MASTER THESIS

Extending Web Technologies to Enable Ad Hoc Cross-Device Interaction

submitted for acquirement of the academic degree of a Master of Science (M.Sc.) by

Mario Schreiner

at the
University of Konstanz
Department of Computer and Information Science

Reviewers:
Professor Dr. Harald Reiterer
Professor Dr. Marc H. Scholl

December 1, 2015

ABSTRACT

When we look around us today, we see digital devices everywhere: The smartphone in our pocket, tablets and desktop computers on our office table, the laptop in our bag, or the smartwatch on our wrist. And still, while all these devices are interconnected and even share data, for example through cloud services, they are still, in a sense, isolated. Devices today are mostly unaware of all the devices that surround them – and although there is great potential in combining devices to use them in concert, this kind of interaction is rarely seen outside of research labs. Subject of this work is the definition of current obstacles in ad hoc cross-device interaction and to evaluate novel ways of solving them. This is attained through the development of a web-based prototype cross-device framework that includes concepts for a) creating multi-device web applications and b) enabling ad hoc combination of off-the-shelf devices in everyday scenarios. These concepts are evaluated, and the results will aid us in proposing new web standards that can help to proliferate cross-device interaction in everyday life.

ZUSAMMENFASSUNG

Wenn wir uns heutzutage umschauen, sehen wir überall technische Geräte: Das Smartphone in der Hosentasche, Tablets und Computer auf dem Bürotisch, den Laptop im Rucksack oder die Smartwatch am Handgelenk. Und obwohl all diese Geräte vernetzt sind und – zum Beispiel über Clouddienste – Daten teilen können, sind sie in gewissem Sinne abgeschottet. Heutige Geräte sind sich anderer Geräte in der Umgebung selten bewusst – trotz des großen Potentials, das in der Kombination verschiedener Geräte und deren gemeinsamer Nutzung steckt, ist eine solche Interaktion außerhalb von Forschungslaboren selten. Thema dieser Arbeit ist das Herausarbeiten der gegenwärtigen Hindernisse von spontaner Cross-Device-Interaktion und dem Finden neuartiger Lösungen für diese. Dies wird durch die Entwicklung eines web-basierten, prototypischen Cross-Device-Framework erreicht. Das Framework beinhaltet Konzepte a) zur Erstellung von Multi-Device-Webanwendungen und b) um Ad-hoc-Kombination handelsüblicher Geräte in Alltagssituationen zu erlauben. Diese Konzepte werden anschließend evaluiert, um mögliche zukünftige Webstandards zu entwickeln, die helfen können, eine Verbreitung von Cross-Device-Interaktion im Alltag zu erreichen.

PUBLICATIONS

Parts of this research were previously issued in the following publications:

Schreiner, M. (2014). Connichiwa: A framework for local multi-device web applications. Master seminar paper, University of Konstanz.

Schreiner, M. (2015). Connichiwa: A framework for cross-device web applications. Technical report, University of Konstanz.

Schreiner, M., Rädle, R., Jetter, H.-C., and Reiterer, H. (2015a). Connichiwa: A framework for cross-device web applications. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 2163–2168, New York, NY, USA. ACM.

Schreiner, M., Rädle, R., O'Hara, K., and Reiterer, H. (2015b). Deployable cross-device experiences: Proposing additional web standards. Workshop Paper at *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '15 — Workshop "Cross-Surface: Workshop on Interacting with Multi-Device Ecologies in the Wild" (http://cross-surface.io).

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER 1

INTRODUCTION

1.1 Of Lonely Devices

> *"Many scenarios of mobile, pervasive, and ubiquitous computing envision a world rich in interconnectable devices and services, all working together to help us in our everyday lives (...) And yet, in reality it is difficult to achieve the sorts of seamless interoperation among them that is envisioned by these scenarios. How much more difficult will interoperation become when our world is populated not only with more devices and services, but also with more types of devices and services?"* (Edwards et al. 2009)

The proliferation of devices in our everyday life is noticeable everywhere: We are accompanied by smartphones, tablets, smartwatches and laptop computers almost all of the time in our daily life, public displays become more and more common, stores start to offer in-store devices to browse their products, and technological advances such as head-mounted displays and smart home technologies further add to this development. Mark Weiser's vision of ubiquitous computing is a work cited countless times (Weiser 1991, 1993), but also controversial in that it was interpreted in a variety of ways. Most people would probably agree, though, that the growing number of devices that we currently witness is an integral part of this vision. Weiser also predicted that, with the emergence of many devices per person, computers would have to deal with the problem of interconnection, communication and of combining their capabilities to achieve a seamless experience for users and "disappear into the background" (Weiser 1991).

Today, a lot of issues in regard to this intercommunication of devices still remain. Modern devices are highly interconnected using technologies such as Wi-Fi, Bluetooth (BT), or Near Field Communication (NFC). Other technologies, such as the Global Positioning System (GPS), help devices to get aware of their environment, for example to offer nearby bus or metro data or proactive suggestions. And despite these technologies, modern devices are still mostly unaware of their immediate surrounding and in particular of other devices nearby. Users cannot easily utilise their many devices and consequently still work in a highly sequential fashion with them (Jokela et al. 2015). In fact, today's users still struggle with simple multi-device tasks such as transferring data, moving a presentation to a public display, or start an ad hoc multiplayer game. The reasons for these issues are manifold and include differences in architecture, technical capabilities, and operating systems but also missing standardisation, security concerns, and economical reasons. In particular, finding a "common ground" for the heterogenous device landscape that occurs in the wild is both a conceptual and technical challenge.

Companies try to compensate for these issues with the help of cloud services, enabling data synchronisation between a user's devices. Prominent examples of such

services are Apple iCloud[1], Microsoft Office 365[2], Google Docs[3], and Dropbox[4], to name just a few. These services help in coping with data management across multiple personal devices, but enable sequential cross-device use at best: Users can move data from one device to another and pick up where they left off, but they cannot utilise multiple devices at the same time (Hamilton and Wigdor 2014). Maybe even more severe, cloud services are unaware of a device's current surrounding: They rely on shared logins, invitations (e.g., to a shared folder in a cloud storage) and a constant high-bandwidth internet connection. This lack of awareness makes it difficult for users to bring their devices together to engage in multi-surface applications spontaneously.

Recently, technologies based on short-range wireless communication, such as Apple's Continuity[5], have gained popularity. They allow data exchange with nearby devices and are used to transfer the current application state from one device to another (e.g., starting an email on your smartphone and continuing on your desktop computer). Unlike cloud services, these technologies *are* aware of surrounding devices, and hence get a step closer to Weiser's vision. As of now, though, they are tailored towards single-user usage, again relying on shared logins, and aim at a sequential use of devices. Furthermore, these technologies currently do not allow devices to cross the gap between different ecosystems.

This thesis will explore how to enable more advanced cross-device interaction in everyday life than currently possible, in particular with a focus on *ad hoc* cross-device interaction and parallel use of multiple devices. There is great potential in the devices that surround us and using them in concert, such as combination of resources (e.g., computational power or screen real estate), ad hoc collaboration or sharing, gaming, interaction between private and public devices, and many more. To achieve a seamless ad hoc interaction, a lot of conceptual and technical issues must still be solved, though, and this thesis aims at giving a starting point for possible solutions and how cross-device interaction could be achieved in everyday life in the future.

1.2 Ad Hoc Cross-Device Interaction

In the field of cross-device interaction, terms such as pervasive computing, ubiquitous computing, or multi-device use are often used synonymously, and a clear separation of these terms is still lacking in research. To establish a common understanding, this section will make an attempt at defining the most common of these terms. In particular, this section defines cross-device interaction and *ad hoc* cross-device interaction as they are used within this text, and differentiates them from other terms used in research.

[1] Apple iCloud Homepage — https://www.apple.com/icloud/ — Accessed December 1, 2015
[2] Microsoft Office Homepage — http://office.microsoft.com/en-us/products/ — Accessed December 1, 2015
[3] Google Drive — https://drive.google.com/ — Accessed December 1, 2015
[4] Dropbox Homepage — https://www.dropbox.com/ — Accessed December 1, 2015
[5] Apple Continuity Homepage — http://www.apple.com/osx/continuity/ — Accessed December 1, 2015

Ubiquitous Computing is a term coined by Mark Weiser in 1991 and describes a vision of technology becoming so ubiquitous that is not perceived as extraordinary anymore and "disappears into the background" (Weiser 1991). In our understanding, cross-device interaction is part of this vision, as the huge number of devices need to interconnect and work together to achieve a seamless user experience. Nonetheless, cross-device interaction can also occur separated from ubiquitous computing.

Pervasive Computing is used synonymously with the term ubiquitous computing.

Device ecologies describe multi-device scenarios with a focus on the devices instead of the interaction between them. A device ecology is considered a collection of devices that are in physical proximity to each other and aware of the devices around them. In this sense, the basis for cross-device interaction is the creation of an ecology of devices.

Distributed User Interfaces (DUIs) describe scenarios where a user interface (UI) is not presented as a single entity, but distributed over multiple places. This must not necessarily describe a distribution amongst multiple devices, it can also mean an interface distributed, for example, across multiple screens. Oftentimes, though, it is mentioned in relation to distributing a single interface across multiple devices. In these cases, the term describes the user interface of a multi-device application.

Cloud Services first seem largely different from cross-device interaction, but can be easily mistaken. Several common interactions in multi-device scenarios are also enabled by cloud services, such as collaboration of multiple users on the same document. Cloud services differ a) in technical notion, as they necessarily require a remote centralised server that handles communication between devices and b) in concept, as cloud services are not aware of (and do not care about) the location of the participating clients, but rather connect "some clients" that are identified to the server by user credentials. Arguably, cross-device interaction could be understood as an overarching theme that includes cloud services, but we see a clear difference between the two: Cross-device interaction is a *localised* interaction across device, where physical proximity of devices plays a key role.

Cross-Device Interaction therefore describes the interaction between the devices of a device ecology – multiple devices in physical proximity. An interaction occurs when these devices get aware of each other and start to exchange information in a way that benefits the user, e.g. by combining their hardware capabilities or forming a distributed user interface.

Ad Hoc Cross-Device Interaction is a special kind of cross-device interaction where the device ecology is created without setting up specialised hardware, without the user(s) going through a software setup and without a priori knowledge of the nature of the device ecology (such as the type or number of participating devices).

Cross-surface / multi-surface / multi-device interaction are used synonymously with the term cross-device interaction.

1.3 Outline

This thesis explores ad hoc cross-device interaction and why such interaction is not currently possible in everyday life, even though a multitude of devices surrounds us all the time. The thesis examines the theoretical background of such interaction and the work other researchers have done in the past in *Related Work* (Chapter 2, p. 7). In *Connichiwa: An Ad Hoc Cross-Device Framework* (Chapter 3, p. 24), we build upon past research to define the current stopping blocks for ad hoc cross-device interaction. We then derive key requirements for future technologies to eliminate these stopping blocks. We argue why web technologies are promising for advancing cross-device interaction and present a web-based prototypical framework, *Connichiwa*, that is built on these requirements. The chapter describes Connichiwa's concepts and implementation using state-of-the-art technologies. We then present a two-fold evaluation of Connichiwa in *Evaluation* (Chapter 4, p. 47), where we investigate a) Connichiwa's fulfilment of our initial requirements and b) how developers cope with the possibilities provided by the framework. In *Ad Hoc Cross-Device Web Extensions* (Chapter 5, p. 63), we derive design guidelines for future web standards that will allow the creation of multi-device web applications and disseminate ad hoc cross-device interaction into everyday life. In *Conclusion* (Chapter 6, p. 67), we then detail future work on the Connichiwa framework and conclude the thesis. Lastly, thanks are given where thanks are due in *Thanks* (Chapter 7, p. 72).

CHAPTER 2

RELATED WORK

Research in the area of cross-device interaction is manifold and deals with many different facets, including discovering devices, establishing a connection, communicating between devices, or developing novel interaction techniques. Some researchers approach the topic in a more general manner, exploring how users utilise multiple devices or how developers can be supported in creating cross-device experiences. This chapter explores the most relevant work to examine why none of the existing solutions have seen a larger acceptance in both the research community and by everyday users. These insights will be used in Chapter 3 to derive key requirements that are then used as the basis for the Connichiwa framework. Please note that parts of this chapter have already been published in a similar form in the seminar paper that preceded this thesis (Schreiner 2014).

2.1 Multi-Device Usage Patterns

Recently, Jokela et al. investigated how today's users utilise the multitude of devices available to them in everyday activities (Jokela et al. 2015). In a diary study with 14 participants, they looked at how people used devices such as computers, smartphones, tablets, televisions, game devices, cameras, music players, navigation devices and wristwatch computers. After a self-reporting period of one week and an interview, they identified patterns of multi-device usage. They conclude that there is indeed a need for cross-device interaction in today's life that is not satisfied by current software and hardware:

> *"the participants wanted all their devices to seamlessly work with each other. However, in practice, they continuously encountered problems in multi-device use, especially between devices of different ecosystems. (...) Plenty of work still remains to be done to realise the vision of smooth and effortless multi-device computing."* (Jokela et al. 2015)

Furthermore, they identified four main ways people used their devices. Note that this study was directed at single-user settings, the area of multi-user multi-device settings was not studied.

- **Sequential Use:** People changed their devices during tasks, such as searching for a phone number online and then switching to the phone to make the call. 37% of multi-device use cases where sequential.
- **Resource Landing:** In resource landing, the focus of users is on one device while the resources of other devices is borrowed, such as connecting a laptop to a TV to watch movies. 27% of multi-device use cases belonged to resource landing.
- **Related Parallel Use:** People used multiple devices for the same task, such as watching a movie on the TV while retrieving additional information on their smartphone or tablet. 28% of multi-device use cases where related parallel use.

- **Unrelated Parallel Use:** People also used multiple devices for multiple tasks. This case involves a primary foreground task and another background task, such as listening to music with the phone while working on the personal computer. 8% of multi-device use cases where unrelated parallel use.

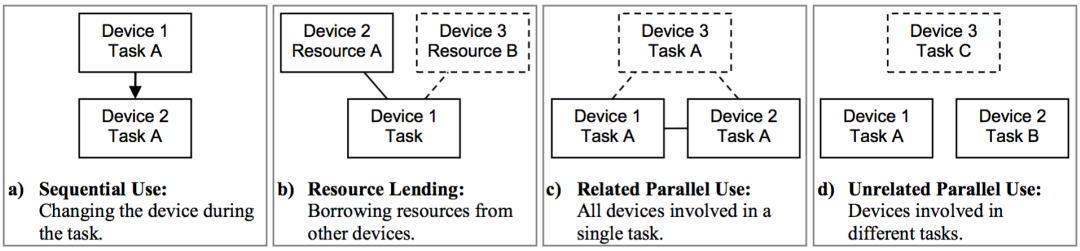

Figure 1: Overview over Multi-Device Usage Patterns as identified by Jokela et al. Source: (Jokela et al. 2015).

Jokela et al. point out that, as of today, most commercial support for multiple devices is aimed towards *sequential use* and *resource landing*. The common case of *related parallel use* is unsupported by modern devices, but users do desire their devices to work together in such a manner. Further, users made heavy use of cloud services, but raised concerns about all their data flowing through the cloud and suggested more direct solutions of content sharing between devices.

2.2 Multi-Device Workflows

Santosa and Wigdor studied 22 professionals and their use of multiple devices in their everyday work life (Santosa and Wigdor 2013). The study was done in situ and using a mixture of technical professionals and novices. They identified four workflows common among most of the participants and identified several problems with current multi-device practices. They describe the following workflow patterns:

- **Producer-Consumer:** A sequential pattern where one device is used to find information and the information is then transferred to another device.
- **Performer-Informer:** A parallel pattern where a device, such as a tablet, is used for reference while the primary work is done on another device.
- **Performer-Informer-Helper:** Similar to the Performer-Informer pattern, but with an additional device, such as a smartphone, as another helper device that is used, for example for calculations or quick look-up.
- **Controller-Viewer/Analyser:** A pattern where different aspects of a single task were executed on multiple devices, such as using a smartphone as a remote control.

Only the Producer-Consumer pattern is sequential while the other patterns are of parallel nature. This might indicate a good support for such patterns in everyday life, contrary to what Jokela et al. found. Nonetheless, Santosa and Wigdor point out that parallel patterns in particular suffered from shortcomings during their observations and they name several problems with them. A key problem was what they called "barriers to parallelism" (Santosa and Wigdor 2013). For example, users noted a lack of support for transfer interactions, such as moving documents or application state between devices instantly. Further, the missing support for distributed UIs was considered a problem. Santosa and Wigdor say that "participants with many devices available have the opportunity for rich parallel usage, yet they remain limited by the lack of cross-device interaction supporting this" (Santosa and Wigdor 2013) and that "parallel patterns in particular would benefit from devices being mutually aware of each other's location, proximity, and orientation" (Santosa and Wigdor 2013).

Santosa and Wigdor also investigated on each device's role in the observed tasks. They conclude that users select devices primarily based on their physical affordance: Tablets are often used for reading because of their size, weight and portability while smartphones are used for quicker, smaller tasks. Still, for heavy work, laptops and desktop computers are the preferred devices, at least in the workplace environment that was the focus of their study.

2.3 Challenges in Distributed UIs

Fisher et al. developed three example applications with distributed user interfaces and investigated them to discover challenges for such applications (Fisher et al. 2014). They conclude seven main challenges for distributed user interfaces. Part of every cross-device application is the design of a distributed user interface, so the challenges of such interfaces are highly relevant for cross-device applications.

- **Consistency:** Maintain consistency between the user interfaces of multiple devices.
- **Synchronisation:** Actions on one device must be reflected on all devices.
- **Heterogenous Hardware:** Fisher et al. emphasise that it is important not to impose hardware restrictions when possible, allowing a variety of mobile and desktop systems to participate.
- **Volatile Device Ecosystem:** Mobile devices in particular are likely to join or leave an application at any time. Applications should be robust against such actions.
- **Limited Resources:** Some resources, in particular files or physical media, are limited and applications should be aware of that and be able to cope with it.

- **Data transfer:** Applications should distribute said limited resources automatically where necessary.
- **Physical Space:** Devices should be autonomous, considering that they may join or leave an application at any time. Therefore, one device should not be dependent on another and the application should be robust against any of the devices leaving.
- **Asymmetric Functionality:** Oftentimes, it makes sense to distribute functionality asymmetrically between devices, so that each device performs different functions. Applications should acknowledge that and distribute functionality according to the current device ecology.

2.4 Detecting and Tracking Devices

To enable cross-device interaction, a basic step is for devices to become aware of each other. Some systems rely on detection only, while others also track devices to enable new kinds of interaction. In this section we will describe related research in these areas.

2.4.1 Adapted Devices

A prominent way of making devices aware of their surrounding is device augmentation or adaption. Such adaptions require either custom-built hardware or attaching additional hardware to off-the-shelf devices. To use custom hardware, specially tailored software must be developed. In turn, such adaptions allow a device to recognise when another device is nearby, moves or leaves.

An early example using an entirely custom piece of hardware is *Smart-Its Friends* (Holmquist et al. 2001), consisting of two boards: One core unit that processes information and one sensor unit that is used to detect other sensor units. Bringing two sensor boards close to each other makes them report to their core units. The units can be included in a device in order to make it a “Smart-It” and therefore make it able to detect other “Smart-Its”.

Some early projects relied on infrared sensors for detection. Cheap and easy to use, infrared sensors are based on the transmission and detection of infrared light, which is invisible to the human eye. Because of the nature of infrared light waves, they require direct line of sight between the sender and the receiver to work (as, for example, with most remote controls today) and also suffer from a rather short range. A very prominent example using infrared sensors are *Siftables* (Merrill et al. 2007). Siftables are small cubical devices (see Figure 2) featuring a color LCD, an accelerometer, and four infrared receivers – one at each side – that allow them to detect other Siftables in very close range (about 1cm).

MagMobile (Huang et al. 2012) uses magnetism as an alternative for device detection. Magnetic fields are produced by most technical devices, in particular modern devices featuring an integrated compass. Bringing two magnetic fields close to each other makes them distort each other. The distortion can be measured and devices can be detected and tracked based on that distortion. While most modern devices have a built-in compass, early experiments of the author of this thesis have shown that the magnetic field is not strong enough to allow an accurate tracking of devices. MagMobile solved this problem by developing a custom, external magnetic sleeve (see Figure 3). Multiple devices using the MagMobile sleeve can then detect each other. An Arduino board[6] is attached to the sleeve and sends measurements to a shared server. Because even small difference in the magnetic distortion can be interpreted, MagMobile can continuously track devices and calculate their relative position. While Huang et al. do not explicitly state the range of the device detection, it can be assumed that the magnetic field is distorted only in close proximity of devices.

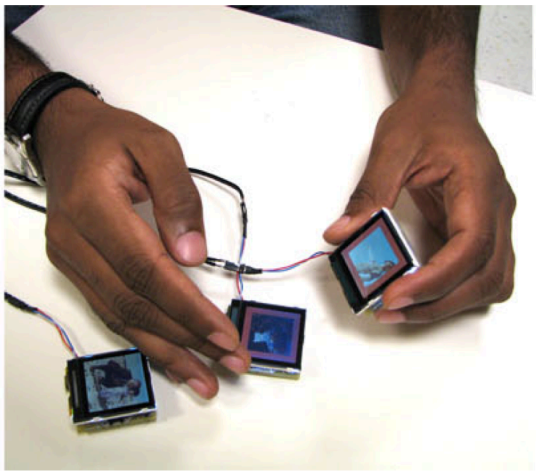

Figure 2: Siftables (Merrill et al. 2007) detect each other via four infrared sensors, one at each side. Source: (Merrill et al. 2007) (parts of the image were cropped).

Figure 3: MagMobile (Huang et al. 2012) uses a custom magnetic sleeve for device detection. Source: (Huang et al. 2012) (parts of the image were cropped).

2.4.2 Digital Image Processing

In recent years, the use of cameras paired with digital image processing has become a prominent way to detect and track people and devices. With the power of modern computers it is possible to live-process images from a video camera in order to detect objects, people and devices. However, accurate image processing algorithms are difficult to develop and still a hot topic in research. Tracking algorithms often have to be adapted to the usage scenario, environment, lighting and the objects that shall be detected. Digital image processing can allow to track objects and devices accurately in real-time while keeping the user mostly unaware of the involved hardware, which is

[6] Arduino Homepage — http://www.arduino.cc — Accessed December 1, 2015

unique to this approach. It is also possible to develop new interaction techniques based on the tracking of hands or other body parts (e.g. Microsoft Kinect[7] or the *Proximity Toolkit* (Ballendat et al. 2010; Marquardt et al. 2011)). While cameras have become much more affordable in recent years, to rely on accurate image processing they still need to be installed and configured and an algorithm has to be developed, which is often a lengthy and difficult process. This is particularly true for complicated systems such as OptiTrack[8].

An often-quoted example of digital image processing is *Phone as a Pixel* (Schwarz et al. 2012). Phone as a Pixel detects device position by encoding a color pattern on each device's screen and picking that pattern up with a camera. Devices must be aligned to the camera properly for the detection to work. Once the device positions are established, a central server tells each device what to display. An example application is the stitching of a giant image across device screens. Detection and tracking of changes in the device positions is limited.

The *Proximity Toolkit* (Ballendat et al. 2010; Marquardt et al. 2011) augments an entire room with cameras. Trackable objects must be augmented using special markers in order to be detectable by the infrared cameras. While the preparation for using the toolkit is extensive and a 3D model of the room has to be predefined, the toolkit allows the tracking of not only digital devices, but also people, gestures, posture, and non-digital objects. This allows for the creation of advanced applications that react to subtle movements of people, to persons entering or leaving the room, or to people redirecting their attention to an object or device.

HuddleLamp (Rädle et al. 2014) mounts a camera into an ordinary desk lamp, which can then be used to track devices on a flat surface such as a desk. The setup is fast and cheap compared to more general-purpose frameworks like the Proximity Toolkit, but in turn the system is more limited. HuddleLamp is able to continuously live-track devices without markers (see Figure 4), but the tracking area is limited by the field-of-sight of the camera, usually about the size of a desk. To identify devices, they show a QR code encoding an ID briefly whenever a new device joins. The device can then receive information about the location and rotation of other devices or messages sent by other devices.

The rising quality of cameras in modern mobile consumer devices allows the usage of the device camera for digital image processing, partly eliminating the need for an external augmentation of the environment. This approach can be problematic due to the fact that a device needs to track other devices while it is being used. Dearman

[7] Kinect for Xbox One Homepage — http://www.xbox.com/en-US/xbox-one/accessories/kinect-for-xbox-one — Accessed December 1, 2015

[8] OptiTrack Homepage — https://www.naturalpoint.com/optitrack/ — Accessed December 1, 2015

et al. developed a method for devices to detect their relative position to other devices based on the back-facing camera image (Dearman et al. 2012). A device can register with a web service and will periodically send a still image of the backside camera to the service. The system assumes that people that want to share information stand in close proximity and in a circle. Based on that assumption, the online service extracts features like feet or legs from the camera images and determines relative device positions from that. The system only features very limited live-tracking and makes large assumptions on the usage scenario of such devices, but in turn does not need any further augmentation of devices or the environment.

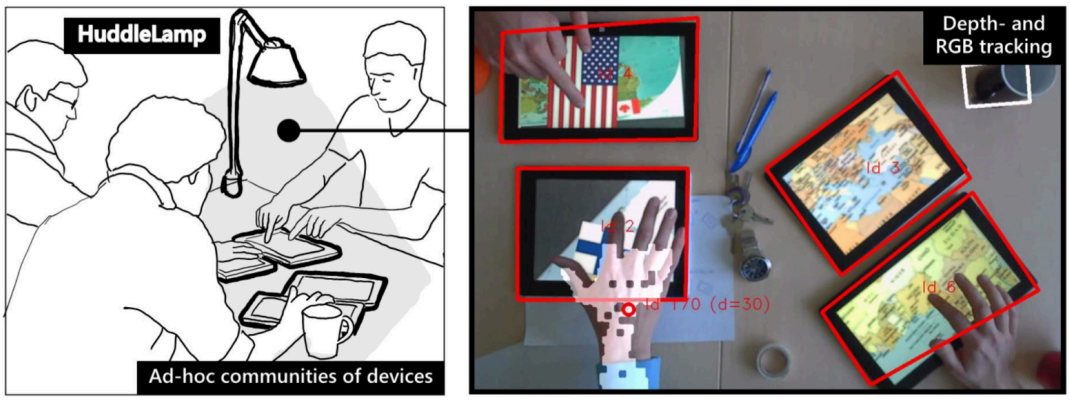

Figure 4: HuddleLamp (Rädle et al. 2014) accurately live-tracks devices on a flat surface using a camera mounted into a desk lamp. Source: (Rädle et al. 2014).

2.4.3 Short-Range Wireless Technologies

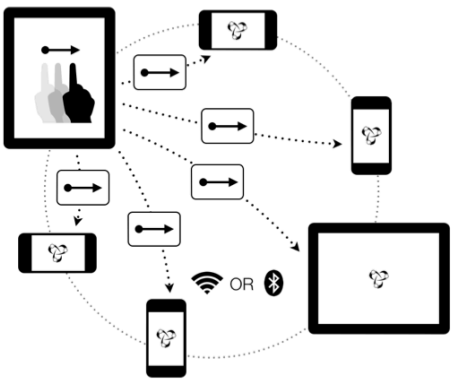

Figure 5: Pinch (Ohta and Tanaka 2012) detects devices ad hoc via Wi-Fi or Bluetooth packet broadcasting. Source: (Ohta and Tanaka 2012).

Short-range wireless communication over radio waves, such as NFC or Bluetooth, have seen a wide distribution and acceptance in modern consumer devices. This availability and the ad hoc nature of such technologies is an advantage over other methods. For example, *Pinch* (Ohta and Tanaka 2012) performs device discovery only via Bluetooth packet broadcasting (see Figure 5). It does so without a-priori agreements of the participating devices and without the need for a common Wi-Fi network. Pinch works with off-the-shelf consumer devices, allowing to detect nearby devices basically everywhere. The range and number of devices is limited, though, and an accurate live-tracking is not possible.

Conductor (Hamilton and Wigdor 2014) uses NFC as a method to make devices join a cross-device application. When two NFC-enabled devices come close to each other, they establish a communication channel and exchange information over NFC that allows the devices to connect to the same remote WebSocket server.

Faragher and Harle investigated on the ability of Bluetooth beacons for tracking devices (Faragher and Harle 2014). They performed an in-depth analysis in a real-world office environment. They concluded that, due to the physical nature of Bluetooth, tracking using Bluetooth is difficult. Nonetheless, they were able to achieve an accuracy of less than 2.6m 95% of the time using 19 beacons in an office floor (about 6-8 beacons at any time).

2.4.4 Acoustic Sensing

SurfaceLink (Goel et al. 2014) proposes an approach on the detection of devices using acoustical sensing. It uses the device's integrated microphone to determine device positions on a flat, rough surface. Multiple devices can be placed on such a surface and movements with the hand or finger on the surface are picked up by the microphones. By communicating with each other, the devices are then able to determine their relative position and the movement of the hand or finger. The exact details of device communication are not detailed by Goel et al. but a mixture of "GPS and Wi-Fi information" (Goel et al. 2014) is used. SurfaceLink only works on a flat, suitable surface but poses a solution that does not rely on adaption of devices or the environment.

2.4.5 Discussion

Off-the-shelf devices offer the possibility to enable cross-device interaction ad hoc, enabling people to join with their own devices in a variety of locations. In turn, they still lack the necessary sensors for an accurate position tracking of devices, a fact that was also concluded by Hamilton and Wigdor in Conductor (Hamilton and Wigdor 2014). With additional augmentations, the sensing capabilities of such devices (or the environment) can be increased, but such setups trade in freedom – they are tied to a location, such as a room or a table, and people cannot participate without a specially prepared device.

2.5 Device Communication

When devices are able to detect each other, they must be able to exchange information to enable cross-device interaction. This can include handshake information, such as network state, form factor, or hardware capabilities, but eventually also information about the content to display, events that occur and interactions by the user. This section will introduce state-of-the-art methods for such a data exchange.

2.5.1 Shared Remote Server

The most prominent way of achieving communication between devices is through a shared remote server that is known beforehand to all participating devices. Setting this kind of communication up is often straightforward and can be adapted to the application needs by adjusting the method of communication (e.g., direct TCP connection, WebSocket, or REST). It requires all participating devices to have access to the shared server – for example through an internet connection – and also a priori knowledge of the server address, type of communication, and communication protocol. It imposes some limitations on the application performance because of the communication delay and bandwidth limitations of network communication. Also, it requires the developer to setup and operate the server. Nonetheless, because of the rather straightforward setup and communication, it is used extensively in research projects, e.g. (Huang et al. 2012; Schwarz et al. 2012; Ballendat et al. 2010; Rädle et al. 2014; Dearman et al. 2012; Yang and Wigdor 2014; Hamilton and Wigdor 2014; Maekawa et al. 2004). The types of communication in these projects spans a large spectrum, including HTTP, REST API, direct TCP, UDP WebSocket, or an entirely custom web service.

Shared remote servers are also the basis of cloud services (i.e. Dropbox). These services allow users to store data on remote servers and thereby make the data available on multiple devices. This enables synchronisation of data, preferences, or accessing purchases on every device. The data synchronisation works over an active internet connection. Sometimes, data or parts of it can be stored locally, allowing access even if no internet connection is currently available. A shared user account on all devices is used to identify which data belongs to a device, requiring users to login on all their devices. This is a typical case of *sequential use* (Jokela et al. 2015), as users store data on one device, move to another device and continue work. Cloud services rarely enable parallel use of multiple devices, as this leads to synchronisation conflicts. Further, they do not allow devices in close proximity to interact.

Dearman et al. (Dearman et al. 2012) do not use a traditional client-server communication but make use of an online service in order to process images taken with the participating devices camera. Still, the benefits and shortcomings of network communication apply to this service as well.

HuddleLamp (Rädle et al. 2014) offers HTTP and WebSocket servers that devices connect to, enabling web applications to know about the location and size of other devices. This approach allows most web-enabled devices to participate in a “huddle” and run HuddleLamp applications, without prior software installation or a priori knowledge of the Huddle system.

2.5.2 Prepared Networking Environments

Instead of setting up a remote server that relays communication between devices, some research projects also assume that all participating devices are connected to the same wireless network, e.g. (Holmquist et al. 2001; Lucero et al. 2011; Schmitz et al. 2010). This way, devices can directly communicate with each other and send information. Again, a priori knowledge about the network and configuration of it is necessary. Also, detecting the correct device on the network and sending packets to it can be a challenge.

To solve this, Lucero et al. (Lucero et al. 2011) use a specially prepared Wi-Fi network that broadcasts all received packets to all connected devices. This way, devices do not need a priori knowledge about the number of devices or their network addresses.

Some projects do not detail how they solved the problem of finding and communicating with devices. For research prototypes, it is likely that the network addresses of participating devices are coded into the application beforehand. While this is feasible for prototypes with a small number of devices, it is not flexible enough for real-world deployment with a volatile nature and a large number of devices.

2.5.3 Short-Range Wireless Communication

NFC and Bluetooth were discussed as methods to detect nearby devices, and newer iterations are also capable of ad hoc data exchange. The range and stability of such communications can vary, though, and relatively large delays and low bandwidth make it difficult to transfer large amounts of data.

Pinch (Ohta and Tanaka 2012) uses Bluetooth packet broadcasting to exchange data and introduces some prototypical applications based on that. This allows Pinch to operate practically everywhere, but also imposes restrictions, such as a small number of devices and the range and amount of data that can be exchanged. Nonetheless, it demonstrates that off-the-shelf consumer devices are capable of ad hoc communication without the need for further augmentation and without extensive a priori agreements.

Since the release of Bluetooth Low Energy (BTLE), the reduced energy consumption and the ability to create connections without user authorisation made Bluetooth feasible for new applications in commercial products. Consequently, multiple commercial products that communicate with nearby devices emerged recently. For example, technologies, such as Apple's Continuity, allow users to move application state from one device to a nearby device instantaneously. It does not require an internet connection, but for security reasons requires a shared login on all participating devices.

NFC has also been explored in commercial products, such as Android Beam[9]. Here, putting two devices back to back will allow them to share the current application state, such as the currently active website, using NFC. Shared logins are not necessary, but in turn the shareable content is limited.

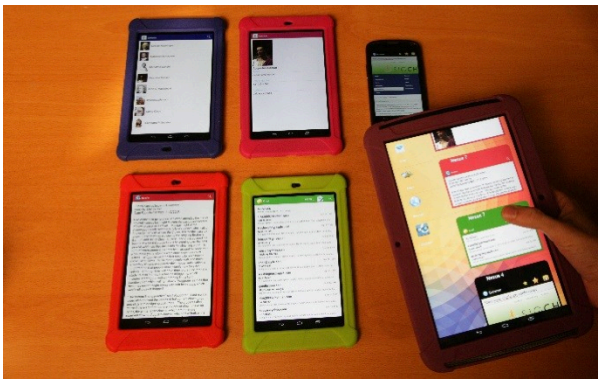

Figure 6: Conductor (Hamilton and Wigdor 2014) allows users to send cues to nearby devices, similar to notifications, and respond to those cues. Source: (Hamilton and Wigdor 2014).

Conductor (Hamilton and Wigdor 2014) uses NFC to handshake information that enables devices to join an application ad hoc. The devices need to be in close proximity to do so. After the initial handshake, the devices join a common, remote WebSocket server and therefore do not need to transfer large amounts of data over NFC. This approach requires an internet connection on all participating devices.

2.5.4 Discussion

Similar to device detection and tracking, there is a trade-off to be made when it comes to device communication: Some projects rely on prepared remote servers for a shared communication, requiring all participating devices to know the server beforehand, the server to be set up and maintained and an internet connection being available on all devices. These solutions often require authentication of devices (e.g., a shared login). Some projects also rely on a shared Wi-Fi network which is the technical solution with the lowest delay and fastest bandwidth but requires setup and configuration prior to usage, imposes restrictions on the application development, and requires devices to reside in the same network.

Other projects use Bluetooth or NFC for ad hoc communication with nearby devices. This makes it more difficult to retrieve shared assets, store permanent information and transfer large amounts of data, but is not dependent on external factors. These technologies have also seen recent deployment in a number of commercial products.

[9] Android Beam - Google Support — https://support.google.com/nexus/answer/2781895?hl=en — Accessed December 1, 2015

2.6 Development Support

Ways to detect, track, connect and communicate with devices are the technical foundation for cross-device interaction, but support for the development of novel applications and interactions based on this foundation is required to establish such technologies outside of the lab. This section examines existing research for the possibility to create and develop cross-device applications.

HuddleLamp's (Rädle et al. 2014) processing server application is offered for download[10]. HuddleLamp applications are built using web technologies and therefore deployable on a wide variety of web-enabled consumer devices. To run, HuddleLamp applications do require appropriate cameras and a server, both of which must be set up appropriately. The Huddle Orbiter[11] is a free service that simulates a HuddleLamp and allows testing without a camera.

Panelrama (Yang and Wigdor 2014) is a web-based framework developed solely for the creation of distributed user interfaces. It expands on existing web standards by requiring developers to divide their UI into panels using HTML. Properties are attached to panels using JavaScript, and Panelrama then distributes the panels to available devices. Panel states are synchronised over a shared server.

Weave (Chi and Li 2015) is a web-based, high-level scripting extension that aims to ease the control of the input and output of multiple devices. It allows for the distribution of UIs across multiple devices using selection criteria such as screen estate, sensors, input modalities and more. Further, Weave features a web-based programming IDE (Integrated Development Environment) that helps developers in creating and testing Weave applications. It has the ability to simulate different devices, which allows debugging and testing without hardware. While Weave is web-based, it does not automatically run on every web-enabled device, but needs a proxy application to be installed on every participating device.

The Proximity Toolkit (Ballendat et al. 2010; Marquardt et al. 2011) is available for download[12]. While being a powerful tool for the creation of cross-device applications, the toolkit requires a large amount of hardware and extensive preparation, including an entire room augmented with tracking cameras and a 3D model of that room, to enable cross-device interaction.

[10] HuddleLamp Homepage — http://huddlelamp.org — Accessed December 1, 2015
[11] Huddle Orbiter Homepage — http://orbiter.huddlelamp.org — Accessed December 1, 2015
[12] Proximity Toolkit Download Page — http://grouplab.cpsc.ucalgary.ca/cookbook/index.php/Toolkits/ProximityToolkit — Accessed December 1, 2015

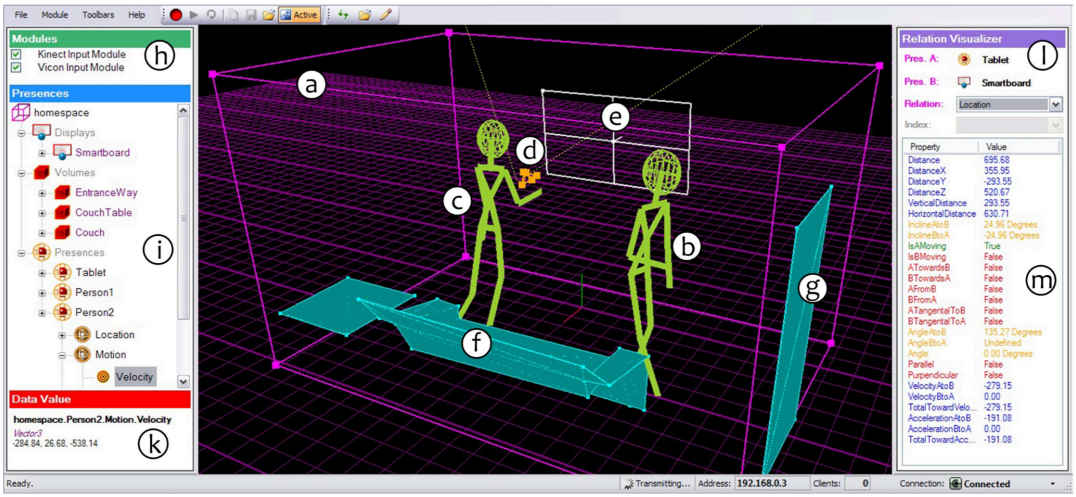

Figure 7: The Proximity Toolkit (Marquardt et al. 2011). It displays a room model as well as a live view of tracked people, devices and objects in a 3D rendering. Source: (Marquardt et al. 2011).

Conductor (Hamilton and Wigdor 2014) is a prototypical framework that explores ways of information sharing between devices. It introduces cues where users can broadcast an item or piece of information to all other participating devices and users then physically select the device to receive the item on. They offer a framework API for developers, but Hamilton and Wigdor do not describe the API in detail.

XDStudio (Nebeling et al. 2014) is a web-based authoring environment that focuses on the development of distributed user interfaces. It is based on "distribution profiles" that define devices and user roles of an application. Interfaces can then be distributed based on these profiles. The IDE allows simulation of devices or development on actual devices.

An example of commercial cross-device development support is *Google Nearby*[13]. It is an API that allows devices to publish payloads and nearby devices to receive them. It is implemented for Android and iOS. It shares data over Bluetooth or Wi-Fi, but requires all devices to be connected to the internet. Devices can then detect each other, connect and exchange payloads. Based on this, applications such as a shared whiteboard become possible.

Apache Cordova[14] is a development framework based on web technologies. It allows for the creation of native applications using HTML, CSS and JavaScript. Furthermore,

[13] Google Nearby Homepage — `https://developers.google.com/nearby/` — Accessed December 1, 2015

[14] Apache Cordova Homepage — `http://cordova.apache.org` — Accessed December 1, 2015

it provides APIs to access system functionalities from those web applications, such as device sensors. Applications can then be exported for every target platform and distributed to devices.

In conclusion, cross-device development is still difficult, although there are recent advances. Most frameworks can be considered special-purpose frameworks, aimed at certain scenarios or interaction techniques. Web-based frameworks seem most prominent, and projects like HuddleLamp demonstrate that the high availability of web technologies across off-the-shelf devices allows web developers to target a large number of consumer devices. Still, a general-purpose framework based on existing standards that is more widely accepted by developers and researchers is still missing, and most of the frameworks either require special setup, scenario or hardware and are therefore difficult to deploy and use in everyday scenarios.

2.7 Interaction Techniques

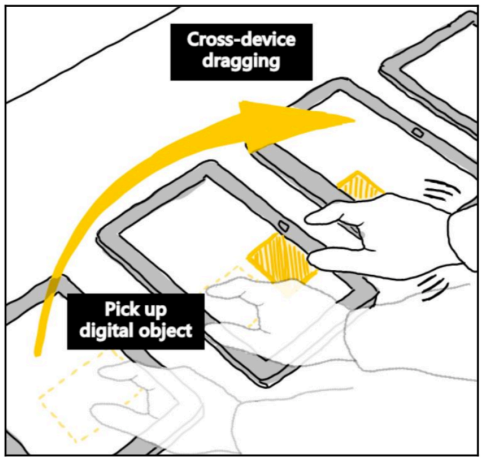

Figure 8: HuddleLamp (Rädle et al. 2014) introduces "pick, drag and drop", where a user picks up an object on one device and moves it to another device. Source: (Rädle et al. 2014) (parts of the image were cropped).

Based on the ability to sense and establish a communication between devices, new interaction techniques can be developed that go beyond interacting with just a single device. This section introduces a selection to illustrate the possibilities enabled by these technologies.

Siftables (Merrill et al. 2007) support a unique interaction language based on "the skill that humans have in sifting, sorting and otherwise manipulating large numbers of small physical objects" (Merrill et al. 2007). Besides other techniques, Siftables support sorting digital objects by sorting the physical devices, cooperative display sharing or bumping devices against each other.

HuddleLamp (Rädle et al. 2014) uses digital image processing to not only track devices but also the hands of users. Based on this, they developed the "pick, drag and drop" gesture. Here, users can pick up an object on one device, drag it across device boundaries and drop it on another device (see Figure 8).

Numerous interaction techniques are introduced by the Proximity Toolkit (Ballendat et al. 2010). Due to their high level of augmentation and preparation, they are able to track different aspects, such as device movements, user movements, user gaze, usage of non-digital objects by users and more. For example, they introduced a media player application that automatically stops playing when users divert their attention to another user or a non-digital object such as a magazine. It briefly displays movie information if somebody enters the room. Furthermore, the toolkit allows objects to be used as remote controls, for example a movie can be paused and resumed by pointing a pen to the screen.

ConnectTables (Tandler et al. 2001) uses specially developed sketching boards that are able to detect and connect to each other. Tandler et al. introduce multiple pen gestures that users can perform, for example to seamlessly send data from one device to another or to combine them into a single large drawing canvas.

SurfaceLink's (Goel et al. 2014) approach of using acoustics to identify devices also allows them to introduce some novel interaction concepts: Users can swipe with a finger on the surface between devices, for example to share content. Furthermore, pinch and expand gestures (moving two fingers to each other or away from each other) between devices become possible, for example to group or ungroup devices. SurfaceLink is able to track the speed, duration, length and shape of such gestures and therefore enables between-device interaction without the need for further augmentation like a camera.

Pass-them-around (Lucero et al. 2011) uses the metaphor of passing paper photos around to share digital photos. Each device becomes a container for individual photos and can be tilted to reveal the next photo, similar to tilting a stack of photos.

Hinckley et al. introduced a cross-device pen gesture to stitch devices (Hinckley et al. 2004). A pen can be used to draw across two devices to bind them together, thus allowing data transfer and other actions between the devices.

Hinckley uses synchronous gestures, such as bumping devices together, to achieve dynamic display tiling (Hinckley 2003). Prototypical tablets are bumped into another. Based on the accelerometer values of both tablets, the bumped edges can be determined and the position of tablets is calculated. Tablets are assumed to lie in a grid for this to work. Based on this, a large image is then distributed across the tablets.

Pinch (Ohta and Tanaka 2012) introduces a cross-device pinching gesture based on synchronised swiping gestures across two devices. A swipe to a device edge on one device is shared with other devices. If two such events occur roughly at the same time, they are considered a cross-device pinch. Pinch concludes the relative device positions from this gesture and by that enables content across more than one screen, such as a tiled image that spans multiple screens. Since Pinch is based on Bluetooth communication, this cross-device gesture is possible in an ad hoc manner.

Figure 9: Example applications implemented using Pinch (Ohta and Tanaka 2012). After pinching, stitched user interfaces across multiple devices are enabled. Source: (Ohta and Tanaka 2012).

These interaction techniques demonstrate the possibilities of cross-device gestures. In order to be able to track people, body parts or non-digital objects, camera augmentation and digital image processing is necessary. When relying on off-the-shelf devices, the devices' internal sensors can be used for novel interaction techniques, such as synchronised touch gestures or bumping of devices.

2.8 Conclusion: State of the Art

In this chapter we performed a deep analysis of the different aspects of existing research and commercial cross-device projects. In conclusion, while the research in all of these areas is extensive, actual cross-device support in everyday life and commercial products is still limited. Our findings indicate that today's applications support serial interaction across devices at best, but lack support for parallel use of devices on the same task.

Existing research focuses mostly on mobile devices, even though studies show that laptop and desktop computers are still a large part of professional work. Furthermore, cross-device projects face a trade-off between heavy augmentation versus light augmentation. While heavy augmentation enables increased awareness and allows for more advanced and diverse cross-device interaction, light augmentation enables ad

hoc use of cross-device interaction with consumer devices and at a multitude of locations and scenarios. In the past, most research has focused on heavy augmentation. Recent research has started to explore cross-device interaction with off-the-shelf devices, in particular using short-range wireless technologies to make devices aware of each other and communicate directly.

Communication between devices was found to be problematic: Cross-device interaction requires all devices to communicate with each other fast and reliably. While remote servers can partly satisfy this need, they hinder ad hoc interaction, require all devices to know the server address prior to use and to have a permanent internet connection. Online data storage is considered a security risk by users and problematic if one wants to enable parallel use of multiple devices.

Furthermore, we discovered a lack of development support for cross-device interaction, but also saw an emergence of web-based solutions that are supported across a variety of platforms and hardware. Nonetheless, most developer tools that supported cross-device interaction were tied to certain hardware or aimed at a certain scenario or use of devices. A more general-purpose solution was lacking.

From this analysis, we will derive requirements for deploying real-world cross-device interaction in the next chapter.

CHAPTER 3

CONNICHIWA: AN AD HOC CROSS-DEVICE FRAMEWORK

3.1 Requirements

Based on the state-of-the-art analysis in Chapter 2, we identified several remaining problems when it comes to ad hoc cross-device interaction. In particular, a tradeoff between augmentation and off-the-shelf devices became apparent, whereas research using the latter only emerged recently with more advanced consumer devices. We also saw that modern consumer devices have the ability to detect and communicate with nearby devices, but nonetheless noticed an absence of cross-device interaction in everyday life. Guided by these conclusions and Weiser's vision of ubiquitous computing, we believe that modern off-the-shelf consumer devices can enable ad hoc cross-device interaction in everyday life. We therefore propose to intensify research in this direction. It is our belief that this will both increase acceptance by developers and everyday users and proliferate cross-device interaction. From our analysis, we derive seven requirements that we belief to be essential to achieve this goal:

R1 Low threshold to join: A lot of current research in cross-device interaction is only feasible for research prototypes and inside labs. The complexity of setting up and joining such interaction is too high for ordinary users. Enabling everyone to participate in cross-device interaction by using the devices they already own lowers this threshold significantly. Furthermore, complex software-side setups are another barrier and an ad hoc technology with minimal to no configuration effort is required.

R2 Independence of location: Augmenting the environment does enable accurate live-tracking of devices but confines interaction to a small space, such as a table or a room. Users must be able to engage in cross-device interaction anywhere and at any time: At home, at the workplace or even in the wild. Therefore, only a location-independent approach will see widespread deployment.

R3 Cope with volatile device ecologies: Users must rely on whatever devices are currently available to them in their surrounding. Device ecologies, in particular when using mobile devices, are very volatile. Users must be able to add and remove devices and change device roles at any time during a task. Cross-device technologies must support the vast amount of fluently changing hardware and role combinations.

R4 Support versatile application scenarios: Multiple research prototypes focus on a specific scenario or example applications. For in-the-wild deployment of cross-device interaction, a future technology should give developers the freedom to implement whatever they desire. This will proliferate a wide variety of applications that integrate into the different parts of a user's life. Supporting not only mobile devices but also traditional desktop or laptop computers is an important part of this requirement.

R5 Support parallel interaction: Current applications support sequential cross-device use, but we identified a clear lack of parallel interaction support. Studies have shown that users desire such interaction and that it can benefit their workflows. Therefore, an everyday cross-device technology must support sequential use as well as parallel interaction.

R6 Direct data exchange: It was shown that users perceive data exchange over the cloud as a potential security risk, that it is mostly sequential in nature and that it is technically limited (e.g., by requiring an internet connection). Therefore, a future cross-device technology must support direct data exchange between devices and cope with distributed resources, e.g. by transferring such resources automatically to devices that need them.

R7 Small development effort: High development effort, for example the need to port an application to all possible platforms, can lead to developers rejecting a technology, and in turn will lead to an absence of applications for users. Therefore, developers must be able to create cross-device experiences for users and adapt to the multitude of platforms without migration efforts.

We developed a novel cross-device technology based on these key requirements. This chapter will introduce the reader to the development of a prototypical cross-device framework. The framework allows us to implement new concepts for cross-device technologies and based on this implementation, we evaluate the potential of the implemented solutions in a two-fold manner: 1) By evaluating the framework with developers to test the programming API and 2) by implementing example applications that test the framework's capabilities, the potential for novel interactions, and the support for *ad hoc* cross-device interaction. This evaluation and its results will be described in *Evaluation* (Chapter 4, p. 47). We will then derive design guidelines for future cross-device technologies in *Ad Hoc Cross-Device Web Extensions* (Chapter 5, p. 63).

3.2 Analysis of Basic Technologies

Before we started developing the framework, a suitable basic technology was required. This choice cannot be taken lightly, as the benefits and restrictions of the chosen basic technology will also inform the design and features of our framework. Therefore, this section presents and analyses a selection of possible basic technologies that were considered at the beginning of the Master's project preceding this thesis.

3.2.1 Java

One of the earliest cross-platform technologies is Java[15]. Java compiles source code into bytecode that can then be executed on different platforms using Java Virtual Machines. The virtual machine has to be developed for every platform that wants to run Java applications. A Java Virtual Machine exists for most desktop architectures and operating systems today, but is often lacking for other platforms. iOS or Windows Phone devices are not Java capable. Android phones, even though based on Java, are not able to execute desktop Java applications due to different base SDKs being used. Other devices, such as gaming consoles, digital cameras, or watches are often not capable of running Java applications as well.

Due to the "common ground" that Java has to seek for its applications, Java often fails to integrate itself into the host system and make use of the host's distinct features, styles and optimisations. Therefore, Java applications often feature a largely different visual style than native applications and fail to adapt to high-resolution screens, multiple graphic cards, integrated versioning systems, or access to central information such as contacts or calendar.

Java applications can be less performant than native applications, and memory usage is generally considered higher. The actual performance is largely dependent on the used virtual machine.

3.2.2 C++

C++ is an object-oriented extension of the C language and available on all major systems. It was highly popular in the 1990s because of its good performance and the ability for machine-oriented programming.

C++ does run on most platforms and even mobile operating systems. It is rarely the preferred language on modern platforms and support for deploying C++ applications is limited. Hence, deployment can be difficult. While the C++ core is highly standardised across platforms, extension libraries are not, and – in reality – it is still difficult to create a C++ application that is easily compilable on all platforms. Further, due to the age of C++ it does not adjust well to hardware differences such as input modality, display density, and so on. Creating unified experiences across devices is therefore difficult in C++ .

In general, C++ is considered complex and difficult to learn. The language is error-prone, in particular because of the absence of features such as a garbage collectors. This increases the threshold for developers.

[15] Oracle Java — https://www.java.com/download/ — Accessed December 1, 2015

3.2.3 Modern Native Languages

C++ has been largely replaced by more "modern" languages. For example, on Windows platforms, C# is the preferred language to create applications while Mac OS X applications are mostly based on Objective-C. While such languages often offer good performance, are integrated into their platform and provide modern programming features, well-supported IDEs and allow for quick development, they do not provide cross-platform possibilities. Mac OS X applications, for example, are based on the Cocoa API, which is not available on other platforms. This makes such languages unfeasible for cross-device development, as applications would have to be ported to a variety of languages to run across different platforms.

3.2.4 Qt

Qt[16] allows for development of GUI-based applications across platforms. It is not a language in itself, but rather a library for C++ . In part, it shares C++'s benefits and shortcomings, namely very good performance but being complex to develop and not including some modern-day features, such as garbage collection.

Qt supports a variety of modern platforms, such as Windows, Mac OS X, Linux but also mobile platforms such as iOS or Android. It is also possible to create native UIs for these platforms, allowing for a good integration into the host system. It does not support more specialised platforms such as gaming consoles or smartwatches. Qt requires separate compilation and deployment for every target platform.

While Qt is open source, it is not an open standard, requiring the purchase of a license for commercial use. This can lead to conflicts between Qt's license and the application's license.

3.2.5 The Web

Today, "the web" describes HTML, CSS and JavaScript, three languages used for markup, styling and scripting. It provides an alternative to pre-compiled applications. The general architecture of the web is somewhat similar to the approach taken by Java, with the source code being interpreted by the web client when an application runs. The two still differ in that no bytecode is generated and, technically, web clients do not compile anything. This makes deployment of web applications very easy. Furthermore, web applications have a client-server architecture, which is unique in the presented technologies. Since web applications are not compiled nor saved locally (except short-term caches), this requires devices to have access to the server, often through an internet connection. On the other hand, it further eases deployment, as distribution through downloading is built into the core architecture.

[16] Qt Homepage — http://www.qt.io — Accessed December 1, 2015

The web further stands out from other technologies with its massive availability and large standardisation across consumer devices[17]. Most web features are available across platforms in a unified manner, giving users a similar experience on all platforms. Web applications run on mobile and desktop systems, TVs, gaming consoles, digital cameras, watches, across different hardware, screen sizes, and input modalities. Almost any modern device has a built-in web client, making setup steps and software installation obsolete. Furthermore, developers can rely on a single set of languages for standardised markup, styling, and scripting across devices. Still, minor differences in how different web clients handle particular features can provide slightly different experiences across platforms. Also, similar to Java, web applications often fail to integrate into their host system. They do not have the look and feel of native applications and cannot take advantage of the host system's optimisations, such as automated versioning or multiple graphic cards. They do take advantage of hardware such as high-density screens and can adjust to differences in display density or input modality.

While the performance of web applications has increased dramatically over the last years, they still fall short compared to native applications.

3.2.6 Python

The scripting language Python[18] allows for scripting across multiple platforms. Python is highly flexible both in programming style – supporting object-orientation but also functional programming – and in functionality. Similar to web languages, the source code is run by an interpreter without compilation. Unlike the web, clients that execute python are not readily available across platforms and different tools are needed to package Python code into an executable. In general, the deployment process using Python can be considered complex, in particular when targeting a large number of platforms. Python, most of the time, can be considered less performant than native approaches.

3.2.7 Conclusion

Not surprisingly, none of the provided technologies comes without its own set of benefits and shortcomings. And while these technological foundations can provide means to develop and run applications across multiple devices, none incorporate a concept for the creation of applications that utilise multiple devices. Therefore, a novel concept for detection and communication with nearby devices is required regardless of the chosen technology.

Furthermore, the availability of these technologies differs: We discussed C++ and Python being widely available on most systems, but often lacking standardisation. Java

[17] Can I use - Web Feature Availability Overview — http://caniuse.com/#feature_sort=usr_score — Accessed December 1, 2015

[18] Python Homepage — https://www.python.org — Accessed December 1, 2015

Virtual Machines exist for most desktop systems, embedded systems and even some mobile systems, although most mobile systems and devices such as gaming consoles or smartwatches cannot run Java applications. Modern native languages are often tied to a small subset of systems. Qt is available on a very broad range of devices and web technologies stand out even more in this regard, being available on almost any modern consumer device.

Most of the technologies require software-side setups to work, for example installation of the Java Virtual Machine or the Python interpreter. Traditional devices (desktops, smartphones, tablets) come with a C++ compiler pre-installed. Web clients are readily available on even more platforms such as smart TVs. Further, no setup or installation is required when running a web application, except moving to the appropriate server address. In turn, web applications require internet access to be downloaded, executed, and during runtime.

Development support for cross-device interaction is practically non-existent with any technology. Even technologies such as Java do not provide unified feature support or a unified user experience across platforms. In this regard, Qt and the web stand out. Qt offers libraries that adjust to all supported platforms and even adjust to the look and feel of the host system. The web tackles this issue with standardised markup, styling, and scripting and a high amount of standards across devices that are implemented by the web clients. Further, the web has integrated mechanisms to cope with device differences such as different resolutions or input modalities.

The deployment of applications is an important factor when facing a large number of devices that want to participate ad hoc. In particular with native applications, the deployment process can be cumbersome, as applications need to be copied or downloaded to every device. More closed platforms, such as iOS, require downloading the app from a store via a user login. The only technology that does not require deployment on every single device is the web, as the server-client architecture allows user to point to the server address and download the application automatically. In turn, web applications require an internet connection to work.

In conclusion, it was decided to focus on cross-device support using web technologies. The availability and standardisation is unmatched by any other technology, allowing for the creation of unified user experiences across a huge variety of different devices, but at the same time harnessing each device's individual strengths (e.g., keyboard, touch or stylus input). Web clients are available for almost any platform and contain an integrated mechanism to download and execute web applications without manual deployment. Web applications can adapt to device differences, such as high- vs. low-resolution screens or mouse vs. touch input. This greatly lowers the threshold for users to join applications. The appeal of the web is also backed up by the number of web-based research explored. Further, a lot of companies are focusing on the web in recent years, in particular transitioning applications to the web that have previously only been available as native applications (e.g. Microsoft Office, Mendeley[19], Spotify[20]).

[19] Mendeley Homepage — https://www.mendeley.com — Accessed December 1, 2015
[20] Spotify Web Player — https://play.spotify.com — Accessed December 1, 2015

We therefore believe web technologies to be a promising candidate to enable cross-device interaction across a large variety of devices. In the remainder of this chapter, we will provide an analysis of the remaining shortcomings of web technologies and describe a prototype framework that aims at reducing these shortcomings, making web development feasible for the creation of cross-device experiences.

3.3 Requirement Analysis: Web Technologies

This section will provide an overview of how well web technologies perform in each of the requirements defined in Section 3.1. From this, we derive remaining problems that still need to be solved.

R1 (Low threshold to join) As discussed, web technologies make installation and setup steps mostly obsolete, taking a lot of possible thresholds away from users. Web clients are readily available on most devices. Devices must be connected to the internet and then need to access a known internet address. While mechanisms such as QR codes can simplify access, this barrier should be removed. Furthermore, web applications rely on shared logins and do not provide means to join an application without users first logging in and sending invitations to other accounts. An ad hoc method of joining devices in close proximity and enable collaboration between multiple users must be found.

R2 (Independence of location) Web technologies are based on network communication and therefore require devices and server to reside on the same network. Most of the time, this means that the server must be reachable over the internet. While the availability of high-speed internet grows every year, some locations still suffer from poor cell reception (e.g., indoors, planes). Becoming independent from a stable, fast internet connection is therefore desirable. Furthermore, current web applications lack the ability to sense nearby devices. This makes it difficult to engage in cross-device interaction with local peers.

R3 (Cope with volatile device ecologies) Web applications support a wide variety of devices and platforms. Web technologies provide means for feature checking and adapting to device differences (e.g. resolution, input modalities). Therefore, for individual devices, the web adapts well to changes. Configuration changes of the device ecology are currently not integrated, due to lack of cross-device support. When developing a new framework, care has to be taken to allow applications to cope with changes in the device ecology in a similar manner as with differences of individual devices.

R4 (Support versatile application scenarios) The web languages are versatile and a wide variety of applications can be implemented using them. The web is under constant development and has gained a variety of powerful features over the last couple of years, such as 3D rendering, local storage, advanced multi-media support, access to device sensors, and many more. In this regard, the web fulfils this requirement well. When adding cross-device centric APIs to the web, care must be taken not to restrict these possibilities.

R5 (Support parallel interaction) Web applications do not currently feature parallel cross-device support. While shared logins can synchronise data across devices, web applications that utilise multiple devices at the same time are rare. Developers must be supported in creating such interaction in the future.

R6 (Direct data exchange) The network-based nature of the web requires all communication to flow through a remote server. This, again, requires shared logins and further is a security concern for most users. It also imposes restrictions on performance, since data must be exchanged over an internet connection. A more direct method of data exchange is missing in current web standards.

R7 (Small development effort) A single set of languages is available across all platforms. Further, adaption mechanisms allow to tailor web applications to devices (e.g., feature checking, CSS media queries). Web applications do not need to be compiled for individual platforms. Therefore, development and deployment effort is minimised compared to other technologies and this requirement is currently well supported.

3.4 Connichiwa

We developed a prototype framework – called *Connichiwa* (jap. こんにちは (konnichi wa, good day) + engl. connect) – that extends current web technologies to solve the shortcomings found in the previous section. This framework was developed in the project preceding this Master's thesis. The concepts introduced in the framework were then evaluated. This section will introduce the reader to the most important aspects of the framework and how the framework was implemented. Please note that this thesis will introduce the reader to the concepts of Connichiwa. For a full technical description of the framework, see the project report that preceded this thesis (Schreiner 2015). For a full overview of all Connichiwa resources available online see *Connichiwa Online Resources* (Appendix B, p. 81).

3.4.1 Prototyping Web Extensions

To extend web standards, pure JavaScript was not sufficient. It was clear that certain features (e.g., the ability to sense nearby devices) require deeper access to system.

Since extending web rendering engines was out of scope for this Master's project, Connichiwa replaces the browser with a custom web client application. This client takes the place of the usual web client, but does not implement all features of a typical web browser in the prototype. It exploits the `JavaScriptCore` bridge of iOS 7 and above and the `JavaScriptInterface` of Android 4.2 and above to extend the core rendering engine (either Safari on iOS or Chrome on Android). These bridges enable redirection of JavaScript calls to an iOS/Android native application and vice versa. This gave us the freedom to implement and test new web features in a prototypical manner without large efforts. These bridging mechanisms will be called *JavaScript bridges* for the remainder of this text.

A device that runs this Connichiwa web client is called the *master device*. Devices that join a Connichiwa application are called *remote devices*. They can be thought of as server and clients. A Connichiwa application must have exactly one master device, but can have an arbitrary number of remote devices. Remote devices can further connect and disconnect at any time, while the master device must maintain the application running during the entire lifecycle. The terms master device and remote device will be used throughout the rest of this document.

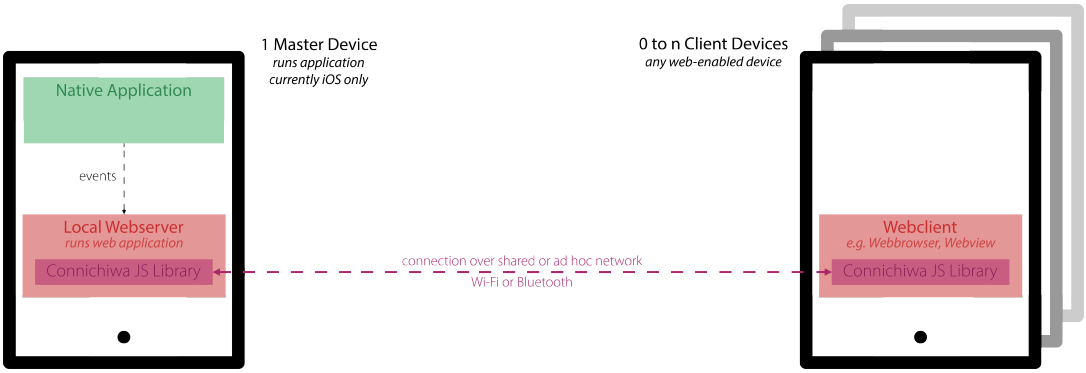

Figure 10: Architectural overview of Connichiwa. One master device runs an embedded web server. Web clients can connect, which will load the JavaScript library and establish a WebSocket connection between the devices.

It must be noted that the need for installation and deployment of the Connichiwa web client is contrary to R1 (low threshold in joining applications). Further, R3 also states that a wide variety of devices must be supported, something that is not guaranteed if a native application must be installed on every device. Connichiwa partly solves these limitations: While the master device does require installation of a native application, remote devices only need an ordinary web client to join an application (see Figure 10). All web-enabled devices can therefore still participate in a Connichiwa application and the benefits of web applications are retained. For certain features, remote devices must also run the Connichiwa web client (see Figure 11). Further, it is possible to mix devices running the Connichiwa web client with devices that use a traditional web client.

We put great development effort in eliminating as many of the current limitations as possible (see *Future Work* (Section 6.1, p. 67)).

In the future, making Connichiwa's features part of web standards and implementing them directly into the browser would ensure that both R1 and R3 will be fully satisfied. We propose such new web standards in *Ad Hoc Cross-Device Web Extensions* (Chapter 5, p. 63).

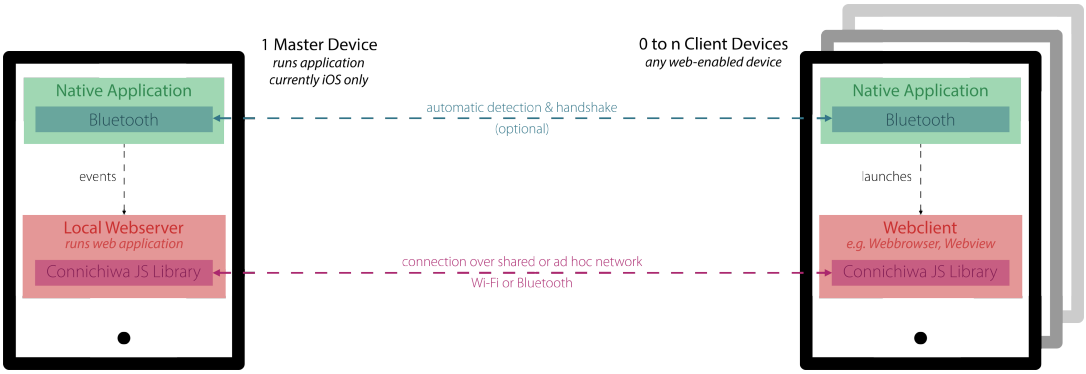

Figure 11: In addition to WebSocket communication (see Figure 10), client devices can run a native application and enable the full feature set of Connichiwa, such as ad hoc Bluetooth detection.

3.4.2 Hardware

The Connichiwa web client has been mainly developed for the iOS platform in the course of the Master's project[21]. Parts of the application have also been ported to Android[22]. Therefore, the master device must currently be an iOS or Android device.

Any device that features a web client with WebSocket support can become a remote device. This includes, but is not limited to, iOS devices back to the iPhone 3GS (2009) and iPad 2 (2011), Android devices running Android 4.4 (2013) or newer, all prominent desktop web browser since 2012 (and earlier for some browsers) but also devices such as modern TVs, digital cameras and more. According to icanuse.com[23], 88% of the web hits currently use browsers that support the WebSocket protocol.

3.4.3 Local Web Applications

Typical web communication runs through a remote server, which is not optimal considering our goal of location independence (R2). To solve this issue, Connichiwa enables *local web applications*. An embedded web server is launched on the master device on demand (see Figure 10). The device can then run a web application using the HTTP

[21] Connichiwa (iOS) on GitHub — `http://www.connichiwa.info` — Accessed December 1, 2015

[22] Connichiwa (Android) on GitHub — `http://android.connichiwa.info` — Accessed December 1, 2015

[23] Can I use... Support tables for HTML5, CSS3, etc — `http://caniuse.com/#search=websocket` — Accessed December 1, 2015

protocol that all web clients already understand. Potential remote devices can connect to the application through a shared network. Since the shared network might also be an ad hoc network (e.g. over Wi-Fi Direct or Bluetooth Personal Area Networks), communication between devices can be kept local and no external infrastructure, such as a dedicated server or a router, is required. Of course, a common public Wi-Fi can also be used to run Connichiwa applications. This makes applications truly location-independent and allows Connichiwa applications to run in the wild, even when no internet connection is available.

The iOS version of Connichiwa uses GCDWebServer[24] as its embedded HTTP server and the Android version uses NanoHTTP[25].

3.4.4 Sensing & Connecting Nearby Devices

While local web applications allow location-independent web applications and therefore support R2, the necessity to manually connect each device to the embedded web server heightens the threshold for joining (R1). In particular with a large number of devices, constant manual connection to the application is not acceptable.

Therefore, Connichiwa required the ability to sense physically close devices automatically and be able to invite them to join an application. This provides a seamless experience for users, allowing them to join or leave sessions spontaneously. Examples for such a local awareness have already been provided in the analysis of related work (Ohta and Tanaka 2012; Hamilton and Wigdor 2014). Therefore, we extended current web standards with support for short-range wireless technologies. Connichiwa supports Bluetooth Low Energy for detection of nearby devices, which is available on most modern devices. Other technologies, such as NFC, can be used as well in future implementations and it is even thinkable that the framework intelligently picks the best technology available. Devices that support more than one technology could also act as a bridge between other devices in such an implementation.

For security reasons, it was decided to not give web developers direct access to Bluetooth and therefore direct means of communication to nearby devices. Instead, the Connichiwa web client continually advertises itself over Bluetooth and at the same time picks up advertisements from other devices in the background. If nearby devices running Connichiwa are detected, the JavaScript bridge is used to send an event to the web application. The application can then decide to invite a remote device to join and, if accepted, the two devices will handshake the necessary information such as the network state of both devices over their Bluetooth connection. The remote device will then use this information and establish an HTTP connection to the master device using an ordinary web view.

[24] GCDWebServer on GitHub — https://github.com/swisspol/GCDWebServer — Accessed December 1, 2015

[25] NanoHTTPD on GitHub — https://github.com/NanoHttpd/nanohttpd — Accessed December 1, 2015

The entire process can be performed seamlessly in the background without user interaction. Therefore, two devices close to each other can join into a single application without any user interaction and provide a truly ad hoc experience. This improves goals R1, R3 and R6.

3.4.5 Communication Between Devices

In *Device Communication* (Section 2.5, p. 14) we have seen multiple ways of communicating between devices, such as a shared server, prepared network environments or short-range wireless technologies. Connichiwa supports ad hoc detection and handshakes over Bluetooth Low Energy. While pure Bluetooth communication could also be used for data exchange, we saw with Pinch (Ohta and Tanaka 2012) that this is not suitable for demanding applications. Further, it would not allow devices with an ordinary web client to join. Because of this, Connichiwa harnesses common web protocols. The HTTP protocol is understood by every web client, and due to the server being embedded into the master device, Connichiwa can bypass shortcomings such as requirements for an internet connection and maintenance of a server. Because of the socket-based nature of web servers, the HTTP connection can be established over different types of IP-based networks, such as public Wi-Fi networks, ad hoc Wi-Fi but also ad hoc Bluetooth networks. This gives Connichiwa a large degree of freedom in its communication, while relying on an established protocol.

To communicate during runtime, Connichiwa uses the well-established and standardised WebSocket protocol. The WebSocket server is launched on the master device at the same time as the HTTP server. The WebSocket server then acts as a message relay system: It receives messages from the devices, determines the target device the message is addressed to and delivers the message. Each device generates a unique ID on launch and makes this ID known to the WebSocket server. The ID can then be used to identify the device uniquely in messages over WebSocket.

The framework will take care of automatically establishing the WebSocket connection once a device connects to the HTTP server. It will further take care of any bootstrapping work. As we will see in *JavaScript* (Section 3.4.8, p. 37), the framework also extends the JavaScript language so that developers can easily work with remote devices and communicate with them. Developers do not need to care about the nature of the communication and do not need to be concerned with the protocol or format used for data exchange.

For its WebSocket server implementation, Connichiwa uses BLWebSocketsServer[26] on iOS and NanoHTTP on Android. Any embeddable WebSocket server is suitable and efficiency is of utmost importance since most communication during application runtime will use this server.

[26] BLWebSocketsServer on GitHub — `https://github.com/benlodotcom/BLWebSocketsServer` — Accessed December 1, 2015

3.4.6 Tracking Nearby Devices

Proximity is a highly researched topic in cross-device interaction. Knowing the position of devices and possibly even people and non-digital objects enables a wide variety of interaction techniques. Unfortunately, current sensors in consumer devices do not allow for such an accurate tracking. As shown in Chapter 2, augmentations such as magnetic sleeves or rooms equipped with tracking cameras are required to enable these styles of interaction, which would be contrary to R1 and R2.

Connichiwa reduces this trade-off by providing an approximation of device distances using Bluetooth. This approach is based on the received signal strength of the Bluetooth signals of other devices. It further uses the estimated signal strength at 1m distance (the so-called *TX Power*). Based on path-loss models and curve-fitting, the distance of devices can be approximated. Connichiwa further applies an estimated moving average on the measurements to increase robustness against outliers. This approach provides radial distances, not directed positions. The details of these calculations can be found in the project report that preceded this thesis (Schreiner 2015).

Changes in a device's approximated distance triggers an event in the web application. This allows for advanced interactions, such as devices changing roles based on their proximity.

In *Static Position Detection* (Section 3.5.1, p. 40), we will further describe means for more accurate but static position detection using cross-device gestures.

3.4.7 Providing Hardware Information

As shown by (Santosa and Wigdor 2013), users select device roles mainly based on the physical affordance of devices. In particular due to the volatile nature of cross-device applications and the potentially large amounts of devices, applications should assign device roles automatically where possible. To do so, web applications must gain access to information about the physical affordances of the current device ecology. As of now, JavaScript can access only limited information about devices, such as resolution or operating system. Other information such as physical screen size, input capabilities, available memory, or attached hardware are not discoverable through an API.

For example, imagine a multi-device presentation software. When the presentation starts, it needs to find nearby large screens or computers with attached projectors to display the actual presentation. At the same time, the presenter's close-by smartphone or smartwatch can be used to display controls, and the presenter's laptop can display presenter notes. Attendees of the presentation can access the application with their own personal device to see the presentation on their device, annotate it and ask questions that will then appear right in the main presentation. This presentation software consists of multiple device roles: presentation device, controller device, note device

and attendee device. With information about the devices available, such as proximity or form-factor, the application can assign these roles automatically.

To achieve this, Connichiwa uses its native application to retrieve additional information, and then makes them available to the application using the JavaScript bridges. Parts of the information – such as the screen size or the canonical name – will be sent to other devices during the initial handshake. Other information will be made available when a device connected. Currently, device name, network state, operating system, screen size, and pixel density are made available. Other information, such as attached hardware or available memory are not retrieved by Connichiwa, but the framework is built in a way that it can be easily extended to include this information.

3.4.8 Web Language Extensions

Current web standards do not provide APIs for multi-device support, a necessary step to give developers the means to create applications across devices. Connichiwa therefore provides a two-fold extension to web languages to tackle this issue: a JavaScript extension as well as an HTML extension. Unfortunately, it is currently not easily possible to inject CSS extensions. An outlook on possible future extensions of the CSS language will be given in *Ad Hoc Cross-Device Web Extensions* (Chapter 5, p. 63).

JavaScript

Connichiwa provides a JavaScript library that takes care of communication with the native application and other devices and provides cross-device functionality in an abstracted way. The framework takes care of automatically loading the library on remote devices. Developers must load the library in their web application's root file.

The JavaScript API provides an event system. System events, such as devices being detected, devices moving, or devices disconnecting or moving out of range, are delivered to the web application. Web applications can request these events with a single call to the API (see Listing 1). Further, developers can trigger and register for custom events.

Furthermore, Connichiwa abstracts information about other devices into *device objects*. Device objects encapsulate all information about a device and offer the ability to retrieve these information and perform actions on the device. For example, one can retrieve the pixel density of another device's display with a single method call, load a JavaScript or CSS file on the other device or update the content shown on the device. An example of how to work with remote devices in Connichiwa can be found in Listing 1. It is important to note that this approach abstracts the underlying communication mechanism from developers: Web developers do not need to know about or care about the established WebSocket connection, the protocol to exchange information and how to retrieve the messages and parse them on the other device. In fact, the

mechanism of communication can be exchanged in the future (i.e. to a WebRTC[27]-based approach) without alteration of the API. The same is true for the protocol used for communication.

```
Connichiwa.on("deviceDetected", function(device) {
  if (device.getDistance() < 5.0) {
    device.connect();
  }
});

Connichiwa.on("deviceConnected", function(device) {
  device.loadCSS("infoPanel.css");
  device.insert(moreInfoPanel);
});

Connichiwa.on("deviceDistanceChanged", function(device) {
  if (device.getDistance() < 1.0) {
    device.replace(moreInfoPanel, expandedInfoPanel);
  }
});
```

Listing 1: Example JavaScript code, illustrating how device objects and events help developers to work with remote devices.

Additionally, Connichiwa allows developers to exchange custom messages with arbitrary data. This allows them to use the same familiar API as with system messages, but gives them complete freedom for their application, something particularly important considering our goal of application versatility (R4).

Based on this, Connichiwa implements plugins that aim at taking often-needed or tedious work from developers. These plugins will be further described in *Plugins* (Section 3.5, p. 40). A full documentation of the entire Connichiwa JavaScript API is available online[28].

HTML

HTML is the markup language used on the web to define the semantic structure of a document. Traditionally, a single HTML file is loaded on a single device, parsed, and rendered by the browser. CSS adds styling to the components of the document and JavaScript can dynamically modify the HTML and CSS.

[27] WebRTC — http://www.webrtc.org — Accessed December 1, 2015

[28] Connichiwa JavaScript API Documentation — http://docs.connichiwa.info — Accessed December 1, 2015

With multiple devices, the user interface defined by HTML becomes a *distributed* user interface. Devices can connect at any time during runtime the UI need to adjust to changes in the device ecology. Hence, HTML needs to adapt. Reusability of HTML code becomes more important, as it can reduce code repetitions. Consistency and synchronisation between the different parts of the UI become important, but is problematic, something already discovered in *Challenges in Distributed UIs* (Section 2.3, p. 9).

Therefore, we added a templating mechanism to HTML that makes working in the web similar to a Model-View-Controller approach used in other languages. The HTML now consists of *templates* (Views) and JavaScript (the Controller) only modifies the data (Model). When the model is altered, Connichiwa a) propagates these changes to all devices and b) ensures that all views update automatically to reflect the latest data. This approach is often called *data-driven* or *reactive*. For distributed UIs, this can provide valuable advantages: Code reusability is increased, views can be reused, either on the same device or on different devices, and it ensures consistency of the UI.

To make use of these features, developers can create `.html` files that contain `<template>` tags. Each template represents a view. Developers can load templates and insert them into a device's UI using JavaScript. A model can be attached to a view and the same model can be used on as many views as required. Views contain *expressions* to access the model. For example, the template content `<h2>Hello, {{name}}</h2>` will replace the expression `{{name}}` with the value of the `name` variable stored in the model. Whenever the value of that variable changes, the view (and all other views using this model) will automatically update to reflect the new value. An example of how developers can make use of this approach can be seen in Listings 2 and 3.

The engine powering this reactive behaviour in Connichiwa is Ractive.js[29]. It provides the ability for reactive HTML source codes. Connichiwa extends this engine with the ability to create reusable templates and for cross-device support (automatic synchronisation of models, reactive behaviour on remote devices, inserting templates into remote devices).

```
<template name="gallery">
  {{#each images}}
    <div class="galleryImage">
      <img src="{{url}}"><br>
      {{description}}<br>
    </div>
  {{/each}}
</template>
```

Listing 2: Example HTML template in Connichiwa. The template can contain expressions (such as loops or variables) that are tied to a data model. The model is set using JavaScript (see Listing 3).

[29] Ractive.js Homepage — http://www.ractivejs.org — Accessed December 1, 2015

```
Connichiwa.on("deviceConnected", function(device) {
  device.loadTemplates("template.html");
  device.insertTemplate("gallery", { target: "body" });
  CWTemplates.set("images", [
    { url: "/images/1.png", description: "A beautiful bird." },
    { url: "/images/2.png", description: "A large building." },
  ]);
});
```

Listing 3: Example JavaScript code of how to load and insert a template. `CWTemplates.set` sets data in the template's model to a value, in this case an array of image information. The model can be accessed in the template as seen in Listing 2.

3.5 Plugins

Based on the support for detection and communication with nearby devices as well as the described extensions for JavaScript and HTML, Connichiwa further implements three plugins that solve often-needed problems in cross-device interaction and communication. This section will describe these plugins shortly.

3.5.1 Static Position Detection

As described in *Tracking Nearby Devices* (Section 3.4.6, p. 36), Connichiwa features a live tracking of nearby devices based on Bluetooth signal strength. While useful when requiring an overview of the current device ecology, this approach only offers an estimated radial distance measurement. To achieve a more accurate position tracking of devices, the integrated sensors are not sufficient. For this reason, Connichiwa uses cross-device gestures to enable a more accurate, but static position detection.

Ohta and Tanaka already described this approach, introducing a pinch gesture to determine the position of two devices lying edge-to-edge (Ohta and Tanaka 2012). Based on the finger position during the pinch, the relative position of the devices can be calculated. A user needs to move a finger on each device from the device center towards the device edge where the other device is located. This gesture must be performed on both devices at the same time, so that the two fingers meet on the device edges (see Figure 12). Together, the finger movement can be thought of as a pinch gesture across two devices.

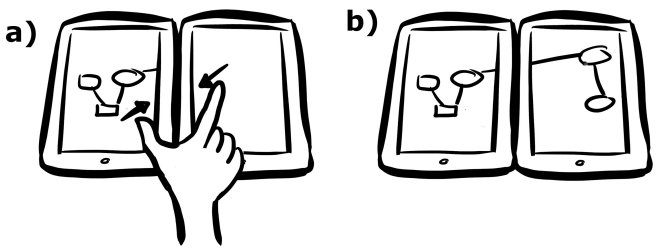

Figure 12: a) The user performs a pinching gesture over two devices b) The gesture location is used to determine the device's position, enabling homogenous content across both devices.

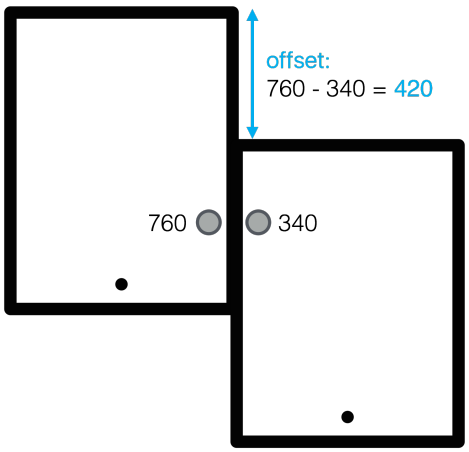

Figure 13: Offset calculation based on a synchronised swipe on two devices. The grey dots mark the y-locations of the two swipes with their local coordinates.

Connichiwa implements a similar approach and supports multi-device pinch gestures to determine relative device position. When the user performs a pinch gesture, the framework detects two individual swipe gestures, one on each device. If the swipes both end near a device edge, they are candidates for a multi-device pinch. The coordinates and timestamp of the swipes are sent to the master device, where swipes are collected. If two such events occur roughly at the same time they are considered to be a cross-device pinch. Based on the assumption that this gesture forms a straight line, the position of the finger on both devices is transformed into a relative position of devices (see Figure 13).

Particular care has to be taken when devices are rotated or have different display densities. A more detailed explanation of how to cope with these differences can be found in the project report that preceded this thesis (Schreiner 2015). Eventually, Connichiwa constructs a global coordinate system where all pinched devices are located and these differences are compensated for (see Figure 14). Developers can easily access each device's location and size in the global coordinate system through the Connichiwa JavaScript API. Further, the API allows for the construction of `CWLocation` objects, where an arbitrary point, size or rectangle in the current device's local coordinate system can be converted to global coordinates and vice versa. This allows an application to send a `CWLocation` to another device without the physical representation changing. Effectively, this gives applications a bridge between the digital world and the physical world, making the physical size of digital objects available to the appli-

cation. An example where this ability is used to create physically homogenous content can be seen in *Dynamic Viewport with Image Tiling* (Section 4.2.1, p. 56), where a high-resolution image is shown across a heterogenous device ecology. Although these devices have different positions, sizes, rotations and pixel densities, the orientation and physical size of objects in the picture stays the same across all devices.

Please note that the pinching gesture is one of a variety of possible cross-device gestures. In a future evaluation, it should be tested against alternatives to determine the appropriate gesture for stitching devices.

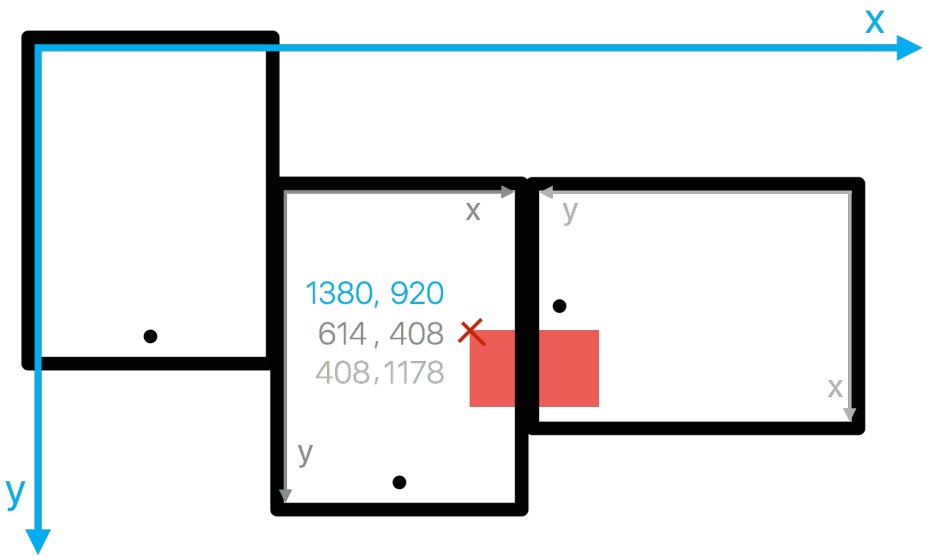

Figure 14: The global coordinate system built by Connichiwa when using cross-device gestures. The location of the "X" has different coordinates on the middle and right device, but the same global coordinate on both devices. Hence, coordinates can be sent between devices without their physical location changing.

The proposed position detection system has some limitations: It is static, and therefore changes in the position of devices are not detected automatically. Connichiwa implements a feature that automatically unstitches a device when it is moved, but the new position of a device is only detected when the device is re-stitched. Furthermore, the accuracy of the described method is varying and depends on how well the user performs the gesture. Discrepancy between the actual position and the calculated position are possible. Lastly, the exact pixel density of a device might not necessarily be available, requiring an estimation of this value. This can also lead to inaccuracies, but has been proven to be neglectable in our tests, in particular on mobile devices. On screens with very uncommon pixel densities, such as TVs with 1080p resolution, these inaccuracies can increase. If future web clients offer a way of retrieving a display's pixel density, as proposed in *Providing Hardware Information* (Section 3.4.7, p. 36), this problem would be solved.

3.5.2 Data Synchronisation

As determined in Chapter 2, data synchronisation is an important challenge when developing cross-device applications. Connichiwa tackled synchronisation using the data models described in *HTML* (Section 3.4.8, p. 38). Extending on this, Connichiwa can also synchronise arbitrary data between devices. Developers can store data into a key-value datastore, and the framework will automatically synchronise the data to other devices and inform them of the changes. Developers can access the stored values. An example of how to use the datastore can be seen in Listing 4. This simple API allows developers to easily synchronise data with a single method call, without caring about synchronisation mechanisms, communication and other technical difficulties.

Furthermore, while Connichiwa's current conflict resolution algorithm is very simple – the last value written will always win –, more elaborated mechanisms can be integrated without changes to the API, for example an automatic merging of objects.

```
$("#userTextfield").on("change", function() {
  CWDatastore.set("userName", $(this).text());
});
```

Listing 4: Example JavaScript code of how to synchronise data between devices in Connichiwa. Data can be stored in a key-value datastore and will automatically synchronise to all devices. The example uses the jQuery library and assumes an element with ID `userTextfield` to be an HTML text input.

3.5.3 Extended Sensor Access

Nowadays, web clients provide web applications with access to different device sensors. This has seen multiple extensions over the last years, such as access to the device orientation, the accelerometer or the gyroscope. Recently, standards have been proposed for access to the proximity sensor or battery status. This ability can be extended using Connichiwa, e.g. with secure access to biometric sensors or the device's current volume. Due to the limited time in development, only some of the possible sensors were implemented, but Connichiwa's architecture allows for an easy extension with new sensors. Currently, Connichiwa does grant access to the proximity sensor as well as accelerometer and gyroscope in a unified manner on all devices.

3.6 Conclusion

In conclusion, we formulated seven requirements that are necessary to establish ad hoc cross-device interaction in everyday life. For the development of a prototype framework, we first compared possible basic technologies. We decided that web technologies seem the most promising candidate due to their extreme availability, large standardisation, and easy dissemination. Nonetheless, we identified remaining problems where web technologies do not meet our requirements. We then developed Connichiwa, a prototypical framework that extends current web standards with cross-device centric features. The goal of this framework was to solve the identified problems and fulfil the requirements as best as possible. The framework consists of a native application representing a future web client and a JavaScript framework that communicates with the native application and offers cross-device functionality to web developers. Connichiwa extends current web technologies in the following ways:

- Connichiwa enables local web applications that do not require an external web server. The server runs directly on the device. This enables web applications over public and ad hoc networks without an internet connection or remote hardware (R2).
- Connichiwa uses Bluetooth to make applications aware of nearby devices. Devices can be detected and invited over Bluetooth, allowing multiple devices to join into a single application spontaneously and seamlessly (R1, R6). This approach lowers the threshold of joining cross-device applications, in particular in multi-user settings, and removes the current requirement for shared logins and manual definition of device ecologies (R1, R3).
- Connichiwa approximates the distance between devices constantly using the received Bluetooth signal strength (R4).
- Connichiwa sends events to the web application for nearby detected or lost devices as well as devices that connect or disconnect from the application. The application is informed when the approximated distance of a nearby device changes. These events allow applications to adjust the current device ecology (R3, R7).
- Connichiwa enables fast, reliable, and standardised peer-to-peer communication using an on-device WebSocket server (R6, R2) and gives developers a simple API to send messages to other devices or broadcast messages to all devices (R7).
- Connichiwa encapsulates information about remote devices into objects. It allows developers to retrieve information such as display density, physical screen size, and canonical name (R7). This allows applications to adjust device roles to the current device ecology (R3). Connichiwa further encapsulates the ability to perform actions on the device objects, such as loading JavaScript or CSS files, updating the content on the other device, send messages, and more (R6, R7).

- Connichiwa extends the HTML standard with a cross-device centric templating mechanism. It introduces a Model-View-Controller pattern to web development. Views are reusable across multiple devices, and developers manipulate the model's data instead of the views. Connichiwa takes care of synchronising the models and updating all views to reflect changes in the models. This approach ensures synchronisation and consistency between devices without manual effort from developers (R5, R7).
- Connichiwa introduces several helpful plugins: Static position detection using a cross-device pinching gesture, simple data synchronisation and extended sensor access (R7, R4).

While Connichiwa aims at solving the given requirements, it comes with its own set of limitations. The sensors of devices are limited and do not allow for an live-tracking of devices. People or non-digital objects cannot be tracked at all. Security concepts are not part of this prototype and communication is entirely non-encrypted. Ad hoc networks can currently not be created and joined programmatically, a fact that hinders the seamlessness of Connichiwa. We will detail the remaining areas of work and possible solutions in Chapter 6. Nonetheless, we conclude that Connichiwa offers solutions to several of the problems we identified. In Chapter 4, we will perform a two-fold evaluation of the framework to see if this claim holds true in the real world. We will then derive design guidelines for future web extensions in Chapter 5.

CHAPTER 4

EVALUATION

This chapter will describe how Connichiwa was evaluated as well as discuss the results of the evaluation. A two-fold evaluation of Connichiwa was performed:

A **study with developers** where we observed students working with Connichiwa over the course of multiple weeks. The students had the goal to create a cross-device application and where given Connichiwa as well as some initial instructions. We then had weekly meetings with the students where we tried to determine their mental model of the framework. The students also described their work, the problems they had and what they particularly liked about the framework. The result of this study is an evaluation of the API and learning curve of Connichiwa as well as an insight into how developers react to the possibilities provided by the framework.

A **technical evaluation** where we implemented six example applications to test Connichiwa's support for the creation of cross-device applications and novel interaction techniques. Further, we tested parts of the applications in the wild, where people joined an ad hoc session without preparation. Lastly, we tested Connichiwa's performance with demanding applications and with large numbers of connected devices.

4.1 Developer Study

The first evaluation was conducted with computer science students over the course of a semester. Students had the task to implement a cross-device application using Connichiwa. Weekly meetings were conducted to understand the participants' mental model of the framework and talk about their progress, problems and benefits.

4.1.1 Goals

This study was conducted with multiple goals in mind:

- To test the JavaScript API functionality and ease-of-use for developers.
- To find concepts of the API that developers have trouble understanding or that are particularly easy to grasp.
- To see if developers understand Connichiwa, and how the understanding develops over the course of multiple weeks.

4.1.2 Theoretical Background: API Evaluation

This study mainly aimed at evaluating the Connichiwa JavaScript API. Zibran et al. state there is a difference between the usefulness of an API (if it can perform a task) and the usability of an API (if a developer is able to perform the task easily) (Zibran

et al. 2011). Evaluating the latter is particularly difficult, and there have been several API evaluation methods available that repurpose existing HCI evaluation methods, including different kinds of interviews (Piccioni et al. 2013), measurements of completion time (Ellis et al. 2007), code walkthroughs (O'Callaghan 2010), or thinking aloud (Duala-Ekoko and Robillard 2012). An overview of traditional HCI evaluation methods for API evaluation is also given in (Beaton et al. 2008). Such methods were originally designed to evaluate GUIs, though, and do not necessarily fit the evaluation of an API for several reasons:

- APIs are often highly complex and large APIs can have thousands of classes.
- Only limited information can be gathered using methods such as observation, as a lot of the interaction between developer and API happens in the developer's mind as he builds an understanding and a mental model of the API.
- Various small factors can impact the understanding of an API, which are often hard to grasp. Examples are good documentation, naming of methods, or consistency within the API and with other APIs.
- Learning effects, sometimes over weeks, are a large part of an API: Even an API that seems difficult at first can become easy-to-use when a developer "got it". On the other hand, an API that seems easy at first can prove to be too restricting when used for a prolonged period of time.
- The exact definition of a "good" API is not entirely clear. Abstract definitions such as a low threshold and high ceiling by Myers et al. can be found in literature (Myers et al. 2000), but are often not specific enough to be used in a thorough evaluation.

Note that (Zibran et al. 2011) thoroughly lists factors influencing an API. For the named reasons, we followed the idea of (Gerken et al. 2011). Here, Gerken et al. conducted weekly meetings with groups over the course of several weeks. Each week, the groups continued work on a concept map representing their mental model of the framework. The groups were asked to highlight parts of the framework they found easy to understand or difficult to use. Based on this approach, we developed our own study design for the evaluation of the Connichiwa API.

4.1.3 Study Design

The study was designed as an observational study with three groups of computer science students over the course of a semester. Unfortunately, due to low participation and scheduling issues, two out of three groups consisted of only one student each, but the rest of this document will nonetheless refer to them as "groups" for anonymity reasons. The third group was made up of three students, making a total of five developers that participated.

In the first half of the semester, the students received a general theoretical introduction into multiple topics, including cross-device interaction, as part of a university course. In the second half, they were introduced to the Connichiwa framework with a brief overview of the architecture, short code examples, and a video of some of the example applications described in *Technical Evaluation* (Section 4.2, p. 56).

Each student group then received a task expressed as a target domain. Their overall goal was to thoroughly design a cross-device solution using sketching techniques. The solution was supposed to enhance the given domain using cross-device interaction. The students were then asked to implement a basic prototype that illustrates the core functionality of their concept using Connichiwa. The target domains were:

- **Hybrid Sketching:** In Hybrid Sketching, photographs are combined with effects and hand-drawings, for example for sketching in early design phases to illustrate ideas.
- **Storyboarding:** Storyboards are a series of sketches or images that tell a story, and often used to easily explain a workflow, use case or an idea embedded in a bigger context.
- **Photo Tracing:** Photo Tracing is a technique to quickly sketch real-world objects or people. A photo of the real world is traced digitally to produce the sketch.

The students worked on sketching and developing the design of their solution for multiple weeks. After that, shortly before the beginning of the implementation phase, the students received an introduction into the evaluation. Each participant answered an introductory questionnaire that assessed the participants general experience in programming and using programming APIs (see *Introductory Questionnaire* (Appendix C.1, p. 83)). Furthermore, Connichiwa's public API documentation was shown to participants and they were told to consult the documentation first before asking questions about the workings of the framework.

During the implementation phase, we conducted weekly meetings with each group. A total of seven weekly meetings were conducted, but students did not necessarily work on the project between each individual meeting. At each meeting, the students were asked to continue to work on a concept map of their mental model of both the application they created as well as the Connichiwa framework itself. They received a short questionnaire (see *Weekly Questionnaire* (Appendix C.2, p. 85)). We further engaged with the students in semi-structured interviews where they could ask questions and tell us about aspects they liked or had problems with during the past week. The iterative nature of these meetings helped us to assess the learning curve and changes in the understanding and usage of Connichiwa.

After the last meeting the students presented their prototype in a short presentation and demo. Further, a final semi-structured interview was conducted were students could summarise their experiences with Connichiwa and talk about general pros and cons of the framework from their perspective.

4.1.4 Participants

Five students of computer science participated in this study, three males and two females. The average age was 22.8 years (SD = 1.01). Initially, we asked the students about their expertise in programming on a scale of 1 (beginner) to 5 (expert). The average answer was 4 (SD = 1.22). Similarly, the participants were asked about their expertise in *web* programming with the average answer being 2.8 (SD = 1.79). Answers spanned the entire range from 1 to 5 for the latter question. Therefore, participants were good programmers but expertise in web programming was partly limited. Two participants stated they worked with Connichiwa before, but only played around with the framework after the introductory session we gave them. None of the participants did prolonged work with Connichiwa or had deeper knowledge of the framework.

4.1.5 Applications

This section will briefly describe the cross-device applications the students created in the course of the semester and presented in the final demo session. This will give the reader an idea of the features the participants implemented and the scope of the applications.

Hybrid Sketching

This application allows users to take photos with their mobile devices and then combine one or more image files with hand drawings. A device with stylus input can be used to perform the drawing, and touch input can be used alternatively when no stylus-enabled device is available. Drawing can be performed on multiple devices and strokes will be synchronised and merged between all devices. This allows users to work on a sketch collaboratively, while each user is working on their own personal device. Following the notion of instrumental interaction (Beaudouin-Lafon 2000), tools, such as color picker or stroke selector, are made available on external devices (e.g., smartphones and tablets). Changing a tool will have an immediate effect on the drawing. This way, users are able to lay out a personal workspace in the way they need it, while still having a full tablet-sized canvas available for drawing.

Storyboarding

Similar to the previous application, this implementation also externalises tool palettes, but with a focus on the area of storyboarding. Here, one devices acts as the overview, where all images of a storyboard are displayed. Selecting an image causes it to be shown on a tablet with stylus input, allowing the user to continue work by drawing or inserting images. Tool palettes, such as selecting the stroke thickness or color, are externalised on nearby devices such as smartphones, giving the user maximum space for the drawing area. Furthermore, this application utilises non-mobile devices as well:

A presentation mode allows the storyboard to be presented on a nearby large screen or wall-sized display. The presentation can be controlled either on the manager device using buttons, or on a smartwatch where the buttons are displayed as well. This allows the presenter to speak freely while controlling the presentation.

Photo Tracing

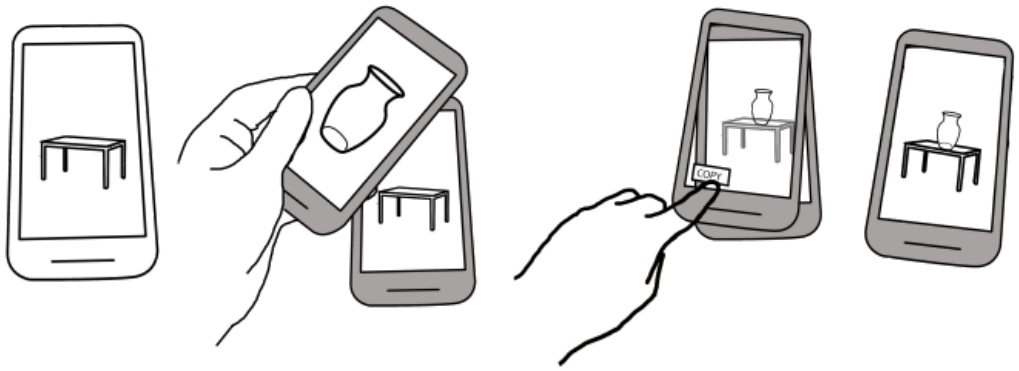

Figure 15: Concept for combining layers. The layers reside on different devices, which can be placed on top of each other and then adjusted to align layers. Source: Created as part of the developer study's university course by participants (parts of the image were cropped).

The group decided to focus on a novel interaction technique during their development and not to implement the entire workflow involved in photo tracing. The result is a concept where devices can be laid on top of each other to combine two layers of drawings (see Figure 15). The application uses a combination of proximity sensor, accelerometer and gyroscope to achieve this behaviour. Through the proximity sensor, the bottom device is detected, as it is obscured by the device lying on top. Synchronisations of the gyroscope values allow to identify the device on top if more than two devices were part of the application. The accelerometer value is used to allow a user to align the two layers as required before merging them. The interaction could be further improved with better algorithms, but due to time limitations only rudimentary algorithms were implemented.

4.1.6 Results

During the weekly meetings we observed that the participants were often reluctant to create a concept map of the framework. This might be due to the way we conducted the study, the presence of the observer, the framework itself, or the fear of making mistakes. Oftentimes, when asked about why they didn't include a certain feature in their concept maps, participants stated it seemed "too obvious" to them to include it,

so this might also stem from the API having a very flat hierarchy. In any case, we unfortunately could not consider the concept maps in a meaningful way in the final results due to them being too incomplete. We therefore focused our analysis on the qualitative results we got from the recordings and interviews during the weekly meetings, which proved to be much more fruitful to get an insight into how the students worked with Connichiwa.

In the initial questionnaire, participants were asked about the APIs they had used so far and what they liked and disliked about them. Mentioned APIs ranged from low-level APIs such as *OpenGL* over high-level APIs such as *Java* and *Android Cloud to Device Messaging (C2DM)* to scripting APIs such as *jQuery* and *Django*. When asked what makes a good API, participants repeatedly stated that examples and a well-written documentation are most important. Next, a meaningful structure was considered important for the general understanding. In turn, missing documentation and a lot of "special rules" are considered to make an API hard to understand and use.

At the beginning of the implementation phase, the groups all had to deal with setting up Connichiwa, getting it to run on their devices and understanding how the API for detecting, connecting and identifying devices works. Most participants reported technical issues when running Connichiwa on their devices, in particular due to the necessity to install Android drivers and getting Android Studio set up and running when using the Android version (which all groups did). Also, for developers that were not very experienced with web development, the general structure of Connichiwa, such as the location of the web application, was part of the questions of the first meeting. "Getting Started"-guides and more detailed examples and tutorials were the main requests in this phase of the project.

While the general concept of master and remote devices was rarely a point of discussion, one group reported problems about how to distinguish devices. One member of the group reported: "I am not sure how to distinguish between two connected devices in order to change the appearance of both". Since the group worked on multiple similar mobile devices it was not clear to them how to assign roles to individual devices. They wished for an easier way of assigning roles to devices besides just doing it based on the order of connecting devices and then manually remembering the role of each device. Not all groups shared this issue, one group even said that "assigning the devices is very easy and structured". The example project that was provided with the framework was generally considered very helpful and a good starting point. One participant noted that "I used the example project as a basic starting point and just added [my content]".

After the initial technical setup was finished, developers were able to achieve a connection between devices and exchange information quickly. Besides the mentioned issue of assigning device roles, participants quickly adapted to the API and were able to use the basic functionality of Connichiwa with only the provided documentation. Developer feedback on the API was mainly positive, and features such as messaging

between devices, templating, unified access to accelerometer and gyroscope, and synchronisation of data were repeatedly called "useful" and "easy-to-use". Besides minor issues with using specific methods, all groups were able to harness these features and create an application that consisted of an ecology of devices, including consistent UIs across devices, transference and synchronisation of drawings (using an HTML5 canvas) and images between devices, and even synchronisation of events such as accelerometer data.

It was continuously reported that debugging, in particular with many devices connected, was very difficult. Each device has to be debugged separately and, if running on a mobile device, remote debugging mechanisms (such as Safari's or Chrome's remote inspector) have to be used. This was considered tedious and time-consuming and participants asked for a more straightforward way to debug on their mobile devices and, if possible, on all their connected devices in a central manner. In one case, Chrome's remote debugger did not work and a lengthy session was required to fix the issue. While it was eventually solved, this illustrates how difficult remote debugging can be.

One group reported issues with the templating API. They were unfamiliar with this kind of architecture on the web and it was not immediately clear to them how to distribute their UI on different devices, and in particular how to decide which template goes to which device and how to assign them. After this was solved during one of the weekly meetings with the help of the study observer, the group had no further issues using data models to synchronise their UI.

A group had issues with communication between devices and the usage of the API methods `.send()` and `.broadcast()`. It was not immediately clear to this group how these methods worked, and how to select the target device. After asking the group about their issues, it became apparent this was due to the documentation of these methods not being clear on some matters. After an adaption of the documentation, the methods were used successfully.

During the project, a problem with transferring large amounts of data very quickly emerged for all three groups. It occurred when sending data such as images between devices or constantly streaming data. The problem is likely related to a bug in the WebSocket server and resulted in random disconnects of devices. During ordinary communication with small text-based data snippets, this did not occur. As a short-term solution, the groups were asked to manually throttle the speed with which they sent data. While this worked, groups clearly considered this solution a workaround and stated that "this problem needs to be fixed for demanding applications to work".

All groups managed to achieve the applications they had designed. In the concluding interview, all groups were very fond of Connichiwa, stating it was "fun to work with"

and “very interesting and useful”. When asked if they would consider the framework for future projects all groups responded positively. When asked about the API, the responses were also generally positive, mentioning that the API was mostly easy to use and understand.

4.1.7 Discussion

Based on the qualitative feedback given by participants, as well as their applications, their progress over the course of the semester, and our observations, we will discuss the study results in regard to the requirements set for Connichiwa:

R1 (Low threshold to join) Joining a running application proved to be easily achievable. A variety of different devices, running different web clients, were able to connect to Connichiwa applications and participate in cross-device interaction. Participants did not report issues with this. The threshold to create a new application, on the other hand, was too high. Installation and setup of Android Studio as well as deployment of the application was time-consuming for all groups, although students with experience in web development had less problems. We think there are two reasons for this threshold: Firstly, “Getting Started”-guides and first steps were not elaborate enough to get developers going without external help. Secondly, the prototypical architecture of Connichiwa complicated deployment. If future web clients support cross-device interaction out-of-the-box, a majority of this threshold will be eliminated. Nonetheless, this issue should be tackled: Expanding Connichiwa to a full web client with an integrated mechanism to deploy and update web applications easily would be of help (see *Connichiwa Browser* (Section 6.1.2, p. 69)).

R2 (Independence of location) All groups used the Android port of Connichiwa, which is not feature complete and does not contain the Bluetooth features of the iOS version yet. Therefore, local communication over Bluetooth could not be tested, but was tested more thoroughly in our second study (see *Technical Evaluation* (Section 4.2, p. 56)). Students worked at home and at the university and had no trouble running applications in those locations. No group used their application outdoors, something that was tested in Section 4.2 as well.

R3 (Cope with volatile device ecologies) Participants used a multitude of devices and tested a number of device configurations using both their private and provided devices. This worked well, except for minor differences in web clients (e.g., a color picker UI is provided by Chrome on Android, but not on iOS browsers). Participants did express that a better mechanism to assign device roles would be helpful. All three applications were designed in a way that the number of participating devices was fixed and stays fixed over the duration of the application use. Device configuration was not supposed to change during application usage, and therefore no definitive statement can be made about this requirement. We test fluid configuration changes in more depth in Section 4.2.

R4 (Support versatile application scenarios) The students designed their applications before knowing about the exact capabilities of Connichiwa. Nonetheless, the framework enabled them to implement their goals. Most participants stated afterwards that they would have liked to implement even more features, but were unable to do so due to the limited time available.

R5 (Support parallel interaction) Parallel interaction was a core component of all three applications. Drawing on multiple devices at the same time, using devices as tool palettes and even aligning devices to synchronise their content was seen in these applications. Participants naturally incorporated parallel interaction without consciously deciding for it.

R6 (Direct data exchange) All applications used direct communication over the on-device WebSocket server for communication. Connecting and communicating with devices, as well as performing actions on remote devices, was rarely a topic of discussion in the weekly meetings. Interestingly, we never discussed the details of communication or the communication protocol with any of the participants during the entire duration of the project and were never asked about these topics by the students. They just used the device objects provided by Connichiwa in a very natural way, calling methods on them to perform actions on remote devices. None of the students had trouble understanding this concept. The mentioned bug when communicating large amounts of data was a problem for the students, and needs to be solved in future versions of Connichiwa.

R7 (Small development effort) During development, students rarely had to adjust their application to specific devices. Students less experienced in web development had to get used to the concept of media queries to adjust the presentation to the devices, but a single code base was used across all devices. This included smartphones, tablets and even smartwatches or wall-sized displays. Furthermore, all basic functionalities of Connichiwa were well understood by developers. The templating mechanism was considered difficult by one group, which might be due to the concept combining HTML and JavaScript more deeply than usual. While none of the participants stated this explicitly, during observations we got the impression that the architecture of Connichiwa was perceived as complex, which was particularly noticeable in the first two to three weeks. For example, it was not always clear to participants which methods are available on only master or remote devices, or both. One participant stated that “it was confusing in the beginning what was available on each remote device (API-wise) in order to send messages”. Simplifying the architecture could allow Connichiwa to get rid of the distinction between master and remote devices entirely, simplifying the API for developers.

4.2 Technical Evaluation

As the second evaluation, we developed six example applications. Each application provides a different focus and examines a different area of cross-device interaction. These example applications were implemented as prototypes to test a) how well cross-device applications can be implemented using Connichiwa and b) if the applications meet our defined requirements. Furthermore, the applications serve as a test for the performance of web applications across multiple devices. Some of these examples were also tested in an in-the-wild situation, i.e. over ad hoc networks. We further implemented a cross-device presentation that was tested with a large number of connected clients in the wild. This test evaluated if the framework can handle an audience with no prior knowledge of the framework in a non-prepared networking situation. The following paragraphs describe the individual applications and what we learned from their development and usage.

4.2.1 Dynamic Viewport with Image Tiling

This image viewer illustrates a) how to combine the screen real estate of devices, b) how to use cross-device gestures to determine the relative position of devices and c) the framework's ability to handle large multimedia data. At first, each device shows a 100 megapixel image and panning the image on any device synchronises the panning to all other devices. Using a cross-device gesture (see *Static Position Detection* (Section 3.5.1, p. 40)), an arbitrary number of devices can be connected and will combine their screen real estate to show the image across all connected devices (see Figure 16). Panning will still synchronise the image, giving users the illusion of a single, large screen that they manipulate. Connichiwa will compensate for rotation and pixel density differences automatically.

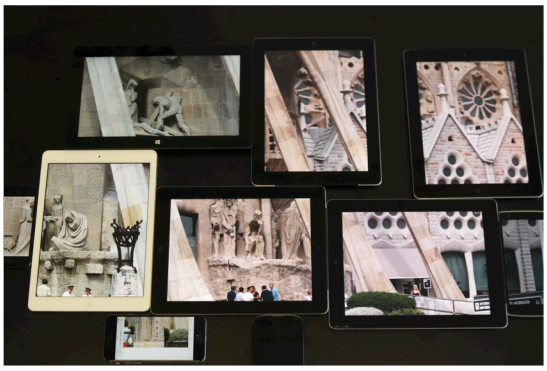

Figure 16: Example Application: A high-resolution image distributed among devices that combine their screen real estate. Panning synchronises across all devices.

While testing the application, it became apparent that modern web clients can handle images of this magnitude without significant problems. We experienced a slight lag when panning on Chrome running on some Windows devices and on older Android devices, but all other devices achieved smooth panning of the image. The cross-device pinching gesture proved to be fast and accurate, although new users had trouble per-

forming the gesture at the beginning. Since this gesture can easily be exchanged, future studies should investigate the performance of the pinching gesture against alternative gestures. When pinched, the illusion of a single large screen worked very well with users and even device edges did not hinder this experience, showing that parallel use of devices (R5) does work and is beneficial to users. Also, we conducted demos of this application where users were invited to join with their own devices, further enlarging the dynamic screen. Users were able to join without issues, and with a large variety of devices, demonstrating that the threshold for joining a Connichiwa application is low enough so users can gather for an ad hoc session, even collaboratively (R1). Some performance issues could be experienced on older Android devices, which was to be expected due to the big advances made in mobile computing in the last couple of years. Newer Android devices worked without problems. All in all, the application worked and performed well on a large variety of different devices, even though the application had not been specifically adapted to one of the devices (R7).

4.2.2 Music Player

A more advanced multi-media example demonstrates the use of sound and external hardware. It also demonstrates how cross-device applications can be used for leisure applications and support multi-room concepts. In this prototype, a web-based music player is shown on one device. ID3 tags are read from a music file, and metadata such as artist, song title and album cover are extracted and displayed. The device allows to control playback. When additional devices are added, a synchronised music visualisation is displayed on each device. Controlling the music on the first device, such as pressing pause, will immediately effect the visualisation. Further, it is possible to redirect the sound output to any connected device and even multiple devices at the same time. This application demonstrates the use of advanced multi-media features (R4) and synchronisation of multi-media content across devices (R6). In general, the synchronisation worked well and delays between music and visualisations were small enough so they were not detectable by users. Only when sound output was set to multiple devices and those devices were lain next to each other, a slight delay was detectable. Future improvements in synchronisations could solve this (i.e. support for synchronised actions across multiple devices). Redirection of sound output allows to make use of advanced audio hardware,

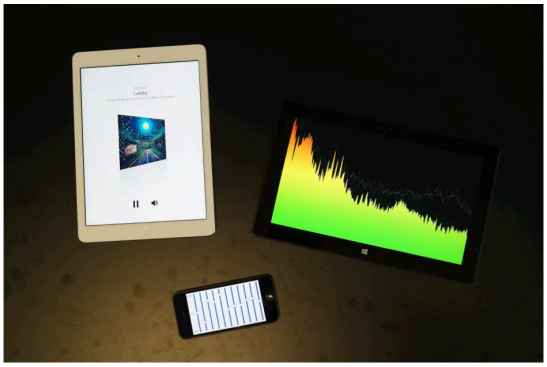

Figure 17: Example Application: A music player (left) with synchronised visualisation (right) and the possibility to redirect sound output to any device. The bottom device does not belong to the application.

such as a sound system attached to one of the devices. In the future, web applications might even support Bluetooth audio protocols directly.

Some problems with multi-media playback on the web did become apparent. For example, automatic playback without user interaction is not always possible on iOS for security reasons. Some devices also had trouble extracting the ID3 information from the music file.

The music player further illustrates the possibilities for new interaction techniques by introducing the clap-to-swap gesture. Using two mobile devices lying on a table, the user can clap the two devices together so that they stand in a 90° angle to the table and their screens touch each other (see Figure 18). This synchronised gesture will be recognised and the roles of both devices will swap – for example the device showing the visualisation and the device showing the music controls will swap their views. At the same time, if any of the two devices were set to output sound, the output will swap to the other device as well.

This concept also demonstrates how cross-device applications can be used for multi-room concepts (R4). Combined with user tracking, i.e. of the user's personal device (Faragher and Harle 2014), music can follow a user through different rooms. A single person can control music that is streamed to multiple rooms in a house. Large-screen displays can show synchronised visualisations that create a mood that fits the music, for example by synchronising the visualisation color across all rooms, e.g. for a house party or even a discotheque.

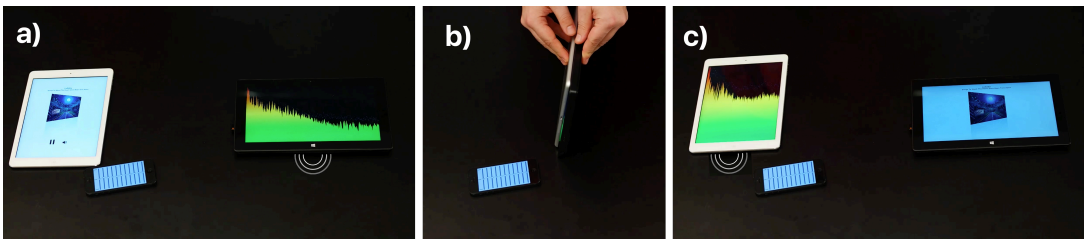

Figure 18: The Clap-To-Swap gesture in the music application. a) The right device shows the visualisation and plays music. b) The devices are clapped synchronously c) Both UI and sound output have swapped devices.

4.2.3 Document Reader

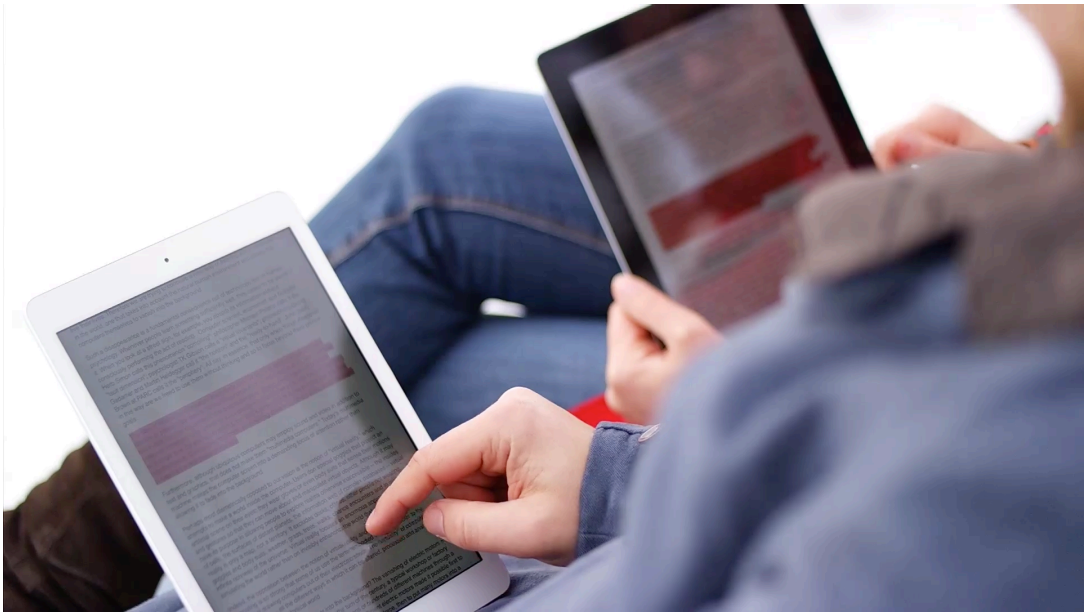

Figure 19: Example Application: A simple document reader where text highlights are synchronised to other devices. The picture was taken in an outdoor test over an ad hoc Bluetooth network.

We created a simple document reader, where text can be highlighted by moving over it with two fingers and highlights are synchronised to another device. While the application itself is simple, we tested it in a park using an ad hoc Bluetooth network created by one of the devices. The connection was stable and performance was reasonable: Highlights appeared on the other devices almost instantaneously. This demonstrates how cross-device applications can be run entirely autonomously without any external requirements, such as a router, server, or internet connection (R2).

4.2.4 Whack-A-Mole

Demonstrating a basic cross-device game, this application is a multi-device adaption of the well-known carnival game. Before the game starts, an arbitrary number of devices can be added to the game. When the game starts, the “mole” (a red circle) appears on a random device and starts shrinking. The user has limited time to tap the device’s screen before the mole has disappeared. If the player taps the screen in time, the mole will move to another device, start shrinking once more and the user has to tap again. If the player fails to tap the screen in time, the game is lost. The number of successful taps are counted as the user’s score. Furthermore, this application is highly adaptive – devices can be removed or added mid-game and will be recognised and incorporated into the game immediately (R3).

The concept demonstrates, in a simple manner, how cross-device interaction can benefit gaming in the future (R4). Furthermore, performance is crucial to ensure a fluid gameplay. We observed that performance was good, but delays were sometimes noticeable. We therefore concluded that for applications that depend on very accurate timing, Connichiwa's performance is not yet sufficient and ways for timed and synchronised execution of actions must be found.

4.2.5 Digital Camera Photography

Illustrating the versatility of Connichiwa when integrating hardware, this application allows the use of advanced digital cameras (we used the Samsung Galaxy NX) to take a photo. The photo is then transferred to one or more device and displayed instantly. The picture can be saved to the devices. This works with digital cameras that run any operating system that features a web client. The application can be run – without changes to the code – on any other camera-capable device, such as smartphones or tablets (R1, R3, R7). Connichiwa applications can therefore integrate cameras with advanced picture taking capabilities (e.g., advanced image sensor, exchangeable lenses).

4.2.6 Cross-Device Presentation

The most advanced prototype built was a presentation software that was used for two real-world presentations. The application shows a presentation on multiple public screens. It supports additional controller devices for moving to the next and previous slide or jump to slides directly. The controller devices also support wireless presenters such as the Logitech R400. Furthermore, presentations are joinable by audience members: A QR code can be displayed that allows viewers to connect to the presentation with their personal devices instantly. If a viewer does not interact with their device, it will automatically stay synchronised with the main presentation. Viewer's can also interact with their device, jumping to any previous slide of the presentation. In this case, they take control of their personal device and the slide will not change until they move back to the current slide of the main presentation.

The tested presentation contained multiple images with up to 4K resolution and a short video. Furthermore, we included the camera photography described in *Digital Camera Photography* (Section 4.2.5): During a presentation, a photo was taken using a digital camera, and the photo was then streamed to all participating devices and displayed.

Both presentations had between 20 and 30 clients connected and were able to cope with this amount of clients. The presentation performance was good, only slight delays when changing slides were noticeable. During testing, it became clear that large videos were not possible with a large amount of clients due to a single device uploading the file to all connected clients. Future improvements in architecture, such as the ability

for decentralised file distribution, could further enhance performance for throughput-demanding applications.

The presentations were tested in the wild over Wi-Fi networks: One in the university using the university's network, one in a hotel using the network provided by the hotel administration. Both networks were able to cope with the demands of the application without problems (R2). Joining via QR codes was quick and worked on all devices present in the audience (R1, R4, R3).

4.2.7 Discussion

In conclusion, the applications worked well, and performance was reasonable. From a developer side, implementations required relatively little code and detection and communication between devices as well as creating and maintaining distributed UIs was easy using the Connichiwa JavaScript API. We discovered that applications that demand very accurate synchronisation of devices still suffered from some problems, and that mechanisms for synchronised actions are required. The applications demonstrate that we achieved the goal of application versatility (R4), as they illustrate a variety of cross-device applications that Connichiwa enables. Nonetheless, these applications are just a small amount of the possible design space enabled.

Applications such as the music player, Whack-a-Mole, or the presentations illustrate that parallel use of devices is possible with Connichiwa (R5, R3, R6). Using multiple devices in concert is not only possible, but APIs for communication and distributed UIs makes it feasible to create such applications without large efforts (R7).

The outdoor test of the document reader as well as the real-world tests of the presentations illustrate the possibilities of Connichiwa for ad hoc usage. Creating ad hoc networks is not seamless as of yet, and limitations in platform APIs requires users to manually create such networks. Nonetheless, running Connichiwa applications over ad hoc networks has been proven possible, and tests in an outdoor park without available Wi-Fi were successful (R2). Furthermore, letting unprepared users with unprepared devices join into our presentations worked very well, and illustrates how lowering the threshold to join applications (R1) is a great help when aiming at ad hoc multi-user applications. This also worked in a public hotel Wi-Fi without preparation of the network.

CHAPTER 5

AD HOC CROSS-DEVICE WEB EXTENSIONS

Based on the evaluation of Connichiwa and our experience with the framework, we conclude by proposing six web extensions for future web clients. Implementation of these web extensions in future web browsers will lay the foundation to eventually enable ad hoc cross-device interaction by everyone, on any device, everywhere. Please note that parts of these concepts were already published in (Schreiner et al. 2015).

5.1 Local Device Detection

A way of detecting and communicating with physically close devices is a basic necessity to enable ad hoc cross-device interaction. In the prototype framework, Bluetooth was used to discover and handshake with nearby devices. This proved to work very well, enabling devices to seamlessly detect and connect to each other. We therefore propose a similar approach to be incorporated into web technologies, and even extend on this: Web clients in the future should scan for and report nearby devices. This can happen over Bluetooth, NFC, or both, whichever technology is available on the device. Web clients should establish a connection over these channels and handshake basic information, such as form factor or input capabilities. These information can help applications in deciding if a device is suited for the current application. Web applications should be able to register for device-related events. If a nearby device is reported, a web application should be able to request a connection to this device. If the request is accepted, the necessary network information are exchanged over Bluetooth or NFC and both devices join into a single application.

For security reasons, users should be informed of incoming connection requests and users should also be able to turn off local communication entirely.

5.2 Local Device Communication

Local communication channels can further be used to transmit information between devices. Web clients can hereby combine multiple technologies to enable a seamless experience for users while giving developers a reliable technology. In Connichiwa, we used on-device web servers to ensure the availability of the application and communication between devices regardless of an internet connection. This approach proved to be successful, and enabled ad hoc applications across multiple devices.

In the future, this could be further improved by incorporating new web technologies: WebRTC can enable peer-to-peer communication and could simplify the WebSocket approach taken by Connichiwa. Service Workers could replace the on-device server. Both technologies would need to be enhanced for multi-device usage, such as Service Workers taking requests from other devices.

This could be further enhanced by supporting internet communication with communication over ad hoc networks. If the current network fails, or devices are not in the same network, web clients can create ad hoc Wi-Fi or Bluetooth networks on the fly to compensate. This should happen seamlessly, without web applications having to care about the current state of the network. Web developers can therefore rely on their cross-device application to continue working, regardless of where users go.

5.3 Extended Device Information

Connichiwa provided extended device information and thereby continued a current trend in web standards. When multiple devices are involved, an application must know the form factor and hardware capabilities of a device to decide its role in the application. Therefore, information such as screen size, canonical name, input modalities, attached hardware (e.g. projectors or printers) and other information should be made available to web applications. Further, our user study showed that mechanisms to assign device roles are required. A feature-based role assignment, similar to that of Weave (Chi and Li 2015), should be implemented to solve this. Of course, care must be taken to only offer non-critical information. Information that can be used to identify or track users must not be transmitted. Users should be able to decide which information to transmit.

5.4 HTML, CSS and JavaScript

In *Web Language Extensions* (Section 3.4.8, p. 37) we described Connichiwa's extensions of the JavaScript and HTML language. In both of our evaluations, these extensions have been invaluable for creating cross-device applications. This included:

- Events about the state of nearby devices (e.g., if they enter or leave the vicinity).
- Encapsulating device information into device objects.
- Encapsulating the ability to perform actions on remote devices into device object.
- A Model-View-Controller approach using HTML templates, where reactive templates were bound to data models that can be manipulated using JavaScript.

These cross-device extensions should become a part of the web standard. Of course, these extensions should be designed in a way that single-device web applications are not influenced by them.

Furthermore, we propose extensions to the CSS language, that unfortunately could not be prototyped in Connichiwa. Currently, media queries can be used to change the

style of a document based on the properties of a device, mainly its resolution. In the future, the styles of a device should be changeable based on the current device ecology, such as the number of devices, the size of the largest or smallest device in the ecology, or the presence or absence of a certain device role. Styles should be able to adapt to the current device's role in the application. Of course, these CSS features would have to be evaluated for their usability and usefulness, but it is our belief that they would further help developers to cope with volatile device ecologies.

5.5 Data Synchronisation

In *Related Work* (Chapter 2, p. 7), we identified the importance of data synchronisation and consistency across devices. It is an integral part of creating distributed UIs, and we therefore propose to add it to the core of the cross-device extensions. The approach taken in the Connichiwa framework proved to work well in our evaluations: Giving developers a key-data store that is automatically synchronised across devices without further effort or caring about the exact details of the synchronisation, worked very well. In the future, web standards could extend to this, enabling smart conflict resolution algorithms and load distribution for synchronisation.

5.6 Extended Sensor Access

Sensors that can be accessed by native applications should also be accessible by web applications, and the same security regulations should apply. This includes proximity sensors, motion data, camera, microphone, biometric sensors, and more. Of course, as it is common in native applications, users should be informed of the sensors and data an application wants to access, and be able to allow or deny permanently. For example, in case of biometric sensors, the application should never receive the actual biometric data.

CHAPTER 6

CONCLUSION

6.1 Future Work

The future work of Connichiwa is two-fold from our perspective: 1) The prototype framework can be further improved and extended to test new web features and to develop and evaluate novel cross-device concepts. 2) We propose a prototypical Connichiwa Browser that is built on the experiences from the framework. The following paragraphs will talk about these topics.

6.1.1 Framework Improvements

Improved Documentation While the documentation of the JavaScript API was generally considered very helpful, our developer study showed that guides for getting started with Connichiwa, or that explain the architecture and general functionality of the framework were lacking. This should be solved in the future to allow developers to quickly understand the framework and get started with the development of applications.

Ad Hoc Network Connichiwa currently suffers from limitations in the APIs of today's mobile platforms. They prevent the creation and joining of ad hoc networks by applications. It is our hope that, in the future, this will change and devices will be able to join into a single network spontaneously, allowing them to exchange information and run web applications over such a network. Currently, an ad hoc network has to be created manually by users (often called "Bluetooth Hotspots"). Fixing this issue would make joining Connichiwa applications fully seamless. For a short-term solution, a Bluetooth relay that forwards both HTTP requests as well as responses could serve a similar functionality, but would be restricted in performance.

Security Connichiwa did not tackle the issue of security, something inevitable for a cross-device technology. Both communication via the HTTP and WebSocket server must be encrypted via SSL to ensure that they are not interceptable. Also, the communication performed via Bluetooth must be encrypted. Furthermore, issues such as secure identification of devices (e.g., personal devices belonging to the same user) must be solved. Besides, a trade-off is to be made between seamlessly connecting devices and security: Users must be asked before another device can connect, but ways of avoiding repeated authorisation must be found.

Evaluate Modern Web Features There are a number of modern web features that Connichiwa could benefit from. Examples are WebRTC, Service Worker[30], or installable web apps[31] [32], which could allow Connichiwa to get even more independent from network state and location. A thorough evaluation and testing of these technologies, and adjustments to make them feasible for cross-device usage would be necessary to integrate them into Connichiwa.

Load Distribution As of now, a single device acts as a master device and distributes the application and all resources to other devices. All communication flows through the WebSocket server running on the master device. This, of course, creates a bottleneck, in particular over slower networks (e.g., Bluetooth networks), a problem that was also noticed when we tested streaming videos to a large number of clients simultaneously. Therefore, methods for load distribution should be explored. For example, remote devices that hold an asset can distribute this asset instead of the master device.

Android Port The iOS application of Connichiwa was ported to Android but the port is currently lacking some of the features of the main version. While the Android version enables running applications over an on-device web server, it is lacking automated detection and connecting of devices over Bluetooth or distance approximation. Implementing these features would be desirable.

Debugging & Authoring As discovered in our evaluation, debugging and authoring can be improved in Connichiwa. This could be done, for example, by the creation of an IDE that embraces the Connichiwa architecture. The IDE could allow developers to add devices and add files (such as views or CSS files) to those devices. The IDE would take the job of hooking up the files so that they are loaded correctly. The IDE could further allow for unified cross-device debugging, giving detailed information about the error, which device it occurred on, what information was communicated between devices shortly before and after the error and more. The IDE could feature device simulation, allowing users to test their cross-device applications without ever touching a real device.

Permanent Storage During the evaluation, it became apparent that the lack of permanent storage is a problem for developers. Cookies and browser storages are available but not sufficient. Ways to quickly store and restore data on multiple devices are required. This could be achieved, for example, by an on-device database that accompanies the HTTP and WebSocket servers. This database could then be accessible through JavaScript. New web technologies, such as Indexed DB[33], could also be used to solve this issue.

[30] Service Worker API — https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API — Accessed December 1, 2015

[31] Web App Manifest - W3C — http://w3c.github.io/manifest/ — Accessed December 1, 2015

[32] Use Cases and Requirements for Installable Web Apps - W3C — http://w3c-webmob.github.io/installable-webapps/ — Accessed December 1, 2015

[33] Indexed Database API - W3C — http://www.w3.org/TR/IndexedDB/ — Accessed December 1, 2015

Synchronised Actions As discovered in our Whack-A-Mole prototype, there is currently a lack of support for applications demanding very accurate execution of synchronised actions. Due to the asynchronous nature and delays in network communication, achieving such exact synchronisation can be difficult and depend on the current network state. A mechanism for such synchronised actions would help developers in creating better applications. Possible solutions might be clock synchronisation between devices or regular ping measurements to anticipate network delays.

Ensure High-Throughput Support During evaluation, a problem with applications that demand a high throughput emerged. This problem must be solved, and applications must be given the freedom to send and receive as many data as the current network state allows over the WebSocket connection.

6.1.2 Connichiwa Browser

It is our belief that a large step in Connichiwa's development will be a simplification of the architecture, deployment and usage of the framework. The Connichiwa web client served our needs for prototyping. In the future, we propose to develop a full web client (the *Connichiwa Browser*) based on the experiences of the Connichiwa framework. This browser would act like any other web client, but allow web applications access to the features and extensions described in this thesis. The browser would have to be developed for all major platforms. Particularly when combined with modern web features such as Service Workers and installable web apps, such a browser could allow web applications to be visited once on a device, and then run locally anytime. Applications could even be distributed through local communication methods to other devices. This approach would allow users to work with Connichiwa applications without any deployment effort and in a manner that they are used to from ordinary web applications.

6.2 Conclusion

This thesis explored the research area of ad hoc cross-device interaction. Since people today are surrounded by a variety of devices that are still mostly unaware of each other, this thesis aimed at answering the question why that is and if this grievance can be solved. In our analysis of the state of the art, we discovered the diversity of this area of research: Researchers and companies have dealt with the topics of device detection, tracking, and communication. They further explored theoretical backgrounds of cross-device interaction, and how to support multi-device software development. An issue we found was that a lot of prototypical systems rely on extensive preparation, something that is rarely accepted by users. And while it is difficult to achieve a seamless interoperation between heterogenous devices – without a priori agreements and

augmentation – our analysis showed that research in this direction has grown in recent years. We suspect this might be due to devices becoming more powerful and gaining more sensors, which makes using unaugmented devices more feasible. Therefore, we concluded that research in this area should be intensified and that a framework that targets comprehensive cross-device support and that uses the possibilities that emerge with modern devices can help to proliferate ad hoc cross-device interaction in everyday life.

From our analysis, we derived seven key points that we believe to be essential to achieve this goal: A low threshold to join applications, an independence of the user's location, the ability to cope with volatile device ecologies, the ability to create versatile applications, support for parallel use of devices as opposed to purely sequential use, the ability for direct data exchange between devices without a mediator, and minimisation of development effort.

We believe that a framework that fulfils these requirements can allow users in everyday situations to participate in cross-device interaction without extensive preparations. After an analysis of suitable basic technologies, it was decided to base such a framework on web technologies due to their large standardisation, extreme availability and easy deployment. We proceeded to build Connichiwa, a web-based cross-device framework. We created a prototype environment that allowed us to extend on current web standards and implemented features to solve the identified problems in cross-device technologies. This included detection of devices using Bluetooth, on-device servers that operate isolated from internet communication, or web language extensions for device communication, synchronisation and consistency.

We then performed a two-fold evaluation of our framework: A developer study where students worked with the framework over seven weeks, and the implementation of six example applications that test the framework's abilities and performance. We concluded that Connichiwa does indeed fulfil most of our initial requirements: It allows a wide variety of devices to come together for ad hoc cross-device interaction while working independent of location. Performance was reasonable and even large numbers of clients were handled well in our tests. Developers are given a tool that makes it feasible to create multi-device applications that are consistent and adaptive. In these areas, Connichiwa shows a future direction for cross-device interaction, and how current problems of interoperability, in particular between heterogenous devices of different ecosystems, can be solved. Nonetheless, the evaluation also highlighted that there are issues that Connichiwa does not tackle, such as security, data availability with distributed systems, and seamless ad hoc networking.

Based on these experiences, we concluded with six possible future web extensions: Local device detection, local device communication, extended device information, multi-device web language extensions, data synchronisation and extended sensor access. These extensions must be accepted as web standards and eventually be implemented by the browser vendors. It is our belief that this can lay the foundation for a novel kind of interaction, where combining our own devices with those of friends or devices found in public becomes an ordinary interaction, just like touching and swiping on our phones has become today.

CHAPTER 7

THANKS

In the end, there are some personal “Thank You”s I would like to express. Because of the personal nature, these will be written in German.

Der mit Abstand größte und wichtigste Dank geht aus tiefstem Herzen an meine Eltern! Ohne die immerwährende Liebe und Unterstützung in allen Bereichen meines Lebens wäre mein Studium, diese Arbeit und so vieles Andere nie zustande gekommen. Auch in schwierigen Phasen hatten sie immer Unterstützung und Verständnis für mich. Danke genügt hier nicht, aber dennoch: Danke!

Vielen Dank auch an alle Freunde, die mich während dieser Zeit mit Rat, Aufheiterung und Ablenkung begleitet haben. Vor allem auch dafür, dass sie in Stresszeiten Verständnis für mich aufgebracht haben, was sicher nicht immer einfach ist und ein gutes Maß an Gelassenheit erfordert.

Des weiteren auch Dank an Roman Rädle, meinem Betreuer während der Entwicklung dieser Arbeit. Trotz vielen weiteren Verpflichtungen und einem engen Zeitplan fand er sooft es ihm möglich war Zeit, Unterstützung, fachlichen Rat und gute Anstösse, was ich sehr zu schätzen weiß.

Zusätzlich gilt mein Dank allen weiteren Mitarbeitern des HCI-Lehrstuhls an der Universität Konstanz, die mir in vielen unterschiedlichsten Formen bei der Entwicklung dieser Arbeit geholfen haben. Im speziellen an Prof. Reiterer für die Möglichkeit diese Arbeit an seinem Lehrstuhl zu entwickeln.

Mein Dank auch allen, die ihre Meinung zu meiner Arbeit kundgetan haben, und sie damit immer nur besser gemacht haben. Insbesondere dabei natürlich an die Leute, die ihre Freizeit geopfert haben um diese Arbeit zu lesen und zu verbessern. Ich nahm mir all euer Feedback zu Herzen und es genau unter die Lupe. Ohne diese Anregungen und Kritik hätte diese Arbeit bei weitem nicht die Qualität erreicht, die sie nun (hoffentlich) hat.

Als letztes noch Dank an die Teilnehmer der Entwicklerstudie im Kurs Blended Interaction an der Universität Konstanz, die einen wertvollen Beitrag zu dieser Arbeit geleistet haben.

Danke!

REFERENCES

Ballendat, T., Marquardt, N., and Greenberg, S. (2010). Proxemic interaction: Designing for a proximity and orientation-aware environment. In *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '10, pages 121–130, New York, NY, USA. ACM.

Beaton, J. K., Myers, B. A., Stylos, J., Jeong, S. Y. S., and Xie, Y. C. (2008). Usability evaluation for enterprise soa apis. In *Proceedings of the 2Nd International Workshop on Systems Development in SOA Environments*, SDSOA '08, pages 29–34, New York, NY, USA. ACM.

Beaudouin-Lafon, M. (2000). Instrumental interaction: An interaction model for designing post-wimp user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 446–453, New York, NY, USA. ACM.

Chi, P.-Y. P. and Li, Y. (2015). Weave: Scripting cross-device wearable interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3923–3932, New York, NY, USA. ACM.

Dearman, D., Guy, R., and Truong, K. (2012). Determining the orientation of proximate mobile devices using their back facing camera. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2231–2234, New York, NY, USA. ACM.

Duala-Ekoko, E. and Robillard, M. P. (2012). Asking and answering questions about unfamiliar apis: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 266–276, Piscataway, NJ, USA. IEEE Press.

Edwards, W. K., Newman, M. W., Sedivy, J. Z., and Smith, T. F. (2009). Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. *ACM Trans. Comput.-Hum. Interact.*, 16(1):3:1–3:44.

Ellis, B., Stylos, J., and Myers, B. (2007). The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA. IEEE Computer Society.

Faragher, R. and Harle, R. (2014). An analysis of the accuracy of bluetooth low energy for indoor positioning applications. In *Proceedings of the 27th International Technical Meeting of The Satellite Division of the Institute of Navigation*, ION GNSS+ '14, pages 201–210.

Fisher, E. R., Badam, S. K., and Elmqvist, N. (2014). Designing peer-to-peer distributed user interfaces: Case studies on building distributed applications. *International Journal of Human-Computer Studies*, 72(1):100 – 110.

Gerken, J., Jetter, H.-C., Zöllner, M., Mader, M., and Reiterer, H. (2011). The concept maps method as a tool to evaluate the usability of apis. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 3373–3382, New York, NY, USA. ACM.

Goel, M., Lee, B., Islam Aumi, M. T., Patel, S., Borriello, G., Hibino, S., and Begole, B. (2014). Surfacelink: Using inertial and acoustic sensing to enable multi-device interaction on a surface. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 1387–1396, New York, NY, USA. ACM.

Hamilton, P. and Wigdor, D. J. (2014). Conductor: Enabling and understanding cross-device interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2773–2782, New York, NY, USA. ACM.

Hinckley, K. (2003). Synchronous gestures for multiple persons and computers. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*, UIST '03, pages 149–158, New York, NY, USA. ACM.

Hinckley, K., Ramos, G., Guimbretiere, F., Baudisch, P., and Smith, M. (2004). Stitching: Pen gestures that span multiple displays. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 23–31, New York, NY, USA. ACM.

Holmquist, L. E., Mattern, F., Schiele, B., Alahuhta, P., Beigl, M., and Gellersen, H.-W. (2001). Smart-its friends: A technique for users to easily establish connections between smart artefacts. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp '01, pages 116–122, London, UK, UK. Springer-Verlag.

Huang, D.-Y., Lin, C.-P., Hung, Y.-P., Chang, T.-W., Yu, N.-H., Tsai, M.-L., and Chen, M. Y. (2012). Magmobile: Enhancing social interactions with rapid view-stitching games of mobile devices. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, MUM '12, pages 61:1–61:4, New York, NY, USA. ACM.

Jokela, T., Ojala, J., and Olsson, T. (2015). A diary study on combining multiple information devices in everyday activities and tasks. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3903–3912, New York, NY, USA. ACM.

Lucero, A., Holopainen, J., and Jokela, T. (2011). Pass-them-around: Collaborative use of mobile phones for photo sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1787–1796, New York, NY, USA. ACM.

Maekawa, T., Uemukai, T., Hara, T., and Nishio, S. (2004). A java-based information browsing system in a remote display environment. In *Proceedings of the IEEE International Conference on E-Commerce Technology*, CEC '04, pages 342–346, Washington, DC, USA. IEEE Computer Society.

Marquardt, N., Diaz-Marino, R., Boring, S., and Greenberg, S. (2011). The proximity toolkit: Prototyping proxemic interactions in ubiquitous computing ecologies. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 315–326, New York, NY, USA. ACM.

Merrill, D., Kalanithi, J., and Maes, P. (2007). Siftables: Towards sensor network user interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, TEI '07, pages 75–78, New York, NY, USA. ACM.

Myers, B., Hudson, S. E., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28.

Nebeling, M., Mintsi, T., Husmann, M., and Norrie, M. (2014). Interactive development of cross-device user interfaces. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2793–2802, New York, NY, USA. ACM.

O'Callaghan, P. (2010). The api walkthrough method: A lightweight method for getting early feedback about an api. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 5:1–5:6, New York, NY, USA. ACM.

Ohta, T. and Tanaka, J. (2012). Pinch: An interface that relates applications on multiple touch-screen by 'pinching' gesture. In *Proceedings of the 9th International Conference on Advances in Computer Entertainment*, ACE'12, pages 320–335, Berlin, Heidelberg. Springer-Verlag.

Piccioni, M., Furia, C., and Meyer, B. (2013). An empirical study of api usability. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 5–14.

Rädle, R., Jetter, H.-C., Marquardt, N., Reiterer, H., and Rogers, Y. (2014). HuddleLamp: Spatially-Aware Mobile Displays for Ad-hoc Around-the-Table Collaboration. In *In Proc of ITS '14*, ITS '14, pages 45–54, New York, NY, USA. ACM.

Santosa, S. and Wigdor, D. (2013). A field study of multi-device workflows in distributed workspaces. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 63–72, New York, NY, USA. ACM.

Schmitz, A., Li, M., Schönefeld, V., and Kobbelt, L. (2010). Ad-hoc multi-displays for mobile interactive applications.

Schreiner, M. (2014). Connichiwa: A framework for local multi-device web applications. Master seminar paper, University of Konstanz.

Schreiner, M. (2015). Connichiwa: A framework for cross-device web applications. Technical report, University of Konstanz.

Schreiner, M., Rädle, R., O'Hara, K., and Reiterer, H. (2015). Deployable cross-device experiences: Proposing additional web standards. Workshop Paper at *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '15 — Workshop "Cross-Surface: Workshop on Interacting with Multi-Device Ecologies in the Wild" (http://cross-surface.io).

Schwarz, J., Klionsky, D., Harrison, C., Dietz, P., and Wilson, A. (2012). Phone as a pixel: Enabling ad-hoc, large-scale displays using mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2235–2238, New York, NY, USA. ACM.

Tandler, P., Prante, T., Müller-Tomfelde, C., Streitz, N., and Steinmetz, R. (2001). Connectables: Dynamic coupling of displays for the flexible creation of shared workspaces. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 11–20, New York, NY, USA. ACM.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, 265(3):66–75.

Weiser, M. (1993). Ubiquitous computing. *Computer*, 26(10):71–72.

Yang, J. and Wigdor, D. (2014). Panelrama: Enabling easy specification of cross-device web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2783–2792, New York, NY, USA. ACM.

Zibran, M., Eishita, F., and Roy, C. (2011). Useful, but usable? factors affecting the usability of apis. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 151–155.

APPENDICES

APPENDIX A

DEFINITIONS

BT Short for Bluetooth, a wireless short-range communications standard[34].

Bluetooth LE / BTLE Short for Bluetooth Low Energy, the current latest iteration of the Bluetooth standard, also known as Bluetooth Smart or Bluetooth 4.0[35].

GUI Short for Graphical User Interface. A GUI is a way of presenting information when interacting with computers. They are based on graphical elements on a screen that can be manipulated using input such as mouse, keyboard, or touch.

GPS Short for Global Positioning System, also NAVSTAR GPS. A system of satellites that allows devices to determine their position with an accuracy of about 5-10 meters[36].

IDE Short for Integrated Development Environment. Describes a software application that supports programming by integrating a source code editor, build tools and oftentimes a debugger.

JSON Short for JavaScript Object Notation, a popular data exchange format[37].

NFC Short for Near Field Communication, a wireless short-range communication technology[38].

SDK Short for Software Development Kit, a set of tools for a specific programming language that allows developing software.

UI Short for User Interface. The visible part of a software that users can interact with. Most modern user interfaces are graphical user interfaces. Mouse-based systems often use the "Windows, Icons, Menus, Pointer" (WIMP) paradigm, while touch-based systems try to find new ways of interactions (post-WIMP interfaces).

Wi-Fi Brand name for wireless technologies following the IEEE 802.11 standard[39]. The currently used iterations of that standard are 802.11n and 802.11ac. Wi-Fi is often used as a synonym for WLAN (Wireless Local Area Network).

[34] Bluetooth Homepage — http://www.bluetooth.com — Accessed December 1, 2015
[35] Bluetooth Smart Homepage — http://www.bluetooth.com/pages/bluetooth-smart.aspx — Accessed December 1, 2015
[36] GPS Homepage — http://www.gps.gov — Accessed December 1, 2015
[37] JSON Homepage — http://json.org — Accessed December 1, 2015
[38] NFC Forum — http://nfc-forum.org — Accessed December 1, 2015
[39] IEEE 802.11 specification — http://standards.ieee.org/getieee802/download/802.11-2012.pdf — Accessed December 1, 2015

APPENDIX B

CONNICHIWA ONLINE RESOURCES

`http://www.connichiwa.info` Connichiwa's main homepage, currently redirecting to Connichiwa's main GitHub repository.

`http://ios.connichiwa.info` Connichiwa's main repository that currently hosts the iOS framework as well as the JavaScript libraries.

`http://android.connichiwa.info` Repository of Connichiwa's Android port, containing the Android Studio project.

`http://mac.connichiwa.info` The client-only implementation of Connichiwa for Mac OS X. Does not contain a server or the JavaScript libraries.

`http://docs.connichiwa.info` The most recent documentation for Connichiwa's JavaScript API.

`http://template.connichiwa.info` A downloadable, fully configured XCode project that can be used to run a Connichiwa application without any configuration effort.

`http://wiki.connichiwa.info` The Connichiwa Wiki on GitHub.

APPENDIX C

DEVELOPER STUDY QUESTIONNAIRES

C.1 Introductory Questionnaire

This is the questionnaire that was handed to developers at the beginning of the evaluation study of Connichiwa. It was handed out in the course *Blended Interaction (BI)*.

Questionnaire BI Project - Start

Age: _____

Gender: □ male □ female

Major: _____

How would you assess your own expertise in programming?

	1	2	3	4	5	
Beginner	□	□	□	□	□	Expert

How would you assess your own expertise in *web* programming (HTML, CSS, JavaScript)?

	1	2	3	4	5	
Beginner	□	□	□	□	□	Expert

Which languages did you program in so far, and for how long? Which language would you say you can program in best?

_____

_____

_____

Are there any programming languages you prefer over others? If so, why?

_____

_____

_____

What programming APIs did you work with so far?

_____

_____

_____

What is important for you when you work with an API? Have you ever been discouraged from an API? If so, which one and why?

_____

_____

_____

Did you work with Connichiwa before this week (excluding the introductory session in BI)?

□ yes □ no

If you did, please describe how/to what extent:

_____

_____

_____

C.2 Weekly Questionnaire

This is the questionnaire that was handed to developers every week during the evaluation study of Connichiwa. It was handed out in the course *Blended Interaction (BI)*.

Questionnaire BI Project - Weekly

Please describe briefly how you have worked with Connichiwa in the past week. This can include feature implementations, bug fixes, UI, other logic, project setup, ...:

If you had problems implementing particular features or understanding particular parts of Connichiwa's API during the last week, please describe:

If you found implementing a particular feature easy or remember certain parts of Connichiwa's API to be easy to use, please describe: