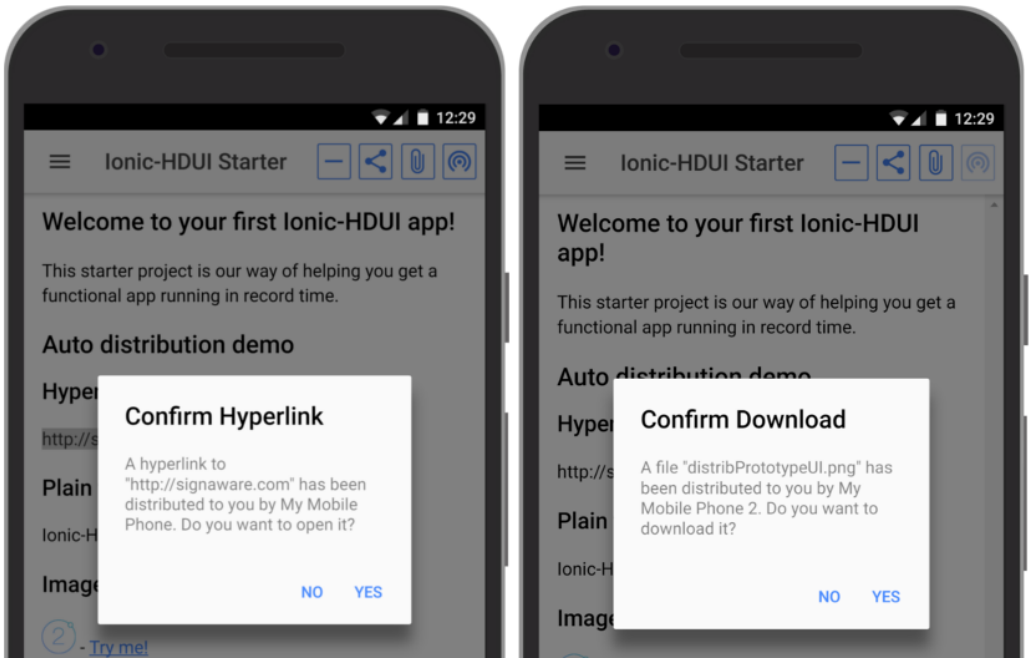

Figure 4.6: A device receiving requests to download a file or open a hyperlink from other users in the cross-device session, using the HDUI framework.

It is important to note however, that files and communications are currently sent across the *NodeJS Router* discussed in Section 4.2, when using the *Socket.IO* protocol. In the future, encryptions should be used for ensuring privacy and safety of files and data sent across the cross-device framework. The security of this framework could be part of a larger future works concerning this framework.

4.6 Device Selection

In order to give users the choice of which devices to distribute to inside the cross-device session, we came up with functionality and UI elements that allows users to select devices based on different criteria. At first, we had chosen

the following criteria from ideas and knowledge from the state of the art in Section 2.2. For instance, the concept of selecting devices by capabilities, as discussed in Section 4.7 was influenced by the Weave framework by Chi et al. [5].

- *All devices* in the current cross-device session
- Distribute to a specific device *by device name*
- Distribute to a specific *role* of a device
- Distribute to all devices which have a specific *capability*

This criteria changed during research and development of the project. In the prototype version of the HDUI framework, the above options were chosen. A device role was defined as a function specific for that application. For example, in a cross-device video application, we could say that there is a *video role* for displaying the video to the user and a *video controller role* for controlling the various settings of the video. While device names, capabilities and so on can be automatically gathered, device roles are specific to every cross-device application. That means that a developer needs to manually input the device roles into the application's configuration and also specify how to detect a specific role. During research it became more evident that making device roles can be confusing to both user and developer, as the underlying mechanism is not obvious through its name. We found that most device roles can be detected by the page the device is currently browsing. This made us choose to change *device roles* to *pages*. Users can quickly understand what a page is while developers can easily add the different page names in the configuration file. This means that the page detection would be automatically done by the HDUI framework without requiring the developer to implement anything except a unique page id along with a page name for the user to recognise. The latest selection criteria for general distribution are the following.

- *All devices* in the current cross-device session
- Distribute to a specific device *by device name*
- Distribute to all devices on a specific *page*
- Distribute to all devices which have a specific *capability*

An example screen displaying the user's distribution possibilities is shown in Figure 4.7. Devices' names are collected the first time a device uses the HDUI framework and stored in cookies for further usage, as discussed in

Section 4.4.

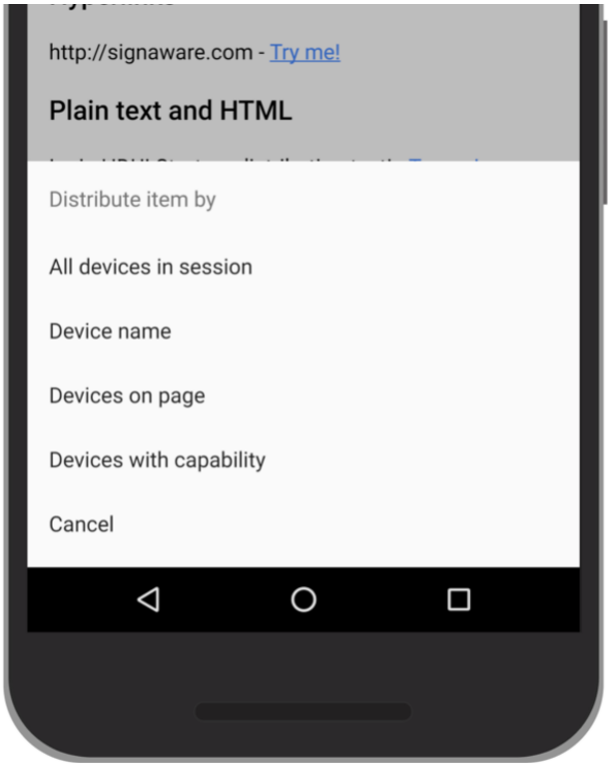

Figure 4.7: The device selection options screen shown after a user clicks the distribute icon in an HDUI-enabled app.

4.7 Device Capabilities

As discussed in Section 4.6, devices can be selected to distribute to depending on their capabilities. This feature was influenced by the Weave framework by Chi et al. [5]. Providing a way to select devices by capability is certainly interesting in the context of cross-device applications and therefore it was included in the HDUI framework.

This feature was implemented in the *HDUICapabilitiesIdentifier* class. This class contains one static manner to determine a capability, while containing a dynamic list with capabilities and their respective identification methods. An example list is shown in Listing 4.4. This list contains two items, a capability, which is part of an *Enum class*, and a method for identifying whether the device has that current capability. By providing these two informations

in the form of a list, the *HDUICapabilitiesIdentifier* can then re-use the same code for identifying the capabilities of the device and providing a final list with supported capabilities. The implementation required for such a set-up is demonstrated in Listing 4.5. The capabilities are also cached on the device's memory during start-up of the app, in order to avoid testing the same capabilities more than once. Multiple functions are provided to the developer, ensuring that the developer has full access to all the required details concerning the capabilities. For instance, the developer could request a list of capabilities using the *getDeviceCapabilities())* method, or a list of all capabilities and whether these are supported on this particular device using the *getTupleList()* method. Another method is also providing for testing single capabilities, using the *hasCapability(HDUICapability)* method. A *refreshCache()* method was also provided, it could for example, be used in case a specific capability or hardware a user needs to manually connect to the device, would not be detected and allows the developer to have full control over the system and allowing to retry finding a capability.

```
capabilityMethods : {capability, method}[] = [
                {capability: HDUICapability.Bluetooth,
                        method: this.hasBluetooth},
                {capability: HDUICapability.Camera,
                        method: this.hasCamera},
                {capability: HDUICapability.GPS_High_Accuracy,
                        method: this.hasGPS},
                {capability: HDUICapability.GPS_Low_Accuracy,
                        method: this.hasNetworkLocation},
                {capability: HDUICapability.Bluetooth_LE,
                        method: this.hasBluetoothLE},
                {capability: HDUICapability.NFC,
                        method: this.hasNFC},
                {capability: HDUICapability.Motion_Tracking,
                        method: this.hasMotionTracking},
                {capability: HDUICapability.Microphone,
                        method: this.hasMicrophone}
        ];
```

Listing 4.4: An example of a capabilities list used within the HDUICapabilitiesIdentifier class.

```
export default class HDUICapabilitiesIdentifier
{
        // Caching capabilities
```

```
4           constructor(private diagnostic : Diagnostic)
5           {
6                   this.cacheCapabilities();
7           }
8
9           private cacheCapabilities() : void
10          {
11                  var context = this;
12
13                  this.capabilityMethods.forEach((c) => {
14                          if(c.method())
15                                  context.capabilities.push(c.capability);
16                  });
17          }
18
19          public getDeviceCapabilities() : HDUICapability[]
20          {
21                  return this.capabilities;
22          }
23
24          public refreshCache() : void
25          {
26                  this.cacheCapabilities();
27          }
28
29          public hasCapability(c : HDUICapability) : boolean
30          {
31                  return (this.capabilities.indexOf(c) != -1);
32          }
33
34          public getTupleList() : {capability, hasCapability}[]
35          {
36                  var list = [];
37                  HDUICapability.forEach((c : HDUICapability) =>
                     ↪  {
38                          if(this.hasCapability(c))
39                          {
40                                  list.push({capability: c,
                                     ↪  hasCapability: true});
41                          }
```

```
42                          else
43                          {
44                                  list.push({capability: c,
                                    ↪  hasCapability: false});
45                          }
46                  });
47
48                  return list;
49          }
50  }
```

Listing 4.5: A possible implementation for usage with the capabilities list shown in Listing 4.4.

4.8 UI State Distribution Method

Next to general distribution, as explained in Section 4.5, UI state distribution is also available in the HDUI framework. This type of distribution relies on the *UI State Distribution Method* discussed in Section 3.4.1. It allows developers to customise and use specific cross-device actions when wanted. For example, an app that allows a user to control the image gallery on another device such as the one introduced by Frosini et al. [9] could be implemented using UI state distribution. All that is needed for the distribution is the ID of the image to be shown on the TV. That means, we could make use of an automatically synchronised variable as introduced in Section 3.4.1. If one variable is synchronised across devices, it would allow the developers to change the image based on the ID stored in that variable.

This allows the developers to fully control what happens, when and how it happens. In order to illustrate the rapid-prototyping aspect of this distribution method, a concrete programming example is provided in a later section of this thesis, Section 5.2.

This kind of distribution can be seen as an *implicit distribution*, as the user does not have to click the distribute button explicitly for distribution to happen. Instead, the developer has control over what happens and when, making this kind of distribution fully controlled by the developer.

4.9 Configurations

While the HDUI framework is highly customisable, it also enables the developers to quickly change the framework settings using a configuration file, kept in the *HDUIConfig* class. The first settings found are the protocol connection settings. HTML distribution, or *mixed content* distribution can also be disabled from this file, for security reasons if required. Users would still be allowed to distribute data such as text or images, but not HTML content. The developers can also opt to not request device names in the framework, as this would purely depend on the use case. For example, in the case of an app for museums, such as the example app by Frosini et al. [9], it would be disadvantageous to request device names. It would make it slower for users to connect, while it would also form a potential privacy issue. Another item that can be disabled as well is the file transfer. For the case of a museum app, it might not be wise to let users, which are probably strangers to each other, distribute files across the devices. Finally it contains the names of the pages within the HDUI app, for the distribution across pages feature described in Section 4.6. An example configuration file can be seen in Listing 4.6.

```
1  export default class HDUIConfig
2  {
3          public chosenProtocol = "HTTP";
4          public useKeepAlive = true;
5          public keepAliveIntervalMs = 2000;
6          public maxKeepAliveDelay = 500;
7          public keepAliveInitialConnectionMultiply = 5;
8          public httpServerAddress : String =
           ↪  "http://localhost:3000";
9          public connectionTimeout : number = 20000;
10         public allowHTMLDistribution : boolean = true;
11         public requestDeviceName : boolean = true;
12         public keepDeviceNameInCookies : boolean = true;
13         public enableFileTransfers : boolean = true;
14         public static pageIdentifiers = {
15                 hduiCenterPage: "hduicenter"
16         };
17         public appPages : {pageClass, nameString}[] = [
18                 { pageClass:
                   ↪  HDUIConfig.pageIdentifiers.hduiCenterPage,
                   ↪  nameString: "HDUI Center" }
19         ];
```

```
20 }
```

Listing 4.6: A shortened example of the configuration file of the HDUI framework.

4.10 Universality: HDUI Center App

While users could use an HDUI app for distributing parts of that specific app and its implemented web pages, it would not be possible to distribute other web pages. In order to solve this problem, we propose the *HDUI Center App*. This app is our solution to the problem of universality, the capability of using this framework for any existing web pages, without requiring any changes to that particular web page.

The HDUI Center App contains basic functionality found in other HDUI apps, but made in a clear manner. It allows users to type a hyperlink or text and then distribute it across the chosen devices in the cross-device session. Now in the case of a user wanting to browse another web page to distribute parts of it, it is entirely possible. The HDUI center app contains a page that allows users to type in a hyperlink. The hyperlink is then loaded into an iframe. Users then select chosen UI elements and are able to distribute it using the regular HDUI menu, as seen in 4.3, available at the top of the app. An example of this screen is shown in Figure 4.8.

While this solution seems ideal, it does have some problems that need solving. For instance, trying to access and distribute UI elements from a frame with a different origin causes a *cross-origin* error, as seen in Figure 4.9. This is called the *same-origin policy* and is blocked by most modern web browsers due to security reasons. There are specific threats when allowing cross-origin content in a web page, as discussed by Wang et al. [30]. Nebeling et al. [19] came across the same problem when developing XDBrowser and reported that a simple proxy server could be used as a solution to this problem. In order to implement this in the HDUI framework, an extra page should be made into the HDUI center app which allows for retrieving cross-origin content. This content will then need to be injected into the frame, letting the frame believe that the content came from the same origin, due to this extra proxy page, powered by another page inside the HDUI framework.

Another part of universality is that it should be easy to create an app for any web page. That is, a new HDUI project template is created with two pages. The first page being similar to the iframe page of the HDUI center

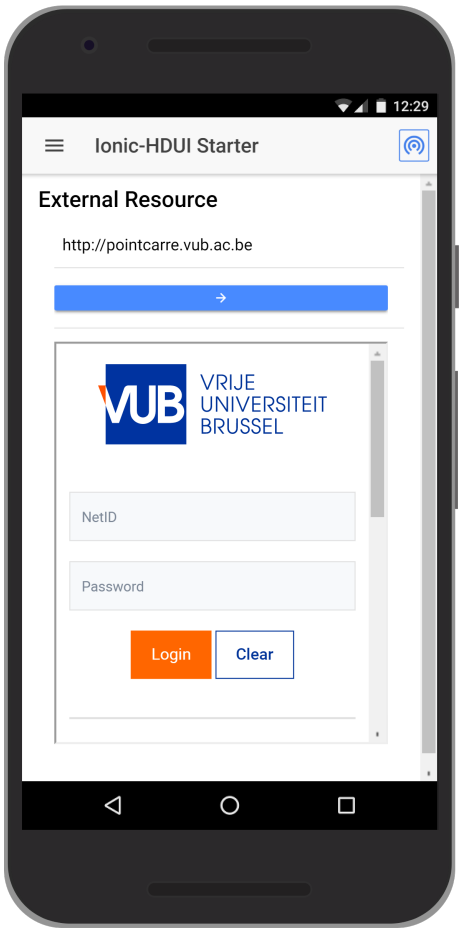

Figure 4.8: The HDUI center application, run on the HDUI framework.

app and the second page being a proxy. One difference is that the input field for a website is removed from the iframe page. The developer then needs to insert the hyperlink to the chosen web page into the configuration file and build the required Android, iOS or other platform apps. This requires very little experience and time from a developer and allows users to download the app from the respective store and distribute its elements across devices. This template project is provided as part of the framework.

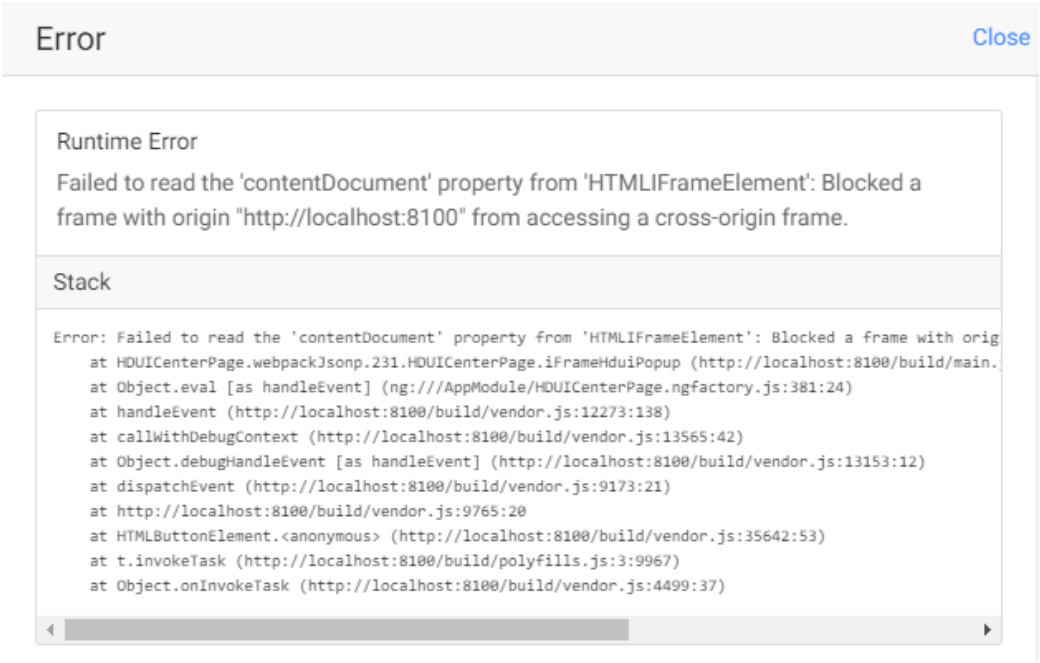

Figure 4.9: A cross-origin frame error thrown when a user tries to distribute content from a different origin.

4.11 Fulfilment of Requirements

The framework needs to fulfil particular requirements defined in Section 3.2. The HDUI framework implements a solution for all the proposed requirements. In this section, we discuss the implementations of the individual requirements and how these were fulfilled. Some of these are automatically fulfilled by the use of a specific technology. For the sake of completeness, these are also included and discussed in this section.

4.11.1 R1. Based on Hybrid Framework

The HDUI framework was implemented using the Ionic framework and thus automatically fulfils this requirement.

4.11.2 R2. Native Functionality and Data

The hybrid apps area is an area that has not been thoroughly explored for cross-device applications. Most cross-device application frameworks make use of programming languages such as JavaScript (web-based) or Java, as discussed in Section 2.5. While some of these languages are multi-platform, they still impose restrictions to the developer in some ways, since the access to the device's native capabilities can be limited, such as the access to a device's contacts list, as discussed in Section 2.5. Hybrid apps and frameworks solve this problem by introducing a container, which runs natively on the platform. This container gives access to the native functionalities of the device

Native functionality and data are also accessible with the use of the Ionic framework. Ionic contains specific plug-ins, such as the contacts plugin[4], allowing developers to access and retrieve the contacts list of a device. Since the HDUI framework is based on the Ionic framework, this requirement is fulfilled.

4.11.3 R3. Rapid and Developer-Friendly

This requirement, allowing for rapid development is fulfilled in various ways. First of all, this framework implements the proposed DUI distribution methods as explained in Section 3.4.1. Due to this categorisation, some distributions can be implemented in an automated manner.

This DUI framework is based on the Ionic 3 Framework, which is a hybrid app framework. A hybrid app framework is advantageous in this situation, since it would allow the developer to create apps for multiple platforms with just one codebase, effectively allowing for rapid development. Another reason why the framework allows for rapid development, is that the framework's template includes everything for building the app on a multitude of platforms, including the necessary building code for building Electron applications[5], which can be used for running Ionic apps on desktop environments. Electron creates a sort of container application with a web browser that opens up the Ionic app, which is a web-based app. Next to this, default configuration files for building an HDUI app to iOS or Android are also provided. While it is true that some other cross-device application frameworks contain one codebase for multiple platforms as well, such as web-based cross-device ap-

[4] `https://ionicframework.com/docs/native/contacts/`
[5] `https://electron.atom.io`

plication frameworks, these lose the ability to have full access to the device's native capabilities and resources, which is not lost with the use of the Ionic framework. The fact that the Ionic framework has the platform continuity feature allows the developer to have the lay-out of the application fit the user's environment automatically, which in turn also contributes to the rapid development of the HDUI framework.

An important aspect of the development phase which is also solved, is the debugging. Husmann et al. [13] state that there are not enough tools for the development of cross-device applications. This is why tools such as XDSession [20] and XDTools were developed, in order to be able to emulate multiple devices in one tab. This cross-device application framework however, does not require any extra tool. A web browser with developer tools such as the Chrome browser and its developer tools is enough, since Ionic apps are web-based apps wrapped into a Cordova[6] container, enabling the application to access the native capabilities of devices while still running on a multitude of platforms and environments. It is interesting to note that developers can use their own preferred IDE to develop cross-device applications using the HDUI framework. As long as the IDE supports web programming languages such as Javascript and Typescript, it will work fine. This is advantageous, because that means developers can use tools they are already familiar with, effectively reducing the learning curve for developers to start using the HDUI framework. Multiple devices can be simulated by the use of multiple browser windows. During the development of the HDUI framework, however, some issues were found with the use of multiple tabs inside the same Chrome window. Unexpected behaviours occurred such as data from one tab interfering with the other. However, there is a simple solution to this problem. When using multiple chrome windows, this issue cannot be reproduced. Therefore, the HDUI framework does not require any extra development tools such as an IDE, debugging tools or device emulators to be implemented specifically for the HDUI framework, effectively contributing to the rapid aspect of the framework.

In order to illustrate the ease of use and rapidness of the HDUI framework, a demonstration and usage explanations of the HDUI framework are provided in Section 5. The framework was also evaluated and the results are discussed in Section 6.

[6] `https://cordova.apache.org/`

4.11.4 R4. Device Agnostic and Universal

Since the HDUI framework was built on top of the Ionic framework, it is automatically device agnostic. Using the same codebase, applications can run on multiple combinations of environments without requiring any extra changes for different environments. As for running the application on desktop devices, a desktop-container such as Electron, which packages the HDUI application for desktop platforms, could be used. An existing Electron configuration is provided within the framework to help kick-start the development using the HDUI framework.

Another part of this requirement is that the framework should be universal. By this, we mean that existing technologies and applications should be able to be used within the HDUI framework, so as to avoid reinventing new frameworks that do not work with current apps. In order to solve this problem, two items were developed. First of all an HDUI center app was made that lets users input the specific website they want to browse. With the use of iframes, the website is loaded into the app and parts of it can be distributed by the end-users inside a cross-device application session. While iframes come with some limitations that needed to be solved, this is further discussed in Section 4.10. A second item developed is a starter template for the HDUI framework that lets developers easily integrate an existing website into a cross-device application. The developer simply needs to add the URL of the website into the configuration of the framework in order to make this work. The app will then automatically load that specific website and let users distribute it across the devices in the cross-device sessions. Currently this still requires a connection to a server for the cross-device sessions, but in the future other protocols could be implemented removing this need, or a shared connection pool server could be used for multiple applications.

4.11.5 R5. Protocol-Independent

The framework should be flexible enough to let developers easily change the protocol from one specific protocol to another. The implementation of the exact protocols is explained in-depth in Section 4.2.

Few of the popular cross-device application frameworks take protocol-independence into account. However, being protocol-independent brings some advantages to the framework. It allows the framework to be utilised for many different cross-device purposes using a broad array of different protocols. It also enables developers to easily and rapidly switch from one protocol to another

without having to switch to another cross-device framework due to protocol-compatibility problems.

The implementation of protocol-independence in the framework depends on the architecture of the framework. Some steps had to be taken in order to make the framework protocol-independent. First of all, the protocol-specific code was developed in the same architectural layer of the framework. Interfaces and contracts were written with the prototypes of the functionality that the developer must program in order to have the protocol functional within the HDUI framework. With the goal of minimising the work for developers using the framework in mind, only the most basic functionality of the protocol needs to be implemented for bringing it to the framework. Other layers in the framework then use a factory design pattern for instantiating the protocol. The developed communication factory class automatically instantiates the correct protocol-related classes depending on the chosen protocol, which is set up in the framework settings. Developers can ignore the fact that the framework is protocol-independent and simply make use of the communication factory whenever a protocol-related class needs to be used. If the developer chooses the HTTP protocol, then all communication classes ending with the *HTTP* keyword will be used, while this suffix would be different for every protocol. For example, *ProtocolHTTP*, *ProtocolHTTPAddress* and so on.

Since web browsers do not allow us to create a WebSocket(Socket.IO) server within a web browser from a client machine, an extra server was required in-between to handle the connections and keeping a connection pool. In order to keep the framework rapid for development, a standard class is provided which can be used to implement most of the required functionality of such a server. This means that when another protocol is implemented which also requires a server, this class can be re-used and time can be saved. When implementing a new protocol which does not require such a server, the developer could re-use this code inside a client of the HDUI framework, or simply not need a connection pool depending on the connection strategy of the chosen protocol.

The following are the steps that a developer must take in order to develop a new protocol for the framework. First of all, the basic classes for the protocol must be created (such as protocol, protocol address, protocol message and protocol client) with basic protocol functionality (connect, disconnect, send message to and listen for messages). In order to allow devices to join the same session when pressing on the join button at a similar time, the session connector class provided by the HDUI framework is re-used, which

is protocol-specific as well. The abstract session connector class must be extended with 2 extra methods. First, a method to add event listeners to received messages, specific to the protocol and secondly, a method to send messages, specific to the protocol. This has to be developed again, since the session connector can be used within, or outside of the framework, and because the steps performed initially might be slightly different for every protocol (due to the way different protocols require some sort of initialisation).

This process was used for adding the Socket.IO protocol to the framework. The Socket.IO protocol was implemented with less than 70 lines of code (including the whitespace for clarity).

4.11.6 R6. Detect Device Capabilities

The detection system consists of one base class, the HDUICapabilitiesIdentifier class. Another important class is the HDUICapability class which contains an enum of possible capabilities. Simply by adding an entry into a list in the HDUICapabilitiesIdentifier class, containing two items, a capability from the the HDUICapability enum and a method for verifying whether this capability is present, the system will automatically start detecting the capability and cache it onto the device. Cache can be forced if required. For instance, for a capability such as Bluetooth, it is better to force the cache to be updated since the user could turn on or off Bluetooth functionality at any point of time.

This means that the HDUICapabilitiesIdentifier will automatically generate a list of all capabilities and whether these are supported on the device, upon initialisation of the class. The capabilities cache can then be re-used throughout the application.

The following procedure explains how to add a new capability into the framework.

First, add the capability to the HDUICapability enum, as shown in Listing 4.7.

```
1 enum HDUICapability {
2         Bluetooth,
3         ...
4 }
```

Listing 4.7: Add an entry to the HDUICapability enum.

The next step is to add an entry to the HDUICapabilitiesIdentifier class with both the capability's enum entry and the corresponding verifier method. This is illustrated in Listing 4.8.

```
1 private capabilityMethods : {capability, method}[] = [
2                 { capability: HDUICapability.Bluetooth, method:
                  ↪ this.hasBluetooth },
```

Listing 4.8: Add an entry to the HDUICapabilitiesIdentifier class.

The capabilities identifier class will then automatically create the list of all capabilities in multiple formats, ready for the developer to use. The following Listing 4.9 shows some of the possible methods a developer can use for handling the device's capabilities.

```
1  // List of capabilities
2  public getDeviceCapabilities() : HDUICapability[]
3
4  // Force a cache update
5  public refreshCache() : void
6
7  // Checks whether the capability is available
8  public hasCapability(c : HDUICapability) : boolean
9
10 // Returns a tuple list of <capability, has capability?>
11 public getTupleList() : {capability, hasCapability}[]
```

Listing 4.9: The available methods in the HDUICapabilitiesIdentifier class.

Users can distribute elements to devices with specific capabilities. An image gallery could be distributed to a TV while the controls for this could be distributed to another mobile device.

4.11.7 R7. Extensible

Originally, the requirement for extensibility of the framework was added in order to solve the problem of updates. Whenever an update is available for the HDUI framework, developers should not lose their changes made to the

inner-workings of the framework. This could be solved by a system for extensibility such as a plug-ins or modules system. However, since the focus shifted throughout the research and development of the HDUI framework towards more cross-device related concepts and functionalities, this requirement was not implemented. This requirement is currently on hold for future works.

4.11.8 R8. Minimal Set-Up and Run Requirements

The run requirements were kept as simple as possible. Unfortunately, for the WebSocket(Socket.IO) protocol an extra server is required for handling the connections. This cannot be bypassed due to the way the technologies are limited for interacting with each other, as explained in 4.2. This problem could however be solved in the future, or by utilising different protocols. Apps can be deployed to the application stores of the relevant platforms and be run directly without requiring extra configuration and thus contribute to the rapid-development aspect of the framework.

4.11.9 R9. User-Friendliness

The HDUI framework provides a consistent customisable user interface for the user, across all pages and functionality of the cross-device framework. Participants of the evaluation in Section 6 will also be asked about the user-friendliness in order to rate it and analyse how it could still be improved.

4.11.10 R10. Platform Continuity

While important for the usability and user-friendliness of the applications, the Ionic framework [7] already provides us with platform continuity. Apps will automatically have the look and feel relevant to the system used for running the app. This is handled by the Ionic framework.

4.11.11 R11. Sufficient Developer Tools

Being based on the Ionic framework, Typescript[7] and AngularJS[8], applications for the HDUI framework can be developed using any developer's favourite Javascript IDE.

Testing and debugging can then be done from any developer's favourite Javascript IDE as well, or by using the developer tools available in the Chrome browser, for example.

4.11.12 R12. Sufficient Cross-Device Functionality

Different features following the concept of distribution methods were implemented, in order to give control over the cross-device aspects to the developers. Users can automatically distribute items across devices by themselves, or distribution can be triggered with the help of logic implemented by developers which would utilise the cross-device functionality implemented by the HDUI framework. This could mean, synchronised variables or remote function triggers depending on the user actions and the device's capabilities. Most features are thoroughly explained in Chapter 4.

4.11.13 R13. Cross-Platform

Being based on the hybrid Ionic framework, the applications run on a number of different platforms and environments, including desktop environments, with or without the use of an extra desktop-container such as Electron[9].

4.11.14 R14. Availability of the Framework

In order to make cross-device application frameworks more accessible to developers, the HDUI framework will be made available open-source as soon as its documentation is fully ready. A documentation will be made in order to kick-start the development of new cross-device applications by developers.

[7] `https://typescriptlang.org`
[8] `https://angularjs.org`
[9] `https://electron.atom.io`

4.12 Conclusion

Compared to other available research, this research takes not only the user experience into account, but the developer's experience as well (requirement R1). An in-depth look is taken at both of these, in order to ensure that it is easy and quick to implement, as well as it is to use. A third aspect is the link between the developer's experience and user experience. Valuable time is saved for the developer by automating tasks with default user interfaces and functionality, while a consistent and usable user interface is presented to the user. While some parts are automated (such as finding devices to distribute to), these can still be easily customised by the developers (custom functions and look and feel). Concepts were proposed and implemented in order to divide the possible cross-device actions into categories in order to maximise the possible automation within the framework.

Another innovative feature is the use of a Hybrid framework for implementing a cross-device interaction framework. For example it will remove the need for extra development or debugging tools, which other DUI frameworks need. For example, the Weave framework [5] and XDStudio [21] require the use of their own GUI builders. Another framework requires the use of an extra debugging tool. By using a hybrid framework, the tools that already exist and most developers know can continue to be used.

A lot of the previously mentioned requirements are innovative as well. For instance, one of the innovative features is the ability to make a protocol-independent framework, which allows the developer to easily and quickly plug-in their own protocol implementation, which to our knowledge, cannot be found in the currently available cross-device frameworks.

5 Usage

In this chapter, the usage of the HDUI framework is shown in order to demonstrate the rapidness and ease of cross-device application development using the HDUI framework. Aside from demonstrating the usage, some demonstration apps are explained as well.

While the focus is put on the manners for cross-device distribution, it is important to notice that the sequence for connecting and disconnecting devices, as well as a protocol for handshakes is already included as well and is fully automated as part of the HDUI framework. Next to this, users can automatically distribute selected general data, HTML or even files across the selected devices in a cross-device session. This gives the users the freedom to distribute parts of applications as wanted, without requiring any developer implementation or intervention.

5.1 General and UI Components Distribution

In the context of general and UI components distribution, the framework is automatic in the sense that a developer needs to follow the template for

creating new pages, which automatically inherits the general and UI components distribution capabilities. The required code and template for a new page in the HDUI framework is explained in Section 4.1.

Developers do not need to do anything in order to include this functionality except starting from the provided templates which were kept as minimal as possible with the provided technologies. An HDUI page automatically contains the HDUI menu, as discussed in Section 4.3, which allows end-users to distribute exactly what they want at any time during a cross-device session.

Extending the Predefined Structured Data for General Distribution

While the general distribution method contains the capabilities to distribute some predefined structured data, it also allows the developer to easily extend and create new predefined structured data types. In order to so, the developer needs to define a way to detect whether specific data is from that data type in the *HDUIGeneralTypes* class, as well as create a template as to how that specific data should be shown to end-users in the *app/pages/general* directory.

5.2 UI State Distribution

UI State distribution works differently, since it is semi-automatic and the developers need to implement the expected behaviours and business logic for their desired applications.

Remote Function Triggers

UI State distribution is implemented in a few different ways in the HDUI framework. First of all, functions can be triggered remotely from one device to other devices within the same cross-device session. These are called remote triggers within the framework and work in the following manner. One trigger carries a name and a function to be triggered. These triggers are initialised on every device in the cross-device session on the corresponding pages. These can also be initialised on every page of a cross-device application, depending on the wishes of the developer. When one device chooses to trigger the remote function on another device, it sends the name of the trigger towards that specific device, along with a tag for identifying that is a function to be triggered. That device then searches through its trigger database for finding

that specific trigger name and executes the corresponding function.

The first step for the developer is to let devices listen to this trigger by initialising it in the application as demonstrated in Listing 5.1.

```
        // The method provided by the HDUI Framework
        HDUIController.registerTrigger(<trigger name>, <trigger
        ↪  function>);

        // An example use of this method
    setupTriggers = () =>
    {
        // Register triggers for this page
        var context = this;
        var aTriggerFunc = () => {
                        context.updateField();
        };

            (this as
            ↪  HDUIController).registerTrigger('btnAction',
            ↪  aTriggerFunc);
    }
```

Listing 5.1: Registering a trigger using the HDUI framework

The final step is to allow these functions to be triggered, as shown in the following Listing 5.2.

```
// The method provided by the HDUI framework
HDUIController.runTrigger(<trigger name>);

// In practice
this.runTrigger('btnAction');

// Or as a button on a page
<button (click)="runTrigger('btnAction')">Run Trigger</button>
```

Listing 5.2: Triggering a remote function using the HDUI framework.

The framework will then automatically trigger the remote function on devices in the same cross-device session, allowing for synchronising specific actions across devices.

Synchronised Variables

While triggering remote functions might not be enough, another method is provided within the HDUI framework, namely synchronised variables. It allows the developer to synchronise variables across devices in the same cross-device session in a manner similar to the previously discussed remote triggers. The available methods for this are shown in the following Listing 5.3. It is noticeable that this is very similar to the previously discussed remote triggers, with the addition of a variable, which is sent and received across the devices in the cross-device session.

```
1 // Listener (receiver) method
2 HDUIController.registerVarTrigger(<variable name>, <trigger
  ↪ function>);
3
4 // Distributer (sender) method
5 HDUIController.distributeVariable(<variable name>, <value>);
```

Listing 5.3: Triggering a remote function using the HDUI framework.

It is interesting to note that the variable synchronised across devices can be an object, allowing the developer to synchronise multiple variables at once instead of being required to create multiple trigger functions and synchronised variables.

Triggering remote functions can be useful for example, for changing the image shown on a device to the next image or previous image. Synchronised variables could be used for example, for synchronising the time of a video across devices.

A summary of the available UI State methods are shown in the following Listing 5.4.

```
1 // Remote function triggers
2 HDUIController.registerTrigger(<trigger name>, <trigger
  ↪ function>);
3 HDUIController.runTrigger(<trigger name>);
4
5 // Synchronised Variables
6 HDUIController.registerVarTrigger(<variable name>, <trigger
  ↪ function>);
7 HDUIController.distributeVariable(<variable name>, <value>);
```

Listing 5.4: Triggering a remote function using the HDUI framework.

5.3 Protocol Implementations

As stated in the framework requirements 3.2, the framework should be protocol-independent. The HDUI framework has been made protocol-independent, in the way that a developer can easily switch from one protocol such as *WebSockets* to another such as *Bluetooth* by changing one word in a configuration file as shown in Appendix A.

While it is easy for developers to switch from one protocol to another, the protocols needs to have an implementation ready. For that matter, we have minimised the required code for new protocols to be added to the framework.

In order to add a new protocol, a developer needs to create a minimum of two classes. One is a *Protocol<Name>* class with the basic protocol connection functionalities and the other is a *Protocol<Name>Address* class, which allows the developer to keep the address of a device in a manner specific to that protocol. These classes are then automatically chosen and utilised within the framework to make the communications to function seamlessly.

The *Protocol<Name>* class needs to extend the *GeneralProtocol* class and implemented a few abstract methods, defined by the GeneralProtocol class. These abstract methods are kept minimal and can be found in Listing 5.5.

```
1  // Connections
2  abstract init(): void;
3  abstract disconnect_protocol(): void;
4
5  // Sends broadcast message (not to the device itself)
6  abstract sendMessage(msg: ProtocolMessage) : any;
7  abstract registerMessageListener(tag, listenerFunc : ((msg? :
   ↪  ProtocolMessage) => any)) : void;
8
9  // When in active session
10 abstract getSessionId() : String;
11 abstract getSelfAddr() : GeneralProtocolAddress;
12 abstract getServerAddr() : GeneralProtocolAddress;
```

Listing 5.5: The abstract methods required to be implemented by protocols in the framework.

The abstract methods are kept very general so that these would fit a multitude of different protocols. The *init()* method is there for the developer to

run any initialisation code required for the protocol. In case a connection is required through an intermediary server, this connection can be set-up in this method. The *disconnect_protocol()* method is used for properly disconnecting and disposing of variables. The next method, *sendMessage(msg: ProtocolMessage*, is used for sending messages to other devices, but not to the device itself.

While it is now possible to connect, disconnect and send messages to a protocol, we also need a manner to listen to messages sent to this device. In order to solve this, the emphregisterMessageListener(tag, listenerFunc) abstract method was added. This allows the framework to listen for specific tags and in case such a tag is received along with a message, the listenerFunc can be run on this data. Behind the scenes, the framework uses a list of different tags, as defined by the *GeneralProtocol* class. Some examples are, *confirmRegistration*, *requestClientDetails*, *deviceDisconnect*, *receiveGeneralData* and so on. However, knowledge of these is not important for the developer, unless the developer wants to extend the framewor, as these are internally handled by the HDUI framework. Finally there are three abstract methods left for querying some general data about the connection, such as this device's unique address, the unique server address in case there is one and a unique cross-device session ID.

It is noticeable that there are relatively few methods required to get a protocol running. A definition is required for connecting, disconnecting, sending messages, receiving messages and finally for gathering some data about the connection itself. The Socket.IO protocol has been implemented following this pattern and the final implementation resulted in a file with less than *95 lines*, including whitespace and comments for readability. This implementation can be found in Appendix B.

In the case that an intermediary server is required, it will need some logic for managing the connections as well. A *HDUISessionConnector* class was made that can be used on any intermediary server, which will handle most, if not all of the logics required for the intermediary server. This file however, does not need to be used only on intermediary servers but can be used in any point of the connections, following the protocol's requirements. This class is also extended in the case of the intermediary Node server for the Socket.IO protocol. The Node server ended up having less than 45 lines, while still managing the connections in the required ways. There are only two abstract methods that an intermediary server needs to implement, as shown in Listing 5.6. The full implementation of this intermediary server is

shown in Appendix C.

```
        // Tag is the type of the message, obj is the data
        abstract sendToClient(tag, id, obj);

        // Method to add 3 required event listeners
        abstract addEventListeners();
```

Listing 5.6: The abstract methods required to be implemented by the intermediary server in a protocol.

5.4 HDUI Demonstration Applications

There are many different possibilities with the HDUI framework. Some examples were already discussed in Section 1.1. Some of these example apps were developed as a way to demonstrate the potential of the framework as well as its rapidness for development.

5.4.1 Remote Image Gallery

In this application an image is presented as well as two buttons for getting the previous or next image to be shown. When the devices are in an active cross-device session, the active image is automatically synchronised across the devices in the cross-device session. This means that any device in the session can control the images shown, while the images are displayed on other devices such as TVs. This example was implemented with the image and the buttons in the same page, however one could also choose to separate these. Making one page with the image full-screen while another shows the controls is certainly also an option and it was also implemented as another example.

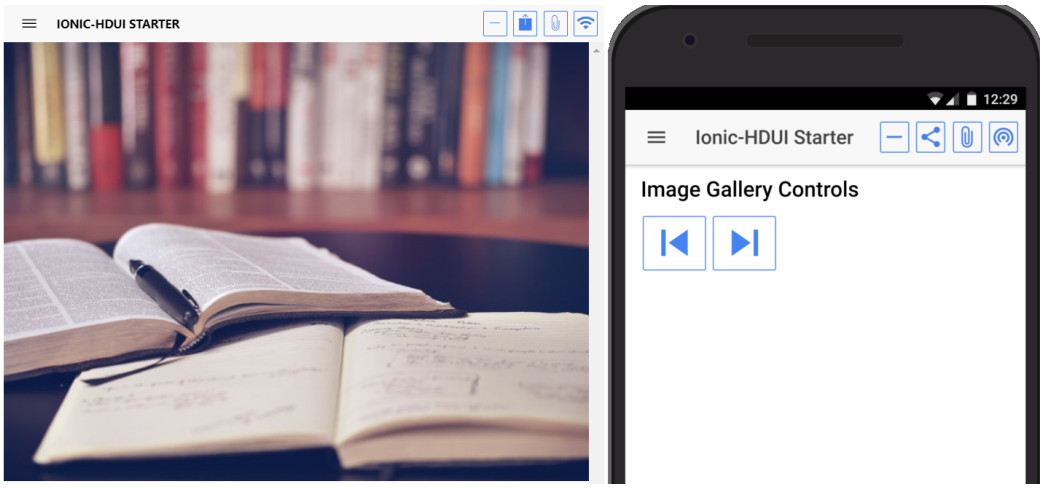

Figure 5.1: A large screen device (left) displaying the image gallery controlled by the mobile phone (right) in the same cross-device session.

In order to implement this, an image index is kept synchronised across devices using the SyncDB of the UI State Distribution, as explained in Section 4.8. Automatic triggers are then set-up to change the image once the synchronised value is changed. The required trigger and callback methods setup for this example application is shown in the following Listing 5.7.

```
1  setupTriggers = () =>
2  {
3      // Register L1 triggers for this page
4      var thisScope = this;
5      var imgTriggerFunc = (data : TriggerData) => {
6
7              if(data.value > 0 && data.value <= this.maxImg) {
8                      thisScope.gallery.nativeElement.src =
                        ↪  "./img/gallery/" + data.value +
                        ↪  ".jpeg";
9                      thisScope.currentImg = data.value;
10                 }
11         };
12         (this as
            ↪  HDUIController).registerVarTrigger("imgGallery",
            ↪  imgTriggerFunc);
13 }
```

Listing 5.7: The setup of the UI state triggers for the example HDUI gallery application.

The buttons to change the image send the new value of the synchronised image index to the server, so that it can then be propagated to all client devices in the cross-device session. The code for changing the value of the synchronised variable is shown in Listing 5.8. The demo application can be seen in Figure 5.1.

```
1 this.distributeVariable("imgGallery", prev);
```

Listing 5.8: Changing the value of a synchronised UI state variable in the example HDUI gallery application.

This means that for implementing such an application as a developer, the following things need to be done. For changing the image from a mobile phone, a message needs to be sent across all devices, using one simple method provided by the HDUI framework. For displaying the chosen image on another device, the device will need to listen to changes to the image index variable. A callback function needs to be made that changes the image depending on the received data. Using a simple method provided by the HDUI framework, this callback method is automatically called upon change of the synchronised image index variable.

5.4.2 Applications for Education

Also described in Section 1.1, cross-device applications could be useful for education. A teacher could input questions, while students could input their answer. The teacher could be able to see the answers from the students in real-time, along with their overall score. Students could have an extra button along with questions to request for assistance from the teacher. In order to demonstrate the potential of the framework, as well as its rapidness, this demonstration application has been developed. The result is an application developed using the HDUI framework with a total of about 270 lines of Javascript code, including comments and enough whitespace for readability. Another 170 lines of code were written for the lay-out of the application, but do not contain the application's logic as these are purely views. The application was developed by one developer in roughly one hour.

Some advantages such as the automatic connection between devices, the automatic management of sessions, the automatic redundancy and reconnection possibilities whenever the network or a device could fail can be noticed from

the use of the framework for developing cross-device applications. The developer does not have to worry about how data is sent across devices, how these devices should connect and redundancy such as a keep-alive system does not need to be implemented, since those items are already provided by the HDUI framework.

The application consists of two screens, one for teachers and one for students. The students are first prompted to enter their student ID after connecting to the cross-device session with their teacher and other students. Afterwards, they are prompted to wait for the teacher to send, or distribute, a question towards them. Once a student receives a question, an answer can be provided, or assistance can be requested from the teacher. Whenever the student requests assistance from the teacher, a message will show up on the teacher's page showing that a particular student has requested assistance from a teacher and is currently waiting for that. Once an answer is submitted, the system corrects the user if required and shows an explanation for the answer. Teachers can then share another question to the students. This is an example application and in the future, this could be made into a question list as this would be more practical and students will not have to wait for teachers to send particular questions. The students' screens can be seen in Figure 5.2.

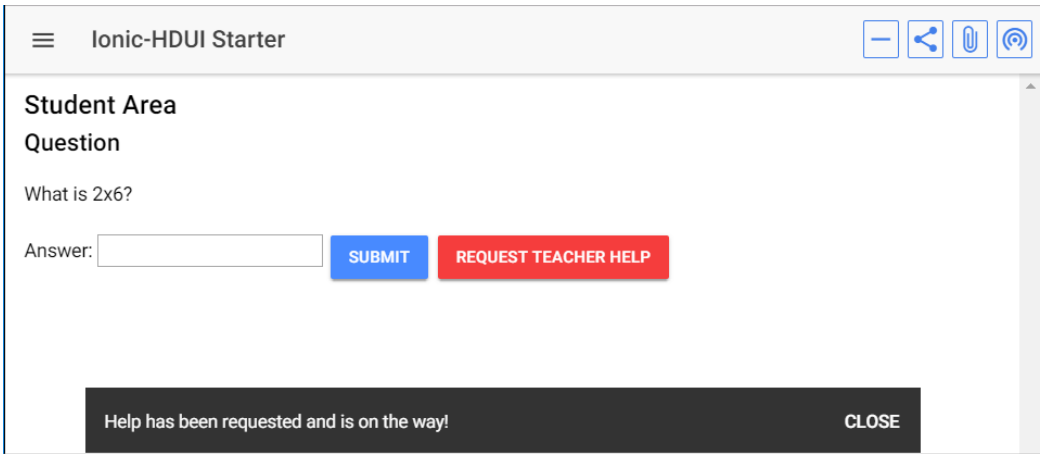

Figure 5.2: The students' control panel for the cross-device education application demonstration.

The teacher's screen can monitor most of the students' activity in real-time. Students' answers and assistance requests are shown along with a score for each individual student. A set of questions are also shown to the teacher, in order to give them the ability to send or distribute these questions to the

students. This can be seen in Figure 5.3.

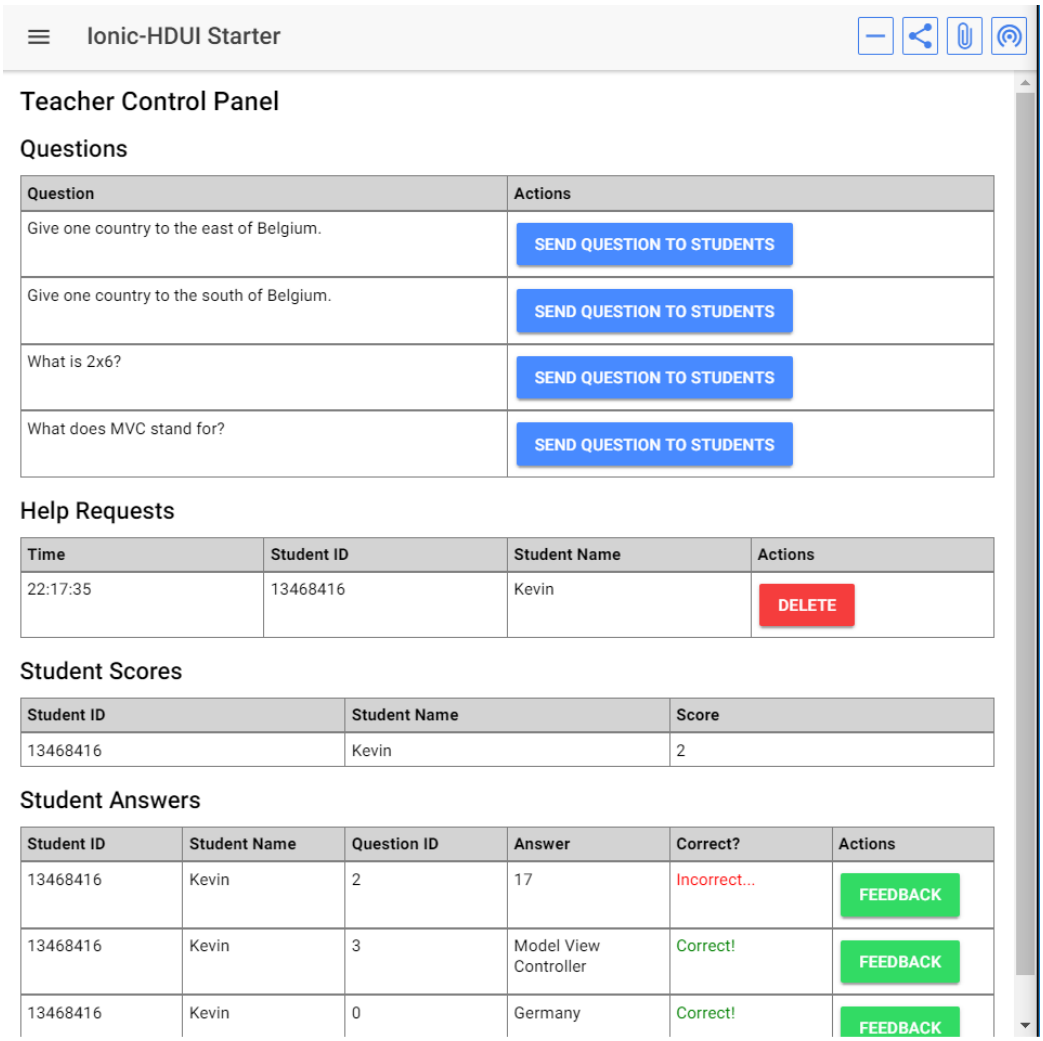

Figure 5.3: The teachers' control panel for the cross-device education application demonstration.

Another use of this application would be for quizzes. Quiz moderators would be able to see all answers in real-time and view the current score of competitors as well, while being able to select a question to let the contestants answer. Contestants would only receive the question once it is sent by the quiz moderator. Quiz contestants could use one device per group, such as a tablet or mobile phone to fill in the answers.

5.4.3 Extended Displays

As discussed in Section 1.1, a possible use of the HDUI framework would be for extended displays in games. One could show the main game on one or multiple screens and have the game scores, a map, user equipment, instructions or other user information on another screen. In order to demonstrate how this could be made, such an extended display was implemented as part of the demonstration. Instead of developing a game simply for the purpose of demonstrating the extended displays capability, we have taken the concept of the education application previously discussed in Section 5.4.2. The application could be considered to be used in schools, or for quiz competitions.

Where a student or quiz contestant would have one device for filling in the answers, a second device could be opened on a different page of the framework, which shows the scores of the different users in real-time. By connecting the device to the cross-device session, communications are allowed across the session and data such as scores can be synchronised to the additional device for creating an extended display effect.

This was rapidly added into the education application for demonstration purposes. The additional displays' screen can be seen in Figure 5.4, while the previously existing main screens are shown and discussed in Section 5.4.2. The lay-out could be improved for this use-case, but the purpose is to demonstrate the functionality possibilities and the different possible usages of the HDUI framework.

5.4.4 Further Notes

Do note that the previously mentioned example applications are proof of concepts that were rapidly sketched and implemented in order to demonstrate the rapidness and capabilities of the HDUI framework. With more creativity and time, more extensive and interesting applications could be built using this framework.

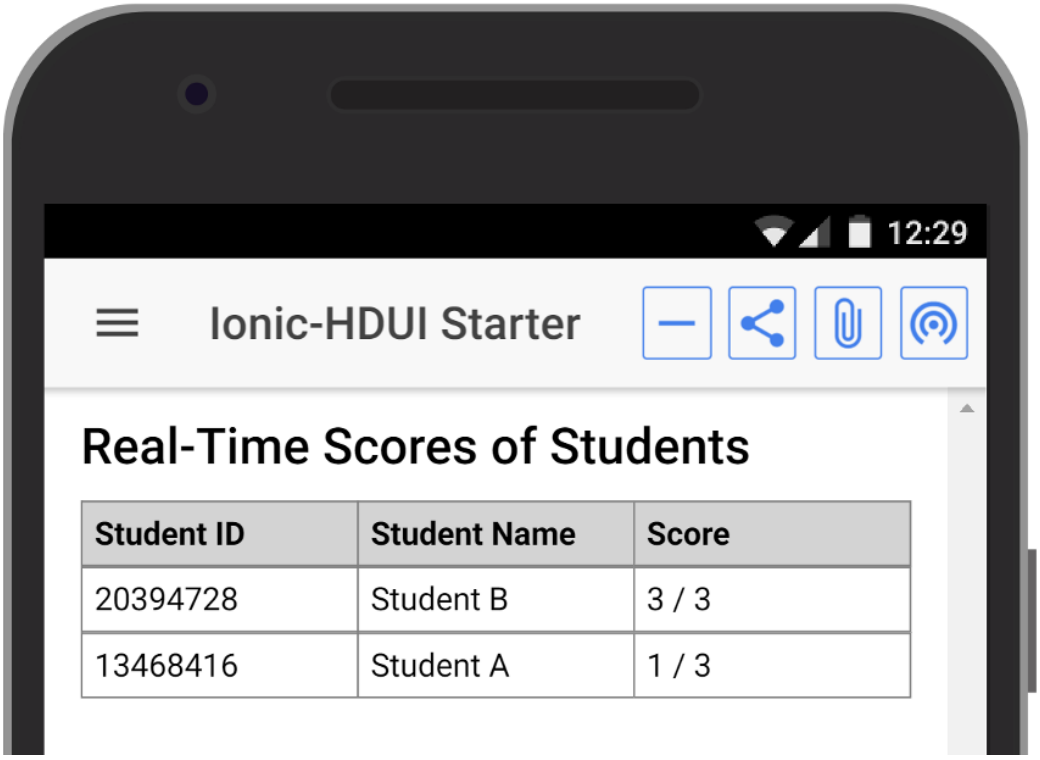

Figure 5.4: The scores page, which can effectively be used as part of an extended display for one particular application, per student or per quiz group.

5.5 Conclusion

In this chapter, potential usages and examples of the framework have been demonstrated. Furthermore, the rapid-development capability has been made clear with the given code examples for the different distribution methods previously defined.

6 Evaluation

An experiment was conducted as part of the research of the HDUI framework. The methodology, initial hypothesis and the results are further discussed in this chapter.

6.1 Hypothesis

Our hypothesis is that the HDUI framework helps developers rapidly develop cross-device applications. The HDUI framework makes it easier to develop cross-device applications with the use of the simple and mostly automated cross-device distribution methods.

In order to test this, an experiment is conducted that measures the speed of developers to implement a simple cross-device application. The participants will be observed and also asked questions before and after the evaluation.

6.2 Methodology

Since we want to measure whether it is true that the HDUI framework help improve the speed of developers to implement a simple cross-device application, the participants will be observed and their speed and ease of development will be measured. They will be asked questions before the evaluation to assess their technical capabilities. Another round of questions will be asked post-development, in order to assess the ease of use of the framework and potential ways the framework could still be improved.

These questions and observations can be divided into both a qualitative and a quantitative study. The qualitative study consists of observing the participants developing the simple cross-device application. After the development, some questions are asked such as, how do you feel about the framework and how do you believe the framework could be improved (these are part of the questions in appendix D). The result will be in the form of words. The quantitative study on the other hand will mostly test numerical values that can be used for a statistical report. An example of that would be, rate the rapidness of development using this framework for the development of cross-device applications, on a 5-level Likert scale.

The observation consists of asking the participants to implement the missing cross-device functionality for a simple remote controlled image gallery application, such as the one demonstrated in Section 5.4.1. The participants are supposed to be using the functionality of the HDUI framework in order to implement the cross-device functionality. An excerpt of the framework's documentation is shown to the participant before the start of the qualitative study. For this qualitative study, the participant is provided a starter version of the HDUI framework, including the basic HDUI templates for the required pages to implement. The HMTL for the required pages is also provided, since the HTML parts do not relate to the cross-device functionality and are not an important part of the experiment. That means the application can already run and show an image on one screen and the remote control buttons on the other screen. As for the back-end code, an initial HDUI template code is provided along with a function run once the button has been clicked, since the linking of events to buttons is not part of the framework, but is part of the Ionic and AngularJS frameworks, which are not the topics of this experiment. The participants need to implement the required functionality for these buttons, in order to let the buttons on the remote control page change the image displayed on the image gallery page. The gallery page needs the subsequent functionality for receiving these updates and actually changing

the images on the screen when requested by the remote control. The location of the images are provided as a comment within the source code. The participant does not need to worry about the intermediate server, since this server will already be started and available to use by the participant.

The structure of the evaluation is as follows. First, the participant is given a document to read together with the evaluator. The participant can ask cross-device related questions at any time, except during the actual development, in order to keep the results consistent across the different participants. After reading the document, the basic HDUI functionality is demonstrated along with a working version of what the participant is expected to implement. The source code is not shown in order to keep the results of the study fair and consistent. Before starting the qualitative study, the participant is asked to fill in a pre-evaluation survey. The participant can then begin implementing the functionality for the cross-device application. Upon finishing the implementation, the participant is then asked to fill in a post-evaluation survey expressing the ease of development and whether the requested goal was achieved. Extra remarks can be given at any time. The evaluator observers the participant during the development and makes notes on whether the participant seems to have easily understood the cross-device distribution methods and concepts introduced by the HDUI framework.

The individual evaluation results can be found in Appendix E.

6.3 Participants

The evaluation took place with 5 different participants. The evaluation was individual and occurred using a screen and voice sharing software, Teamviewer. A Windows environment specifically for the evaluation of the framework was set-up, where the participants would have access to the instructions file, a Javascript IDE of choice. The necessary set-up and installations had already been done for the participants. The evaluator takes charge of running the application whenever the participant finished implementing the application.

The 5 participants all have a background in Information Technology. Some of these are developers, while others are Infrastructure Engineers with a passion for development as well. Their skill is measured in the pre-evaluation survey in order to make this count in the statistics as well.

6.4 Results

The results are split up into two sections, the results from the qualitative study and the results from the quantitative study. The answers from the participants can be found in Appendix E.

6.4.1 Qualitative Study

The results from the quantitative study come from observing the participant's develop a simple cross-device application using the HDUI framework, as well as from some of their answers during this observation study.

3 Participants have a bachelor in applied informatics, while 2 participants are studying for a master in Computer Science, at this time of writing. 2 of these participants are focused on becoming developers, while others are infrastructure, testing or integration engineers. Their development skill rating is available for every participant. This will be further discussed in the quantitative study, since this was evaluated using a 5-level Likert scale. 3 participants are avid developers while others have shown an interest in development but are not focused on development.

4 out of 5 participants showed a considerable understanding on how the distribution methods are categorised and how these could be used in their advantage for the development of the simple cross-device application. This includes, but is not limited to the asking of relevant questions and demonstrating few hesitations during the cross-device development. Most of the participants needed some time to adjust to the way of thinking for cross-device development, using the distribution methods, but once the switch had been made, the participants found it quite easy to very easy and fast to develop cross-device applications using this framework. 4 out of 5 implementations were technically very sound, while 1 implementation did finally work, but the implementation did not display that the concepts were understood by the participant, this is also the participant with the lowest self-rating on the development skills question, rating themselves a 2 on a 5-level Likert scale.

None of the participants had any previous experience with Typescript. All of the participants noted this as an added difficulty in completing this evaluation. In order to reduce the impact of this variable as much as possible, participants were allowed to ask any questions on syntax related to the Typescript technology. All participants used this opportunity to query about the

Typescript syntax.

3 out of 5 participants explained that the usability of the HDUI framework could be better by improving the icons in the HDUI menu. The aforementioned HDUI menu is discussed in Section 4.3. In a future study, a manner should be found to give users who are new to cross-device applications access to the necessary functionalities using a more user-friendly manner than the currently confusing icons for participants. The recurring remark was mostly about the connect button. Participants asked why this button is still shown after a device is already connected to a session. The reason is that devices in an active cross-device session can invite more devices to join their already existing cross-device session by clicking the connect icon again. This re-opens the session and allows new devices to click the connect button at a similar time, to be connected to one another.

One participant remarked that the distribution of hyperlinks and files are user-friendly in the way that a confirmation dialog is asked for security purposes, as well as the fact that hyperlinks distributed in this way are opened in a new tab instead of disconnecting from the cross-device session.

Participants were hesitant about the framework at first. Some participants were afraid the required development would be too difficult. However, at the end of the development, the participants were very positive about the ease of development of the framework as well as its rapidness. It seemed the participants had suddenly understood the way the HDUI framework works after a very short development period. Even IT engineers who are not developers were able to finish the development of the cross-device application and were convinced of the framework.

One participant made a remark that there should be more documentation for this framework. Another participant said they'd like to have a more clear overview of all the available cross-device methods. These remarks are correct and the development of more documentation is part of the future works of this framework. This participant also remarked the real-time aspect of the HDUI framework.

6.4.2 Quantitative Study

Unfortunately it is not possible to make an extended quantitative study with the collected data, or with potentially collected data. This research field is still growing and does not contain enough usable data or frameworks that could be re-used for comparison with this evaluation. This is one of the

reasons why the qualitative study was very detailed and useful in this research, since most of the evaluation data came from the observation of the development using the HDUI framework.

Since we want to evaluate the framework's developer-friendliness and rapidness, participants needed to develop a small cross-device application using the framework. However, it is difficult to make an accurate comparison of development time or ease of development with other existing frameworks, since these frameworks might not contain the same functionalities and abstractions. Furthermore, what is possible in the HDUI framework is potentially not possible in another framework. Secondly, as we have discussed, cross-device frameworks are hardly accessible. While it is true that a few frameworks had available source code, the documentations were too limited to be able to give this to our evaluation participants. A third minor problem would be that it would take a lot of time for the participants to develop the application in two cross-device frameworks and also fill in the surveys in order to get data from two frameworks using the same group. This would have to be done in order to preserve the group's skill set, making for more consistent results. Another thing that could have been done, if there were another framework to compare with, is letting one group of participants test one framework first and the other group test the other framework first, in order to balance the results. With the rise of more and more cross-device interaction frameworks, this could still be done in a future works. Newer frameworks could also use the evaluation results from this framework in order to compare their required development time. The project used for evaluation will be made publicly available as soon as the entire framework is made open-source and available with a minimum of documentation as well.

However, a basic analysis of the results is still done using the collected data.

The means for the individual numerical questions can be found in Table 6.1. The participants are called P1 to P5. The ordinal data are on a 5-level Likert scale. The only exception is the ratio data, development time, which is in the form of minutes. In Table 6.2, yes or no questions were filled in with a total sum of participants that answered yes for the specific evaluation items.

From this data, it is mostly noticeable that participants were convinced of the framework. The ease of development, rapidness and recommendation all have a mean of above 4, on a 5-level Likert scale. One outstanding evaluation item is the framework's rapidness. With a mean of 4.8, participants were very convinced that this framework would allow for a rapid implemen-

Evaluation Item	P1	P2	P3	P4	P5	Mean
Development Skills Self-Rating	5	3	2	5	4	3.8
Ease of Development	5	5	4	4	4	4.4
Rapidness Framework	5	5	4	5	5	4.8
Recommend to Developers	5	4	4	N/A	4	4.25
Development Time (minutes)	7.67	11.83	19.17	5.33	11.17	11.03

Table 6.1: Results data including mean values.

tation of cross-device applications. It is also noticeable that there is a clear correlation between the development skills self-rating and the development time. The lower the self-rating, the longer it takes to develop the example cross-device application. The development times range from 5.33 minutes to 19.17 minutes. This is a pretty wide range, however, it makes sense considering the variety of development skills and backgrounds of the participants. There is an average development time of 11.03 minutes per participant for the development of an application, while the participants need to understand the concepts of cross-device applications and the introduced concepts of distribution methods. The participants also need to switch to a cross-device mindset. Taking this in mind, these results seem very promising.

Evaluation Item	P1	P2	P3	P4	P5	Total
Experience with Javascript	1	1	1	1	1	5
Experience with Typescript	0	0	0	0	0	0
Experience with Ionic	1	0	0	0	0	1

Table 6.2: Results of yes no questions including total participants that answered yes. Yes is illustrated as 1, while no is illustrated as a 0-value in the table.

All of the participants had previous experiences with Javascript. However, none of the participants had previous experiences with Typescript, which made the development task more difficult. Syntax-related questions were allowed, but did cost some time for these participant that was also counted into the aforementioned development time. One participant had some minor experiences with the Ionic framework.

It is interesting to notice that this also shows that the initial learning curve for hybrid frameworks can be higher than other technologies. The partici-

pants all had web development knowledge, but still none of them had any experiences with Typescript or had any knowledge about it. Since the HDUI framework is a hybrid framework using Typescript, there is a higher initial learning curve. However, the developers would be able to save time in the long-term due to the ease of development and rapidness of the HDUI framework, as evaluated by the participants.

6.5 Conclusion

Since other cross-device frameworks have not held similar evaluations comparisons to other existing cross-device frameworks cannot be made for this evaluation. Since most cross-device frameworks are hardly accessible or lack clear documentation, we have not been able to evaluate other frameworks as part of this study. Participants consistently said that Typescript was a difficulty for this evaluation, but the HDUI framework itself was very easy and clear to use. All participants were positive about the rapidness of the HDUI framework. Some participants had issues understanding the cross-device development concepts at first, but after developing the simple cross-device application, a shift in mindset was noticed. Most participants understood the framework better after having used it in the evaluation.

This is a positive outlook for the HDUI framework, since all participants had positive remarks about the framework concerning its rapidness and ease of development.

Some items could be improved in the HDUI framework such as the HDUI menu and the documentation of the framework.

Participants took an average time of 11.03 minutes to implement the cross-device functionality into the application. The minimum time a participant took to implement the cross-device functionality was 5.33 minutes, while the maximum time used by a participant for the implementation was 19.17 minutes. Given the variety of participants with different development skill sets and the initial learning curve, including the switch of mindset to a cross-device development mindset, this seems positive. Development of a similar application without the use of a framework by participants with the same skill sets would take a lot longer, since technologies would have to be found and integrated, protocols would have to be defined for communication between the devices and sessions would have to be managed. A system for exchange of variables and distribution of general elements would have to be built. The

HDUI framework automates all of these tasks and proves its potential in the fields of rapidness and ease of development.

7

Conclusion

There is an abundance of technology that can be used for the development of cross-device applications. Cross-device applications could be developed with native technologies, web technologies, hybrid technologies or others. However, to our knowledge, hybrid technologies have not been researched in the context of the cross-device applications research field.

In this research we have presented the HDUI framework, a framework based on the hybrid Ionic framework, which enables the rapid prototyping and development of cross-device applications. Not only did we develop a framework easily accessible by developers, discover the possibilities of hybrid frameworks for cross-device applications, but we also made rapid development of cross-device applications a possibility, without lacking some of the important cross-device functionalities. The HDUI framework makes development of cross-device applications rapid while still giving enough control over the cross-device aspects of the framework to the developers. We have analysed and demonstrated how such a rapid framework can be developed with the support of our defined *distribution methods*, while also defining how cross-device interactions can be developed using just three different distribution methods, which enable the developers to rapidly understand the possibilities of the framework without having to learn, find or understand too many

methods. Furthermore, we have focused on the ease of use for a developer such as automating as many parts of the framework as possible, including the user interface. This allows us to provide the users a streamlined user experience throughout the cross-device applications developed using the HDUI framework.

7.1 Advantages of the Use of a Hybrid Framework

Some advantages linked to the use of a hybrid framework for the development of a cross-device application framework became clear during development and testing of the HDUI framework. First of all, it is a hybrid framework and one codebase is all that is needed for multiple environments. In the context of cross-device applications this is highly interesting, since that would reduce the work required from developers, while still supporting the different environments. Some other forms of solutions such as web applications have a similar advantage, however, these do not give the developers the ability to fully make use of the device's capabilities. For example, it is not possible to query the device's contacts list using HTML5 [27], while this is possible in hybrid frameworks such as Ionic[1]. The Ionic framework is based on Cordova, which runs a container providing the developer the necessary access to the device's capabilities.

One of the biggest advantages are the required tools. This is not the case for every hybrid framework, but for web-based hybrid frameworks, it usually is. This advantage is discussed in the context of the Ionic framework, as it was used for this cross-device research. Developers can use their IDE of choice with Ionic, as Ionic is based on existing programming languages (AngularJS, Typescript and Cordova). This means that any Javascript IDE would work fine for the development of Ionic applications and thus applications developed using the HDUI framework as well. Furthermore, no extra debugging tools are required. Ionic allows developers to debug their applications using the developer tools of web browsers. Developers can keep using the tools they are familiar with, effectively reducing the learning curve related to the use of the framework. Beyond that, Ionic also provides a *debugger* keyword which acts a breakpoint. This is very important for the creation of cross-device frameworks, since the purpose is to make it easier and rapid to develop

[1] `https://ionicframework.com/docs/native/contacts/`

cross-device applications. Having to develop extra tools just for the framework takes more time for both the developer of the tools and the developers using the tools. In this aspect, re-using as much as possible of the existing development tools is certainly a positive advantage from the use of a web-based hybrid framework. This does not only hold true for development tools, but also for testing tools. Since the code of the Ionic application is mostly Typescript, Typescript unit testing and other testing tools can be used for creating automated tests, without requiring any extra tools.

An advantage specific to the Ionic framework is the platform continuity as previously discussed in Section 3.2. Not only will the hybrid framework allow the application to run on multiple environments, it will also adjust the developed applications to the look and feel of native applications made for that specific environment. Ionic enables developers to debug and test the lay-out for multiple platforms in any environment. By adding the *?ionicPlatform=[ios/android/others]* parameter at the end of the URL, Ionic changes the look and feel according to the chosen environment.

The Ionic framework also allows developers to easily package and distribute their applications for multiple platforms. With a build file set up, such as the default build file in the HDUI framework, the application will automatically be built for the different environments.

While a hybrid framework comes with many advantages, it does also have disadvantages such as a steeper learning curve for less experienced developers. However, once that obstacle has been crossed, the advantages behind a hybrid framework for cross-device development are clear. The simplicity behind the HDUI framework still makes it an excellent framework even for less experienced developers.

Hybrid frameworks come with big advantages for the cross-device interaction field and should at least be considered by cross-device developers.

7.2 Contributions

First of all, a classification of currently existing cross-device frameworks, applications or tools, by technology is proposed and illustrated in Table 2.1. This overview can give more insights about the available technologies and which are typically used the most in the cross-device field. Furthermore, it becomes clear that some technologies, such as hybrid technologies have not been researched for the cross-device interactions field.

Since a hybrid technology had to be chosen for the development of the HDUI framework, the existing popular hybrid frameworks had to be analysed. This comparison of popular hybrid frameworks is presented in Section 3.3.

We have explored and discovered the advantages of hybrid frameworks for cross-device applications. Furthermore, distribution methods were defined. With the help of these clearly defined distribution methods, it can be defined which parts of the cross-device functionalities can be automated, in an attempt to automate as much as possible within the HDUI framework. This does not only include automation of the logic, but client side lay-out and usability as well.

As a result of these distribution methods and the possible actions within a cross-device framework are clear to implement and easy to understand.

As a result, developers can create cross-device applications in a matter of minutes, since all that is required for universal HDUI applications, is to fill in the settings file of the application. This allows users to automatically create or connect to existing cross-device sessions and distribute their chosen elements across the devices.

Next to distribution methods, we have also researched, implemented and explained how a protocol-independent framework can be achieved.

With the discovery that current cross-device frameworks are too inaccessible for developers, we hope to solve this problem by making the HDUI framework publicly available with a minimum of documentation.

An evaluation was also held with 5 participants with varying degrees of development skills, in order to assess whether the framework would be suitable for a rapid development of cross-device applications with a considerable ease of development. The results from this evaluation came back with interesting remarks, such as the usability of the icons and the documentation of the framework that could be improved. Other than that, all participants agreed that the HDUI framework positively affects development speed for cross-device development. Participants were surprised by the simplicity of the framework for an otherwise complex development area, cross-device development.

7.3 Future Works

While the current version of the HDUI framework has many features to offer both developers and users, improvements can always be made. Some potential features of the framework were out-of-scope of this research and thus were not researched or implemented.

First of all, the framework is very functional, but the notions of both privacy and security within a cross-device framework need to be analysed in a further study. While the current implementation of the Socket.IO protocol does not make use of encryption techniques, messages could potentially be intercepted. This is especially true for the intermediate NodeJSRouter server. Since the HDUI framework is a protocol-independent framework, a solution could easily be implemented as part of a protocol. It would be interesting to have a future study about the privacy and security of the framework.

A limitation of the Socket.IO protocol is that the connection is disrupted every time the user browses to another page. A solution for this could be the loading of pages in a dynamic manner, without refreshing the entire page, or a manner to reconnect devices as soon as an active user browses to another page. Note that this limitation does not apply to any other protocol that could be rapidly plugged in to the framework.

While the HDUI framework is protocol-independent, the Ionic framework does not yet have the capability of handling bidirectional Bluetooth connections between two devices running the Ionic application. Due to this limitation, a Bluetooth example protocol was not implemented as part of the framework.

Another contribution from the HDUI framework is the redundancy-handling system for client-server architectures.

While it is possible for users to selectionally distribute elements to specific devices, it is currently not possible for a developer to do so using the UI state distribution method. While this could easily be implemented in the future, this was not implemented along due to time constraints.

Requirement R7. extensibility was put on hold for future works, since the focus of the research was shifted to more cross-device oriented concepts and functionalities. The development of an extensibility system would be useful for the developers to actually use the framework, but would not be an enormous contribution to the cross-device research field, since plug-in systems can be found in many different research fields. Instead, more focus was

put on developing a framework that is easy to use for developers, where as much as possible is automated and handled by the framework itself, so the responsibilities are relieved from the developers' shoulders. This would make the development of cross-device applications easier for developers and thus potentially catch the developers' interest in cross-device development.

The current implementation of the HDUI framework's universal template currently uses cross-origin frames. This is not supported by most web browsers due to security reasons. This was previously discussed by Nebeling et al. [19] and these authors proposed a solution involving a proxy. An extra private page could be added into the HDUI framework which would act as a proxy server, receiving and handling requests. This would make any integrations of this page into iframes come from the same origin and thus solve the cross-origin problem. For the moment being, the security in the Chrome browser can be disabled. The idea and solution already exists, but the implementation of it into the HDUI framework is still a future works.

Since the focus was set onto the development of a rapid, hybrid cross-device applications framework, less attention was spent to the actual method of pairing devices. More methods exist to pair devices into the same cross-device session, for example, gestures. Further research should be done as to the advantages of these different methods and their user-friendliness and impacts on privacy.

7.4 Conclusion

We have researched, implemented and presented a complete and functional hybrid cross-device application framework called the HDUI framework. In order to develop this framework, a multitude of concepts such as cross-device fundamentals, hybrid technologies, protocol-independence had to be researched and new concepts were introduced such as distribution methods and a new classification of currently existing cross-device frameworks.

Appendix A: Example HDUI Configuration File

The configuration file of the HDUI framework is shown in Listing A.1.

```
export default class HDUIConfig
{

        /* Protocol Connection Setttings*/
        // The protocol to be used, with the same suffix as its
        ↪  class names
        public chosenProtocol = "HTTP";

        // Polls the server to make sure it is still there
        // If there is no reply, a new server is assigned in
        ↪  the cross-device session
        // Use this in case your chosen protocol does not
        ↪  notify about devices who got disconnected
        // due to connection errors or a crash
        public useKeepAlive = true;
        public keepAliveIntervalMs = 2000;
```

```
14          // The maximum time the keep alive message can be
            ↪  delayed without assigning a new server
15          public maxKeepAliveDelay = 500;
16          // For the initial connection setup, the keepalive
            ↪  timeout will be multiplied by this number
17          // in order to avoid issues during connection setup
            ↪  with the keepalive.
18          // The keepalive will be reset to the original value
            ↪  after the first keepalive message is broadcasted.
19          public keepAliveInitialConnectionMultiply = 5;
20
21          // Config for HTTP Protocol, node server IP
22          public httpServerAddress : String =
            ↪  "http://localhost:3000";
23
24          // Connection timeout in ms
25          public connectionTimeout : number = 20000;
26
27          // Allow mixed content (html) to be distributed by
            ↪  users
28          public allowHTMLDistribution : boolean = true;
29
30          // If set to true, users will be requested a device
            ↪  name
31          public requestDeviceName : boolean = true;
32
33          // Keeps the device name in cookies so it is not
            ↪  requested in future requests
34          public keepDeviceNameInCookies : boolean = true;
35
36          // If set to true, users can transfer files to other
            ↪  devices
37          // Devices are prompt with a confirmation before
            ↪  downloading starts
38          public enableFileTransfers : boolean = true;
39
40          // Make sure these are unique, for the purpose of
            ↪  page-specific DUI
41          public static pageIdentifiers = {
42                  hduiCenterPage: "hduicenter"
```

```
43          };
44
45          public appPages : {pageClass, nameString}[] = [
46                  { pageClass:
                    ↪ HDUIConfig.pageIdentifiers.hduiCenterPage,
                    ↪ nameString: "HDUI Center" }
47          ];
48
49  }
```

Listing A.1: The configuration file of the HDUI framework.

B

Appendix B: Implementation of the Socket.IO Protocol

The implementation of the Socket.IO protocol within the HDUI framework is shown in the following Listing B.1.

```
1  import ProtocolHTTPAddress from './ProtocolHTTPAddress';
2  import * as io from "socket.io-client";
3  import ProtocolMessage from "../interfaces/ProtocolMessage";
4  import {GeneralProtocol} from "../general/GeneralProtocol";
5  import SyncDB from "../../lvl1/SyncDB";
6  import {Diagnostic} from "@ionic-native/diagnostic";
7
8  export class ProtocolHTTP extends GeneralProtocol
9  {
10     // Socket IO variables
11     socket: SocketIOClient.Socket;
12
13     private sessionId : String = null;
14     private serverId : ProtocolHTTPAddress = null;
15
```

```
16      // Interface Variables
17      selfAddress : ProtocolHTTPAddress = null;
18
19      public constructor(l1db : SyncDB, diagnostic: Diagnostic,
        ↪  pageName: String)
20      {
21          super(l1db, diagnostic, pageName);
22      }
23
24      init()
25      {
26          this.socket = io(this.config.httpServerAddress);
27
28          var thisProtocol = this;
29          var successFunc =  () => {
30              thisProtocol.setAddress(this.socket.id);
31              thisProtocol.connect();
32          };
33          var errorFunc = () => {
34              thisProtocol.connectionErrorCallBack();
35              thisProtocol.disconnect();
36          };
37
38          console.log("Initialising Socket IO Connection");
39
40          this.registerMessageListener("connect", successFunc);
41          this.registerMessageListener("connect_failed",
            ↪  errorFunc);
42          this.registerMessageListener("connect_error",
            ↪  errorFunc);
43          this.registerMessageListener("sessionConnect",
            ↪  this.doSessionConnect);
44      }
45
46      private setAddress(addr : String)
47      {
48          this.selfAddress = new ProtocolHTTPAddress(addr);
49      }
50
51      registerMessageListener(tag, listenerFunc: (msg?:
        ↪  ProtocolMessage) => any): void {
```

```
52          this.socket.on(tag, listenerFunc);
53      }
54
55      getSelfAddr(): ProtocolHTTPAddress {
56          return this.selfAddress;
57      }
58
59      getServerAddr(): ProtocolHTTPAddress {
60          return this.serverId;
61      }
62
63      doSessionConnect = (sessionDetails) =>
64      {
65          // Extract details given by Socket IO
66          this.sessionId = sessionDetails.sessionId;
67          var serverId = sessionDetails.serverId;
68
69          // Set the server's ID
70          this.serverId = new ProtocolHTTPAddress(serverId);
71
72          // If this machine is the server, start up the server.
73          this.doInitClientServer();
74      }
75
76      disconnect_protocol()
77      {
78          this.socket.disconnect()
79      }
80
81      // Broadcast message
82      sendMessage(msg: ProtocolMessage)
83      {
84          this.socket.emit('message', msg);
85      }
86
87      public getSessionId()
88      {
89          return this.sessionId;
90      }
91  }
```

Listing B.1: The implementation of the Socket.IO protocol in the HDUI framework.

C

Appendix C: Implementation of the NodeJS Server for the Socket.IO Protocol

The implementation of the intermediary NodeJS server for the Socket.IO protocol within the HDUI framework is shown in the following Listing C.1. This intermediary server is required as client web devices are not able to start their own server for security purposes.

```
1  import HDUISessionConnector from './HDUISessionConnector';
2  import ProtocolMessage from "./ProtocolMessage";
3
4  export default class NodeSessionConnector extends
   ↪  HDUISessionConnector
5  {
6          protected io = null;
7
8          constructor(io)
9          {
10                 super();
```

```
11                  this.io = io;
12                  this.addEventListeners();
13          }
14
15          public sendToClient(tag, id, obj)
16          {
17                  try {
18                          this.io.sockets.connected[id].emit(tag,
                            ↪ obj);
19                  } catch (e) {
20                          console.log("WARN: client
                            ↪ disconnected.");
21                  }
22          }
23
24          public addEventListeners()
25          {
26                  var thisClass = this;
27                  this.io.on('connection', function(socket){
28                          socket.on('message', function(msg :
                            ↪ ProtocolMessage) {
29                                  switch(msg.tag) {
30                                          case "poolConnect":
31                                                  thisClass.onPoolConnect(socket.id,
                                                    ↪ msg.message);
32                                                  break;
33                                          case
                                                    ↪ "sessionBroadcast":
34                                                  thisClass.onSessionBroadcast(msg);
35                                                  break;
36                                          default:
37                                                  thisClass.onMessageTo(socket.id,
                                                    ↪ msg);
38                                  }
39                          });
40                  });
41          }
42  }
```

Listing C.1: The implementation of the NodeJS intermediary server for the Socket.IO protocol in the HDUI framework.

D

Appendix D: Evaluation Instructions for Participants

This appendix is meant for participants of the developer evaluation of the cross-device application framework, the HDUI framework.

First of all, thank you for your participation within the evaluation of the HDUI framework.

D.1 What are Cross-Device Applications?

Cross-device applications are applications that make use of an array of devices at the same time. These could be devices such as mobile phones, desktop computers, TVs or other smart furniture such as smart fridges.

A simple example could be an image gallery application where one device could act as a remote and the other device could act as the screen, actually showing the chosen images.

There many other possible applications in other fields such as education, museums and gaming.

D.2 About the HDUI Framework

The HDUI framework stands for the *Hybrid Distributed User Interfaces Framework*. It is a cross-device application framework which facilitates the development for interaction between multiple devices. To our knowledge, it is the first cross-device framework based on a hybrid framework, the Ionic framework. Aspects such as communication, connection and cross-device sessions management are all handled by the framework. Developers do not need to worry about these aspects and can use high-level abstracted methods to easily control the cross-device interactions between the devices without requiring knowledge of the inner-workings of the framework.

D.3 Distribution Methods

The framework is based on the newly introduced concept of distribution methods. There are three distribution methods proposed.

- General Distribution - this lets the user distribute basic elements on the screen such as text, images or hyperlinks. **This method is fully automated by the framework**.
- UI Components distribution - this lets the user distribute not only basic elements, but also components of the page. An example of this could be HTML elements. **This method is fully automated by the framework**.
- UI state distribution - The synchronisation of variables or the triggering of remote functions. **This distribution method is used by developers** to communicate necessary components between devices.

D.4 Demonstration: the HDUI Menu

The evaluator will now show a quick demonstration of the possibilities of the HDUI framework and its menu.

D.5 Available Methods

Since two distribution methods are fully automated, developers that wish to control the cross-device application flow only need to know how to use one distribution method. The UI state distribution. In this method, variables can be synchronised across devices and functions can be remotely triggered.

D.5.1 Remote Function Triggers

For triggering a remote function across the devices, the following code can be added inside an HDUIController (a page class). This can be seen in the following Listing D.1.

```
// The method provided by the HDUI Framework
HDUIController.registerTrigger(<trigger name>, <trigger
↪  function>);

// An example use of this method
// (!) To be used within ngAfterViewInit() method provided
setupTriggers = () =>
{
        // Register triggers for this page
        var aTriggerFunc = () => {
                // Remote function triggered, do something
                ↪  here
        };

        (this as HDUIController).registerTrigger('btnAction',
        ↪  aTriggerFunc);
}
```

Listing D.1: Registering a trigger using the HDUI framework

The final step is to allow these functions to be triggered, as shown in the following Listing D.2.

```
// The method provided by the HDUI framework
HDUIController.runTrigger(<trigger name>);

// In practice
// (!) To be used within ngAfterViewInit() method provided
```

```
(this as HDUIController).runTrigger('btnAction');

// Or as a button on the HTML of a page
<button (click)="runTrigger('btnAction')">Run Trigger</button>
```

Listing D.2: Triggering a remote function using the HDUI framework.

D.5.2 Synchronised Variables

Synchronised variables are similar to remote function calls, but allow a parameter to be sent as well. Do note that objects cannot be sent in plain format. In case objects have to be sent instead of simple variables, these need to be converted in JSON format. However, for this evaluation, no objects need to be transmitted.

The code for synchronised variables is explained below, in Listing D.3.

```
// Listener (receiver) method
// To be used within ngAfterViewInit() method provided
HDUIController.registerVarTrigger(<variable name>, <trigger
 ↪  function>);

// Distributer (sender) method
// Distributes and runs the trigger functions across the
 ↪  devices
HDUIController.distributeVariable(<variable name>, <value>);
```

Listing D.3: Triggering a remote function using the HDUI framework.

D.6 Pre-Development Survey

Please copy the following text, fill in and send your answers before starting this evaluation.

> What is your function in IT? Developer / Infrastructure / Other, please state:
>
> On a scale of 1 to 5, how do you rate your development skills? Inexperienced [1-2-3-4-5] Very Experienced

Do you have experience with Javascript development? Yes / No

Do you have experience with Typescript development? Yes / No

Do you have experience with the Ionic framework? Yes / No

D.7 Assignment and Goals

One goal of the HDUI framework is to be easy to use for developers and be a rapid prototyping or development framework. In this evaluation, we want to test the time required for developers to implement a simple cross-device application using the rapid HDUI framework.

Before beginning, the evaluator will explain the basics of an Ionic application, if required. The evaluator will also demonstrate how a simple value can be transferred across devices.

You are supplied with a version of the HDUI framework with the required templates. Your goal is to make the functionality behind the application. There are two pages, a gallery page and a remote control page. The remote control page should change the image shown on the gallery page. This can be done by using synchronised variables or remote triggers (multiple implementations are possible, there is no single correct solution). A demonstration of the finished application will be shown by the evaluator.

If you have any questions about this evaluation or framework, please ask them now. Tell your evaluator once you are ready to start the development of the simple cross-device application. During the cross-device development, no questions may be asked, in order to provide reliable statistics for the evaluation.

The tasks required to develop the cross-device parts of the application are described below.

D.8 Post-Development Survey

Please copy the following text, fill in and send your answers before starting this evaluation.

Were you able to create the requested cross-device application? Yes / No

How do you rate the ease of cross-device development using the HDUI framework? Difficult [1-2-3-4-5] Easy

Rate the speed of development using this framework. Very Slow [1-2-3-4-5] Very Rapid

Would you recommend this framework for cross-device development? Not recommended [1-2-3-4-5] Very Recommended

Which aspect of the framework do you think could still be improved?

Did you have any specific difficulties developing this simple cross-device application using the HDUI framework? Yes: ..... / No

Do you have any extra remarks?

D.9 Thank You!

Thank you for taking part of the HDUI framework's evaluation. Your participation is truly important for this research and is very appreciated.

E

Appendix E: Submitted Evaluations

E.1 Evaluation #1

E.1.1 Pre-Development Survey

What is your function in IT?	Developer
On a scale of 1 to 5, how do you rate your development skills?	5 (very experienced)
Do you have experience with Javascript development?	Yes
Do you have experience with Typescript development?	No
Do you have experience with the Ionic framework?	Yes

E.1.2 Development Data

Development Time*	7 minutes and 40 seconds
Solution correct?	Yes
Solution displays understanding of HDUI framework?	Yes

* Time of development until the application is fully functional.

E.1.3 Post-Development Survey

Were you able to create the requested cross-device application?	Yes
How do you rate the ease of cross-device development using the HDUI framework?	5 (easy)
Rate the speed of development using this framework.	5 Very Rapid
Would you recommend this framework for cross-device development?	Yes, because it is easy to understand and development is very easy also.
Which aspect of the framework do you think could still be improved?	I don't have anything yet in mind that could be improved, since I was just introduced to the cross-device topic.
Did you have any specific difficulties developing this simple cross-device application using the HDUI framework?	No, it was easy to implement. The only difficulties I had are because of my lack of experience in Typescript. However, the evaluator helped me solve the Typescript-related errors. (Since these were syntax errors and are not related to the actual cross-device aspect of the framework.
Do you have any extra remarks?	It is very noticeable that the framework is rapid and easy to use. But I can see that the framework could be used for a lot of different applications due to its broadness, since it eases the part of communication between devices.

E.2 Evaluation #2

E.2.1 Pre-Development Survey

What is your function in IT?	Infrastructure, Testing and Integration Engineer
On a scale of 1 to 5, how do you rate your development skills?	3 - basic
Do you have experience with Javascript development?	Yes
Do you have experience with Typescript development?	No
Do you have experience with the Ionic framework?	No

E.2.2 Development Data

Development Time*	11 minutes and 50 seconds
Solution correct?	Yes
Solution displays understanding of HDUI framework?	Yes

* Time of development until the application is fully functional.

E.2.3 Post-Development Survey

Were you able to create the requested cross-device application?	Yes
How do you rate the ease of cross-device development using the HDUI framework?	5 (easy)
Rate the speed of development using this framework.	5 Very Agree
Would you recommend this framework for cross-device development?	4 due to lack of expertise in cross-device domain.
Which aspect of the framework do you think could still be improved?	The security because data is currently sent in plaintext. The icons are not easy to understand for end-users and could be confusing.
Did you have any specific difficulties developing this simple cross-device application using the HDUI framework?	Yes, but the difficulties were related to Javascript and Typescript development, not the understanding and implementation of the cross-device concepts.
Do you have any extra remarks?	While the usability could be better for the icons and the security as well, the framework clearly enables a rapid development for developers.

E.3 Evaluation #3

E.3.1 Pre-Development Survey

What is your function in IT?	Technical Consultant, mostly Infrastructure Engineer
On a scale of 1 to 5, how do you rate your development skills?	2
Do you have experience with Javascript development?	Yes
Do you have experience with Typescript development?	No
Do you have experience with the Ionic framework?	No

E.3.2 Development Data

Development Time*	19 minutes and 10 seconds
Solution correct?	Yes
Solution displays understanding of HDUI framework?	Half. Difficult grasping the concept of function callbacks makes it difficult to understand and use the UI state concepts. With perseverance and trial, the requested goal was achieved.

* Time of development until the application is fully functional.

E.3.3 Post-Development Survey

Were you able to create the requested cross-device application?	Yes, with difficulties, see above.
How do you rate the ease of cross-device development using the HDUI framework?	4 (quite easy) it seems easy to develop since it requires very few functions and devices can connect without any extra code.
Rate the speed of development using this framework.	4 Agree that it is very quick, but as an infrastructure engineer, the programming concepts were hard to grasp.
Would you recommend this framework for cross-device development?	4 yes, for developers, but the concepts are more difficult to grasp for an infrastructure engineer.
Which aspect of the framework do you think could still be improved?	The icons were unclear and hard to understand without context.
Did you have any specific difficulties developing this simple cross-device application using the HDUI framework?	Yes, see above.
Do you have any extra remarks?	Remarks can be found as answers to previous questions.

E.4 Evaluation #4

E.4.1 Pre-Development Survey

What is your function in IT?	Student, Developer
On a scale of 1 to 5, how do you rate your development skills?	5
Do you have experience with Javascript development?	Yes
Do you have experience with Typescript development?	No
Do you have experience with the Ionic framework?	No

E.4.2 Development Data

Development Time*	5 minutes and 20 seconds
Solution correct?	Yes
Solution displays understanding of HDUI framework?	Yes, while the concepts were hard to grasp at first, after the development they became very clear.

* Time of development until the application is fully functional.

E.4.3 Post-Development Survey

Were you able to create the requested cross-device application?	Yes
How do you rate the ease of cross-device development using the HDUI framework?	4 (quite easy) it is easy once you understand the syntax and the different methods the framework provides.
Rate the speed of development using this framework.	5 Very Rapid, once the syntax of the couple different methods is clear, it is a fast framework.
Would you recommend this framework for cross-device development?	Cannot say due to lack of experience in this research field.
Which aspect of the framework do you think could still be improved?	The usability of the icons, especially the reopen session icon.
Did you have any specific difficulties developing this simple cross-device application using the HDUI framework?	No.
Do you have any extra remarks?	Find a solution for the icons for a better usability.

E.5 Evaluation #5

E.5.1 Pre-Development Survey

What is your function in IT?	Infrastructure. Development is a hobby.
On a scale of 1 to 5, how do you rate your development skills?	4
Do you have experience with Javascript development?	Yes
Do you have experience with Typescript development?	No
Do you have experience with the Ionic framework?	No

E.5.2 Development Data

Development Time*	11 minutes and 10 seconds
Solution correct?	Yes
Solution displays understanding of HDUI framework?	Yes

* Time of development until the application is fully functional.

E.5.3 Post-Development Survey

Were you able to create the requested cross-device application?	Yes
How do you rate the ease of cross-device development using the HDUI framework?	4 (quite easy) the cross-device application concepts can be hard to understand, development is quite easy and relatively fast.
Rate the speed of development using this framework.	5 Very rapid.
Would you recommend this framework for cross-device development?	4 it seems interesting for real-time cross-device applications.
Which aspect of the framework do you think could still be improved?	More documentation should exist about the framework so the available methods are clear.
Did you have any specific difficulties developing this simple cross-device application using the HDUI framework?	I'm not familiar to Typescript syntax.
Do you have any extra remarks?	More documentation is necessary.

Bibliography

[1] Ville Ahti, Sami Hyrynsalmi, and Olli Nevalainen. An Evaluation Framework for Cross-Platform Mobile App Development Tools: A Case Analysis of Adobe Phonegap Framework. In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*, pages 41–48, Palerno, Italy, June 2016.

[2] Sriram Karthik Badam and Niklas Elmqvist. Polychrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, pages 109–118, Dresden, Germany, November 2014.

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

[4] Jonathan Bech Bunde-Pedersen. *Distributed Interaction for Activity-Based Computing*. PhD thesis, Aarhus Universitetsforlag, 2009.

[5] Pei-Yu Peggy Chi and Yang Li. Weave: Scripting cross-device wearable interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3923–3932, New York, USA, April 2015.

[6] Pei-Yu Peggy Chi, Yang Li, and Björn Hartmann. Enhancing Cross-Device Interaction Scripting with Interactive Illustrations. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5482–5493, California, USA, May 2016.

[7] Katie GV? To do. Platform continuity | the official ionic blog. `http://blog.ionic.io/platform-continuity/`.

[8] Ericsson. Ericsson mobility report, June 2015.

[9] Luca Frosini, Marco Manca, and Fabio Paternò. A Framework for the Development of Distributed Interactive Applications. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 249–254, London, UK, June 2013.

[10] Jose A Gallud, Ricardo Tesoriero, Jean Vanderdonckt, María Lozano, Victor Penichet, and Federico Botella. Distributed User Interfaces. In *Extended Abstracts on Human Factors in Computing Systems*, pages 2429–2432, Vancouver, Canada, May 2011.

[11] Maria Husmann, Nina Heyder, and Moira C Norrie. Is A Framework Enough?: Cross-Device Testing and Debugging. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 251–262, Brussels, Belgium, June 2016.

[12] Maria Husmann, Michael Nebeling, and Moira C Norrie. MultiMasher: a Visual Tool for Multi-Device Mashups. In *International Conference on Web Engineering*, pages 27–38, Aalborg, Denmark, July 2013.

[13] Maria Husmann and Moira C Norrie. XD-MVC: Support for Cross-Device Development. In *1st Intl. Workshop on Interacting with Multi-Device Ecologies in the Wild (Cross-Surface 2015)*, Madeira, Portugal, November 2015.

[14] Maria Husmann, Nicola Marcacci Rossi, and Moira C Norrie. Usage Analysis of Cross-Device Web Applications. In *Proceedings of the 5th ACM International Symposium on Pervasive Displays*, pages 212–219, Oulo, Finland, June 2016.

[15] Minh Q Huynh, Prashant Ghimire, and Donny Truong. Hybrid App Approach: Could It Mark the End of Native App Domination? *Issues in Informing Science and Information Technology*, 14:049–065, April 2017.

[16] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. USIXML: A Language Supporting Multi-Path Development of User Interfaces. In *International Workshop on Design, Specification, and Verification of Interactive Systems*, pages 200–220, Hamburg, Germany, July 2004.

[17] Jérémie Melchior, Jean Vanderdonckt, and Peter Van Roy. A Model-Based Approach for Distributed User Interfaces. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 11–20, Pisa, Italy, June 2011.

[18] Michael Nebeling. XDBrowser 2.0: Semi-Automatic Generation of Cross-Device Interfaces. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 4574–4584, Colorado, USA, May 2017.

[19] Michael Nebeling and Anind K Dey. XDBrowser: User-Defined Cross-Device Web Page Designs. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5494–5505, New York, USA, May 2016.

[20] Michael Nebeling, Maria Husmann, Christoph Zimmerli, Giulio Valente, and Moira C Norrie. XDSession: Integrated Development and Testing of Cross-Device Applications. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 22–27, Duisburg, Germany, June 2015.

[21] Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. Interactive Development of Cross-Device User Interfaces. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2793–2802, Toronto, Canada, May 2014.

[22] Michael Nebeling, Christoph Zimmerli, Maria Husmann, David E Simmen, and Moira C Norrie. Information Concepts for Cross-device Applications. In *DUI@ EICS*, pages 14–17, London, UK, June 2013.

[23] Audrey Sanctorum and Beat Signer. Towards User-defined Cross-Device Interaction. In *International Conference on Web Engineering*, pages 179–187, Brussels, Belgium, October 2016.

[24] Mario Schreiner, Roman Rädle, Hans-Christian Jetter, and Harald Reiterer. Connichiwa: A Framework for Cross-Device Web Applications. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2163–2168, Seoul, Republic of Korea, April 2015.

[25] Lucia Terrenghi, Aaron Quigley, and Alan Dix. A Taxonomy For and Analysis of Multi-Person-Display Ecosystems. *Personal and Ubiquitous Computing*, 13(8):583–589, November 2009.

[26] Ricardo Tesoriero. Distributing User Interfaces. In *Proceedings of the 2014 Workshop on Distributed User Interfaces and Multimodal Interaction*, pages 1–10, Toulouse, France, July 2014.

[27] Richard Tibbett. Pick Contacts Intent. `https://www.w3.org/TR/contacts-api/`.

[28] Jean Vanderdonckt et al. Distributed User Interfaces: How to Distribute User Interface Elements Across Users, Platforms, and Environments. *Proc. of XI Interacción*, 20, September 2010.

[29] Pedro G Villanueva, Ricardo Tesoriero, and Jose A Gallud. Performance Evaluation of Proxywork. In *Proceedings of the 2014 Workshop on Distributed User Interfaces and Multimodal Interaction*, pages 42–45, Toulouse, France, July 2014.

[30] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 635–646, Berlin, Germany, November 2013.

[31] Spyros Xanthopoulos and Stelios Xinogalos. A Comparative Analysis of Cross-Platform Development Approaches for Mobile Applications. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 213–220, New York, USA, September 2013.

[32] Jishuo Yang and Daniel Wigdor. Panelrama: Enabling Easy Specification of Cross-Device Web Applications. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2783–2792, Toronto, Canada, April 2014.