TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Niko Mäkitalo

On Programmable Interactions

Principles, Concepts and Challenges of Co-Located and Social Interplay

Julkaisu 1386 • Publication 1386

Tampere 2016

Tampereen teknillinen yliopisto. Julkaisu 1386
Tampere University of Technology. Publication 1386

Niko Mäkitalo

On Programmable Interactions

Principles, Concepts and Challenges of Co-Located and Social Interplay

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 15th of June 2016, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2016

ISBN 978-952-15-3749-3 (printed)
ISBN 978-952-15-3764-6 (PDF)
ISSN 1459-2045

Abstract

Computing machines and humans interacting has long followed similar principles – A human gives an input command to a machine, which the machine then executes, gives an output, and waits for the next human input. Thus the interactions are user-initiated, requiring constant active participation, and much attention. Despite this, the number of such interactions has kept increasing since computing has now pervaded all areas of human life. Take mobile devices as an example: they are now considered as magic remote controllers that enable interaction with the whole world. Hence, people are now glued to their mobiles, which makes them more detached from their surroundings and other people nearby. Consequently, there is no need nor desire to socialize with other people in close proximity. Presently, the physical world and the cyber world are melting into each other, and new cyber-physical devices are rapidly emerging. This means that an ever-increasing number of computers are awaiting user input.

This wide array of computing devices and heterogeneous networking capabilities have great potential for improving the ways human interactions with computing can work. The problem is that the current ways of implementing software are not well-suited for implementing interactions where multiple co-located people and devices participate. The tools mainly support implementing apps where a sole user interacts with the device, and possibly, remotely with another person. Vendor-neutrality also causes many challenges as some manufacturers only focus on improving interoperability within ecosystems.

This thesis approaches computing with a novel concept of programmable interactions. The idea is to consider the interactions as first class citizens in software development. Instead of focusing on how a human interacts with a machine, the focus is on how the machines in the same space can share resources and jointly interact with each other, serving the humans – the programmable interactions are based on principles that put humans into a central role in the interactions. For developing such interactions, the thesis presents an Action-Oriented Programming model and its runtime environment. Human and social aspects are considered with a concept of companion devices. These companions carry personal profiles about their owners, and represent them for other devices that are nearby. The devices socialize and interact with each other as well as with their owners proactively, meaning that they are also allowed to initiate interactions.

The approaches and concepts that are presented form the basis for developing software where interactions play a key role. These programmable interactions are based on a set of human-centric principles, and the task of enabling them is highly demanding. Therefore, enabling programmable interactions should rather be considered as a continuous process that improves over time. The most crucial challenges have been identified in this thesis together with a view on how the current technology can be used to respond to them.

Keywords: Programming Model, Co-Located Interaction, Internet of Things, Ubiquitous Computing.

Preface

My doctoral thesis on Programmable Interactions peeks into the adjacent possible – a shadow future where interactions between humans and the computing machines of today in their physical surroundings have truly been reinvented. The thesis concludes the research work that I have been passionate to carry out during the years 2011 - 2015 here in TUT. Although I mostly have worked independently, the work would not have been possible without many great people to whom I want to express my gratitude.

First and foremost, I want to thank my supervisor Professor Tommi Mikkonen whose comments and guidance, encouragement and support has helped me throughout this at times so heavy process. You are such a great person, and the best supervisor one could ever have. Thank You Tommi!

I thank Prof. Jukka Manner for acting as opponent in the public defending of this doctoral thesis. I hope our discussion will be engrossing for us as well as for the audience. For the reviewers Prof. Dr. Marc Jansen and Prof. Cristian Bogdan I'm very grateful for their constructive feedback and valuable comments that have helped me improving the thesis.

I truly am grateful for all the colleagues and co-authors I have had the opportunity to work with. A special thank you to Dr. Timo Aaltonen, with whom I have had such a pleasure to work during all these years, and whose great ideas are behind of some of the most central pieces of work of this thesis. A big thank you also to Heikki Peltola with whom I was pleased to work with during my early years as a researcher. I'm grateful for Prof. Kaisa Väänänen for giving me the great opportunity to work in CoSMo project and research group. Thank you to Prof. Väänänen, Dr. Thomas Olsson, and Pradthana Jarusriboonchai. I also want to express my gratitude and thank Prof. Juan M. Murillo and his great team for so warmly welcoming me and openly taking me as part of his team during my research visit to Cáceres. I also want to thank my co-authors and colleagues from Nokia and Aalto University. Thank you Dr. Juha Savolainen, Tapani Leppänen, Prof. Tomi Männistö, Mikko Raatikainen, Dr. Varvana Myllärniemi, and Jari Pääkkö. Thank you Academy of Finland for funding most of my research, mainly conducted in CoSMo (264422) project, but also partly in D2D (283276) and LiquidIoT (295913) projects. I also want to thank DIGILE for Cloud Software programme, and Nokia Foundation for partly funding this research.

At the end, I want to thank my beloved ones. I want to express my gratitude to my parents for all you have done, and for always supporting and encouraging me. Finally, I want to express my gratitude to my beloved Jenni who truly is illuminating my life. Thank you for all your love, support and encouragement – I love you.

Niko Mäkitalo
Tampere, April, 12th, 2016

Contents

List of Included Publications

This thesis is based on the following publications. The publications are presented in the thesis, in which they are referred to by their roman numerals.

I J. Miranda, N. Mäkitalo, J. Garcia-Alonso, J. Berrocal, T. Mikkonen, C. Canal, and J. M. Murillo. From the Internet of Things to the Internet of People, In *IEEE Internet Computing*, Volume 19, Issue 12, pages 40 – 47, March, 2015, IEEE Computer Society.

> The publication presents joint work of two research groups, Spanish and Finnish, and thus has two responsible authors. The two responsible authors are listed in alphabetical order, and the candidate is the other one of these authors. The publication contains a manifesto which was drawn by the responsible authors. The introduced software architecture is consolidated work of these research groups, and the candidate has implemented the other one of these systems.

II N. Mäkitalo. Building and Programming Ubiquitous Social Devices, In *Proceedings of the 12th ACM International Symposium on Mobility Management and Wireless Access (MobiWac'14)*, pages 99 – 108. Montreal, QC, Canada, September 21 – 26, 2014, Association for Computing Machinery (ACM).

> The work introduced in the publication is completely the candidates own work. The introduced system and the example application are solely implemented by the candidate. The candidate did all of the writing by himself.

III N. Mäkitalo and T. Mikkonen. At the Edge of the Cloud: Improving the Coordination of Proactive Social Devices, In *Proceedings of the 23th International Conference on Information Systems Development (ISD2014)*, Varaždin, Croatia, September 2 – 4, 2014, AIS Electronic Library (AISeL).

> The publication introduces and evaluates four different experiments which were all based on the candidate's implementations. The candidate did most of the writing and did coordinate the writing process.

IV N. Mäkitalo, J. Pääkkö, M. Raatikainen, V. Myllärniemi, T. Aaltonen, T. Leppänen, T. Männistö, and T. Mikkonen. Social Devices: Collaborative Co-located Interactions in a Mobile Cloud, In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia (MUM'12)*, pages 10:1 – 10:10. Ulm, Germany, December 4 – 6, 2012, Association for Computing Machinery (ACM).

The candidate had a major role in designing the introduced concept, as well as in implementing the proof of concept system. The candidate also coordinated the writing process.

V N. Mäkitalo, T. Aaltonen, and T. Mikkonen. First Hand Developer Experiences of Social Devices, In *Advances in Service-Oriented and Cloud Computing*, Volume 393 of the series *Communications in Computer and Information Science*, C. Canal, M. Villari (Eds.), pages 233 – 243, 2013, ISBN 978-3-642-45363-2, Springer.

The publication reports a case study to get outside developer input of the programming middleware that was implemented solely by the candidate. For conducting the case study, a team was hired, and the candidate interviewed the team afterwards. The candidate did about half of the writing, and coordinated the writing process.

VI M. Raatikainen, T. Mikkonen, V. Myllärniemi, N. Mäkitalo, T. Männisto, and J. Savolainen. Mobile Content as a Service A Blueprint for a Vendor-Neutral Cloud of Mobile Devices, In *IEEE Software*, Volume 29, Issue 4, pages 28 – 32, July, 2012, IEEE Computer Society.

The publication introduces an approach and a blueprint software architecture that are based on the system reported in the candidate's M.Sc. thesis [86]. The candidate was the lead developer, and thus had a major role in implementing the described system. M. Raatikainen did coordinate the writing process, and thus is listed as a first author. Otherwise, the writing contributions were equal, and the rest of the authors have been listed in alphabetical order.

VII N. Mäkitalo, H. Peltola, J. Salo, and T. Turto. VisualREST: A Content Management System for Cloud Computing Environment, In *Proceedings of 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'11)*, pages 183 – 187. Oulu, Finland, August 30 – September 2, 2011, IEEE Computer Society.

The publication is based on the candidate's M.Sc. thesis [86] work. The candidate was the main developer, and thus had a major role in implementing the system reported in publication. The candidate also did most of the writing and coordinated the writing process.

Chapter 1

Introduction

Interactions are fundamental for operating within the world. They come in many forms, taking place everywhere and on various levels. At the very least, to interact with the world an entity needs to be able to detect events, then interpret and act upon them. Altogether, these give a meaning for the interaction. An interaction can be considered a process that consists of *sensation* – input from the world, *anticipation* – what events are expected from the world, *adaptation* – how to react to unforeseen events, and *action* – output to the world. Whereas these four apply to humans [94], similarities can be found in how computers interact with people as well as other computer. Today people live in digital world, and interactions are increasingly taking place with computing. Irrespectively, however, the interactions between computers come from an entirely different dimension than the interactions between humans, and using the technology requires a lot of interventions from humans. Consequently, interactions within computing take place on various levels that remain separate from the way people are accustomed to interacting.

Programmable interactions approach computing from a different perspective: the idea is that the interactions themselves should play a key role in computing, and they should follow a fresh new set of principles. From the software development perspective, the focus is then on implementing enablers that support the interactions between co-located machines, humans, and the world – as opposed to developing traditional software applications, where a sole user must pick up the device, open an application, and then actively start interacting with it by giving input.

1.1 Motivation

The importance of interactions has been recognized from the beginning of the information age. For instance, J.C.R. Licklider presented his vision of *Man-Computer Symbiosis* in the 1960's, and analyzed some problems of interactions between humans and computers [81]. A few years later, Licklider introduced his idea of an *Intergalactic Computer Network*, which eventually led to the development of the Internet [107], and which now enables billions of people and entities to communicate. On the other hand, in 1991 Mark Weiser presented his well-known vision of *Ubiquitous Computing*, which described ambiguous interactions between humans and their surroundings [142]. According to Weiser, new goals can be focused on, when things disappear so that they can be used without thinking.

With a view to these early visions, at present a lot of talk is around topics such as the *Internet of Things* [7], fostered by heterogeneous and new networking technologies such

as *5G networks* [40]. These concrete examples show how the paradigm of computing is actually changing at the moment. The world is quickly becoming a place full of computer-enabled objects that are interconnected [42, 128]. Already now, people with apps on their mobiles can remotely control different entities such as smart home electronics that operate within separate networks. Although these kinds of interactions may become useful if one has a very specific task that needs to be done, this forces the human users to act as servants to the machines. With ever more interconnected and computer-enabled objects, the situation should be the other way round: Humans should be the centerpieces of these interactions, but not in the sense of being operators. The entities should be enabled to interact with each other, and with their joint behavior serve the humans that are present with them in the same space. Current computing infrastructures, however, do not support or encourage implementing such interactions, which leaves much room for improvement.

Looking at today's app stores for mobile devices, there seems to be an app for nearly every purpose. For instance, Apple's App Store contains 1,5 and Google Play 1,6 million apps at present [122]. Despite these vast numbers, the apps only employ a single device at a time, and only little, if any, attention is paid to human-to-human, or entity-to-entity interactions. In particular, interplay with multiple entities and humans present in the same physical space is completely sorely lacking.

In the last several years, the concept of ecosystems has been proposed for improving interoperability of devices operating within the same hardware/software environment [15]. However, these ecosystems do not enable actual interactions between and among the devices in the sense that the devices would actually be playing or co-operating with the users. Instead, the ecosystems are typically targeted to *single user, multiple device scenarios*, and can lock the user into a single vendor "silo" [128]. For application developers, ecosystem support is also very limited, and the existing support typically enables synchronizing data over cloud services. Vendors of the hardware platforms also tend to protect their ecosystem businesses, and may even set limitations for their platforms that prevent apps from operating in certain ways. Indeed, this does not promote communication and free interaction with other entities.

Web technologies, on the other hand, have long been based on open standards, and hence offer tools for implementing applications in more vendor-neutral ways – in contrast to native apps where one platform's apps cannot be run on other platforms. However, even though the communication is built-in to the Web, the interactions still happen in the same way as with with native apps. Also, the Web browser essentially is an app itself and only offers a sandbox for interactions. For these reasons Web apps suffer from the same and even more limitations than native ones. Despite these limitations, however, Web technologies can still have advantages over native apps [127], and be used for enabling some interactions. Moreover, Web technologies can teach a lot how software should work, and about standardization for enabling vendor-neutral interactions in the future.

Consequently, all this raises the question of whether the current ways of implementing software are feasible for enabling interactions within the modern computing environment? This doctoral thesis approaches the interactions from a programming model perspective. The programming model is designed to take the most out of today's computing environment by utilizing different resources from a diverse set of the computer-enabled devices and heterogeneous networking technologies. Many devices now enable sensations and actions that allow them to interact with humans, their environment and each other. Some of these even go beyond human abilities. The devices, however, need to be programmed to be able to anticipate events and then to react to them. The ability to adapt, on the other

Figure 1.1: Programmable interactions put into context.

hand, can improve over time by observing the users and by learning.

Programmable interactions are based on four fundamental principles according which the interplay between humans and machines should *be social*, *be personalized*, *be proactive*, and *be predictable*. Figure 1.1 depicts some example scenarios about the programmable interactions in different contexts. These novel interactions are built with a model named Action-Oriented Programming which is especially targeted for implementing the inter-

actions between computing machines that are co-located within the same space with humans. Thus social aspects are an important part of these interactions. The computing machines then respect the social relationships between humans while they are interacting, and they may also appear in human-like ways for the co-located people by interacting with modalities that humans are accustomed to interact with each other. A concrete example could be a car navigator and a mobile phone using voice modality to communicate the user that that the two are negotiating about the destination.

For making the interactions personalized, the information available from cyberspace can be utilized in the physical space. Also, digital content is now an essential part of human life, and thus user-generated digital content is important for making the interactions more personal. Such interactions have then a lot of potential for enriching social behavior and aiding people in their everyday activities. Concrete examples of these types of programmable interactions include sharing life events and activities similarly than people now share them in social media, but now the sharing takes place in face-to-face encounters. The devices may for instance help people find other with similar interests in a conference, automatically exchange contact information, and even help in breaking the ice in some situations. Typical use cases include for example social games.

In many ways programmable interactions reconsider the current concept of app – the boundaries have been removed, they are not tied to any specific platform or device, and they are not necessarily used actively by the user. Instead, programmable interactions proactively take place between co-located humans and machines. Hence, these interactions can be considered as some kind of ambient intelligence [108] since they can be used for changing the state of the physical world based on humans' preferences and ongoing activities. A concrete example could be adjusting the atmosphere of the physical environment where certain activity, like a social game, is ongoing.

Since people are not accustomed to this type of novel approach where interactions with technology take place proactively, it naturally will take some time to adapt to the new principles. For this same reason, it is important to pay special attention to the predictability of the interactions, and that the users may trust the system. Moreover, it is also important to make the user feel that they remain in control over the technology, and enable them to adjust the level of proactivity.

1.2 Research Approach

This doctoral thesis consists of seven included publications, and an introductory part that draws these publications together. The following section introduces the research questions of the thesis. Then, the research methods used in the included publications are described.

1.2.1 Research Questions

The main contribution of this thesis is to explain how interactions can be made programmable. These interactions can open unlimited opportunities and enable completely new types of interactions between humans and technology. Conversely, enabling programmable interactions is not an easy task since they are based on a demanding set of principles. Moreover, many limitations exist regarding current hardware and software platforms. These technological limitations, together with the principles of programmable interactions, set challenges for enabling such interactions. For these reasons, enabling

programmable interactions should be considered a continuous process with many problems to solve and challenges to face.

The main research question of this thesis can be formalized as follows:
What are the challenges of enabling programmable interactions?

This main question, however, remains hard to answer as many aspects of programmable interactions are not yet imagined. Hence the thesis aims at identifying as many of these challenges as possible, and then describes the approaches taken to respond to these challenges. In addition to the main research question, three other more specific research questions of the thesis are:

Q1: *What should the programmable interactions be like, and why?*

Q2: *How to harness the various resources of each device?*

Q3: *How to enable proactivity in social interactions?*

The answers to the research questions are summarized in the conclusion of this thesis.

1.2.2 Research Methods

Design science research method is often used in information system research while designing novel artifacts for problems that have not yet been solved, or while designing more efficient artifacts for problems that have already been solved [54]. As ensuring the artifact's utility for the problem is important, the designed artifact must be validated before it has been transferred to practice, or the performance of the implemented artifact evaluated afterwards [143]. The thesis includes three novel concepts reported in Publications [I], [IV] and [VI]. These concepts and their features were developed in an iterative process, and the developed features of these concepts were validated and evaluated in workshops and meetings with professionals before and after implementing them. The principles that form the basis for programmable interactions were designed in collaboration with another research group studying similar concept than the author of this thesis. These principles, reported in Publication [I], are used for answering the research question Q1. Since validating an artifact beforehand can be challenging, other methods have been used for evaluating the artifacts after the concepts have been realized.

Proof of concept is one of the most common methods in information systems research for showing that introduced new or improved concept can be realized in practice [28]. This method was used by the author of this thesis for showing that the designed concepts for enabling programmable interactions can be implemented. The proof of concept implementations helped in identifying the challenges related to programmable interactions, and hence answering the main research question of this doctoral thesis. Publication [III] describes four proof of concept implementations that were build for showing how the proactivity of the social interactions can be improved in practice (Q3). These proof of concept implementations have also helped in answering the research question Q2 regarding the computing and connectivity resources of the different types of devices.

Action research method in information system research is sometimes used together with design science method for evaluating and improving the designed artifact in its real context [69]. Action research is a cyclic and iterative process consisting of five phases as depicted in Figure 1.2. The process starts from diagnosing the problem. Then after

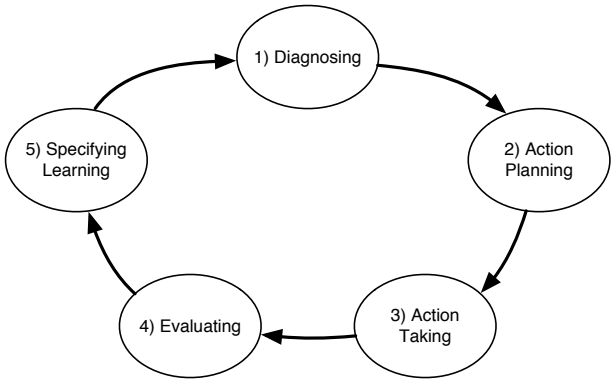


Figure 1.2: Action research is a cyclic and iterative process [124].

planning and taking actions for solving the problem, the results are evaluated, and the lessons learned are specified. One of the characteristics of action research is that the researcher is in some way involved in the research process. Since the programmability is one of the key aspects of programmable interactions, action research method has been used for improving the programmability aspects over the years. The process has been iterative in the sense that several MSc students have used the programming model, middleware, and capability frameworks, described in Publications [IV] and [II], for developing software for various purposes. The author of this thesis has participated in the processes by giving guidance when that has been needed, and by fixing bugs and implementing new features when needed. Working with the students has helped evaluating the programmability of the approach, helped improving the approach further as well as identifying the challenges of enabling the programmable interactions. Publication [II] describes the current state of the programming model.

Case study method is an empirical approach to investigate phenomena in their real-life context, especially when the boundaries between the phenomena and the context are not clear [27, 145]. Case study was used in Publication [V] to get input from outside developers from the concept presented in Publication [IV], and the model for programming the interactions. In contrast to the action research, where working with the students was done with familiar students, the idea of the case study was to arrange one clear use case where the role of the researcher was minimized. In the case study the developer team was given only a short ten-minute presentation about the concept, and then free hands to implement what they themselves wanted. Afterwards the developers were interviewed to get their feedback about the concept, programming model, and especially about the programmability. Interviewing is a commonly used method in case studies for gathering feedback from the participants. The case study helped in identifying the challenges related to enabling programmable interactions, that is the main research question of the thesis. Moreover, the case study also revealed some developer ideas about what the programmable interactions should be like (Q1).

Literature survey method was used to get an overall idea about the most crucial features of content management in distributed and heterogeneous environments. The results of this survey were used for evaluating the concept and the proof of concept system described in Publication [VII]. These results were later used for defining a blue print architecture

presented in Publication [VI]. The survey and the proof of concept implementation both helped studying how the binary resources of various devices can be harnessed for the purposes of programmable interactions (Q2).

1.3 Technical Contributions

Most of the candidate's technical work for enabling programmable interactions has been published under open source and are freely available. The following lists the systems, but their contributions for enabling programmable interactions are described later in this introductory part as well as in some of the publications.

Orchestrator.js (OJS) is the technical counterpart for the theoretical contribution of this thesis, the abstract programming model. Together they form the main contribution of this thesis. OJS enables the programming of interactions for various purposes, and on multiple platforms. It is an open source, JavaScript-based middleware containing a Web-based integrated development environment and a runtime environment. The whole system has been completely implemented by the author of this doctoral thesis. The system is up and running in (`http://orchestratorjs.org`), and the open source code freely available from GitHub (`https://github.com/nikkis/OrchestratorJS`). Moreover, several device-end frameworks have been implemented by the candidate, and these frameworks are listed on the OJS website.

Social Device Platform (SDP) is a proof of concept system developed for the Social Devices concept. SDP was with a joint development of Tampere University of Technology, Aalto University, and Nokia Research Center. The system consists of a set of cloud-based components and a client software. The author of this thesis has implemented some of these components, as well as the client software. The whole open source system is available from GitHub (`http://socialdevices.github.io`).

VisualREST system was developed by a team from Tampere University of Technology, where the author worked as a lead developer from the beginning of the project. The whole VisualREST system is open source, and available from GitHub (`https://github.com/nikkis/VisualREST`).

SocketIO.NetMF is an open source communication library, based on open Socket.IO protocol. It enables relaying events between Microsoft's Micro Framework platform and other Socket.IO enabled entities. The library is available from GitHub (`https://github.com/nikkis/SocketIO.NetMF`).

1.4 Outline of the Thesis

The introductory part of the thesis is organized as follows: Chapter 2 depicts today's computing environment from an interactions perspective, how it's being researched, and how the candidate's work contributes to improving interactions in this environment. Chapter 3 defines the programmable interactions. Chapter 4 describes how the entities can share their resources for enabling more seamless and social interaction. Chapter 5 describes the programming model for building programmable interactions, and its runtime implementation for the modern computing environment. Chapter 6 gives an overview of the included publications. Finally, Chapter 7 concludes this thesis.

Chapter 2

Computing Environment

For decades, first desktop and then mobile computing have controlled the way software is developed, and as a consequence defined how people experience and understand computing. However, during the past few years computing has become more social. This has happened due to two major revolutions. The first one is the appearance of the public Internet, and the second one is the emergence of mobile devices. These two together have enabled completely new ways to be constantly connected and interact with others. This paradigm where all information is available in any location is often referred to as pervasive computing.

Today the computing environment is quickly changing. The number of physical entities capable of participating in computing continues to accelerate. The connectivity is becoming heterogeneous and available in any location. This forms a basis for a new type of *computing environment* where everything is connected, and computing takes place directly everywhere in our surroundings. The interconnected entities should no longer be considered as individual devices, but rather as set of resources that form the basis of new kind of computing. Figure 2.1 depicts the paradigm shift from personal computing to pervasive computing, and further towards this new computing environment.

Figure 2.1: Paradigm shift in computing.

Ever more entities mean ever more interactions between them. This places the interactions in a key role in relation to how our devices behave and appear to us. Previously, computers had mainly been used for supporting remote interactions, where many elements of the interactions remained hidden. Today, however, we are becoming increasingly connected

directly to our surroundings. This addresses special challenges for the interactions, since while we are co-located, the interactions become more perceptible.

This chapter gives an overview of the today's computing environment from the interaction perspective. We first explore the current environment, defining the kinds of interactions it currently enables. Next, the main research areas related to the study of computing environments are introduced. Finally, the chapter introduces the author's visions and approaches that contribute to improving interactions.

2.1 Exploring the Computing Environment

There are many ways to look at the computing environment of today. From the interaction perspective, they can be seen to take place within three different worlds: The *social world*, which reflects society, and consists of human relations and communication, the *cyber world*, which is an abstract space and the world of inter-computer communication, and the *physical world*, where humans, machines, and the environment meet, governed by the laws of physics. Respectively, the following explores the technological interaction enablers and needs from three different perspectives that are in this thesis called as the *Interaction Dimensions of Computing*.

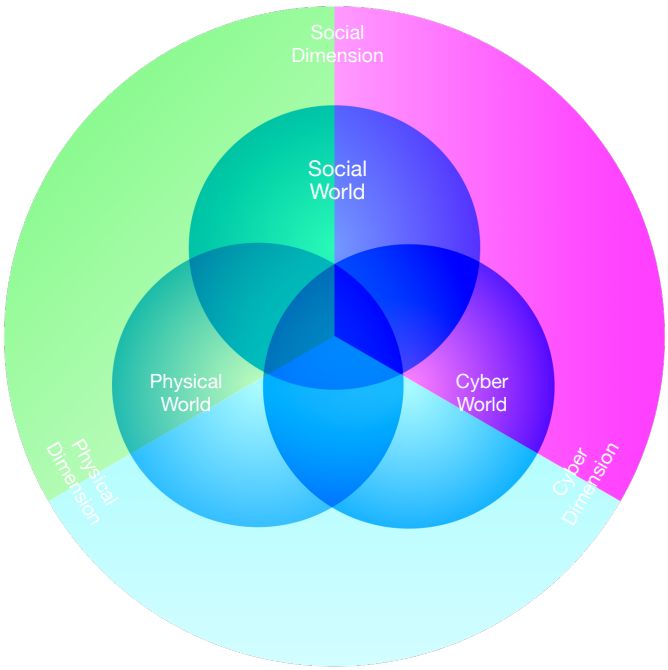


Figure 2.2: The Interaction Dimensions of Computing.

2.1.1 Physical Interaction Dimension

The physical dimension is comprised of the enablers for human-entity interaction, as well as any other interaction that takes place in the physical world. The term "entity" here refers to all tangible objects that have computing capabilities, or some link to computing.

From the human-entity interaction perspective, the appearance of the entity is important and directly related to how humans themselves relate to computing. The following gives an overview to the different types of entities that exist today. However, many other types of entities are emerging all the time, and excellent hardware and software tools enable endless possibilities for building new types of entities. Thus only some of the most common types of entities are mentioned.

Entities of the Physical World

During the last couple of decades, the computing paradigm has changed from desktop to mobile computing as technological development has enabled building smaller and lighter devices that can easily be carried around. Smart phones, tablets, and laptops are now essential for our daily lives. By the end of 2014 the number of mobile phones worldwide was estimated to be almost equal to the number of people [18].

Despite the fact that *mobile computing* has now spread to almost everywhere and for everyone in the world, a new paradigm shift is already taking place: It has been estimated that the number of physical objects connected to Internet will reach 26 billion [42], or even 50 billion [41] by the end of 2020. This trend, named *Internet of Things* [7] or *Cyber-Physical Computing*, aims at connecting physical things from our surroundings to the Internet and other networks. Smart home electronics are one of the most visible of these connected entities, as recently, many end-user products have started to emerge. As a simple and basic, but yet an interesting example, are smart light bulbs that can be controlled directly with other devices. This shows how one of the first electric devices has been upgraded to the modern age and can now participate in computing. Although not all the light bulbs need nor will ever need to be connected to other devices, this example hints at the large the scale of this emerging new paradigm. Other examples of these types of ready-for-use physical computing objects are thermostats, air conditioners, water boilers, different types of lights, relays, switches, security systems, and locks.

Wearable devices are another emerging trend, but from a different track: These highly personal wellness and health devices enable measuring data about the user's physical activity and wellbeing, such as heart rate and calories consumed. This data is then stored on mobile devices or cloud services where some analysis may be performed. Another, more advanced instance of wearable computing is the smart watch that enables usage similar to activity trackers, but also enables similar interactions as current mobile devices. Consequently, smart watches are often used as companion devices for mobile phones. Some other examples of emerging wearable devices are virtual and smart glasses, as well as smart clothing. The former can either aim at showing virtual content over the physical world (augmented reality), or aim at taking the user deeply into virtual worlds. The latter aims at embedding electronics and user interfaces into clothing.

Examples of more social-physical technology include gaming and home entertainment systems. These offer playful and entertaining experiences that can engage multiple people in the same space, and can also support interacting them with sensor-based technology. Moreover, these systems typically offer SDKs for developers. Whereas *computer-supported cooperative work (CSCW)* has previously been based on online tools, tools supporting co-located collaboration have now also started to emerge. As an example, Microsoft's Surface Hub (`https://www.microsoft.com/microsoft-surface-hub`) allows multiple people to interact locally and remotely with the same big board. These are only few examples of recently-emerged entity types, and diversity in the physical dimension is

growing every day. Car entertainment systems, social robots, drones, and self-driving cars are some examples of the next emerging physical entities.

Building New Types of Entities

Multimodal and multisensory integration research studies how several different types of input components should be utilized together for composing new and improved human-computer interaction modalities [76]. Whereas in the past researchers had to build everything from printed circuit boards to writing software from scratch, today many alternatives are available for prototyping purposes [55]. Gadgeteer (`http://www.netmf.com/gadgeteer/`), Arduino (`https://www.arduino.cc`), and Raspberry Pi (`http://raspberrypi.org`) are only some of the most well-known prototyping platforms. Many of these offer actuators, connectivity, and sensor modules that can easily be used for building new types of gadgets. Moreover, excellent IDEs and SDKs now make it very easy to develop software for newly-built entities. For this reason hardware prototyping has also become popular now among amateurs and online open-source communities, and made it much easier to implement new types of entities.

3D printing and scanning technologies are also interesting from the entity building perspective, and is quickly becoming practical for average customers. This means that it becomes very easy for people to print components or even whole objects as discussed by Schmidt *et al.* in [112]. Already today people can either purchase models, or use CAD design software to design enclosures and objects. Currently, however, 3D printing technology does not allow printing components from many materials, and hence it still needs to be used in conjunction with the platforms introduced above for building the entities. The outcome, however, is that building, distributing, and purchasing intelligent objects will most likely face a revolution in the near future.

Physical Appearance

From a human perspective, physical appearance defines how appealing an entity is to use, or how ambient it is in its surroundings. Hence, appearance has a direct effect on how humans interact with entities. For instance, according to Weiser [142] things need to disappear within our surroundings before we are freed to use them without thinking. Thus 3D printing technology can play a role in helping entities blend more seamlessly into their environments.

As the number of entities continues to grow, new types of interfaces are needed for controlling them. Previously, for example, gesture-based controlling has only been possible in fixed locations and research experiments based on Microsoft's Kinect platform (`https://dev.windows.com/en-us/kinect`). However, over the years gesture detection with mobile devices has evolved that can enable new types of human-computer interactions, and also enable ways of interacting with the devices that are more social [132]. As an example, the new Apple TV platform now allows controlling games with one's iPhone as well as with a gesture-detecting remote controller. Moreover, Google's current Project Soli (`https://www.google.com/atap/project-soli/`) studies a chip that can be used for detecting gestures to support new types of interactions with mobile devices. Also of interest is Google's Project Jacquard (`https://www.google.com/atap/project-jacquard/`) which studies embedding interfaces into fabrics.

2.1.2 Cyber Interaction Dimension

The cyber dimension encompasses the enablers for entity-to-entity interaction. While a human and an entity typically interact within the physical world, the entities essentially require established connectivity to enable intercommunication. Thus the connectivity and exploitation of heterogeneous networking technologies play a key role in entity-to-entity interactions. Connectivity on its own, however, does not enable interactions between entities. For this purpose the connected entities offer APIs that other entities then use for remote communication and command. For improving the developer experience, some hardware manufactures also offer SDKs with communication frameworks.

Connectivity

As mobile devices have greatly evolved and new types of devices have emerged, so too has the networking side. Figure 2.3 illustrates the growth of mobile-cellular subscriptions according to ITU [18]. In developing countries the mobile-cellular penetration was estimated to be 90%, and in developed countries 121%. The numbers of mobile-cellular subscriptions in developed countries between 2005-2014 show that the growth has almost stopped. A similar trend can also be seen regarding mobile phone subscriptions in developing countries. Considering these numbers, it can be estimated that a great deal of the world's population already has or will soon get constant Internet connectivity. The number, at any rate, will most likely go up in all countries, since other new entity types can also utilize the mobile networks. This type of connectivity is typically referred as Wide Area Network (WAN) as it enables communication around the world.

Infrastructure-based networks, like the aforementioned mobile cellular networks, enable indirect communication between entities. Today for example LTE networks offer tremendous transfer rates at a reasonable price. Another form of indirect, infrastructure-based networking is Local Area Networks (LAN) that can also offer wireless access points with Wi-Fi technology, and thus are called Wireless Local Area Networks (WLAN). LAN technology allows communication between entities within a limited area, but typically also offers access to WAN. Mobile-cellular and Wi-Fi technologies offer the basic connectivity and backbone for communication. However, considering the vast development in the physical dimension, it becomes essential to consider other communication possibilities as well.

Direct device-to-device (D2D) connectivity today is fortunately well represented, and can be supported by several options. Moreover, these network types are typically infrastructure-free, and work in an ad hoc manner. Some examples of D2D communication are personal area networks (PAN), where the idea is to form network around the user. Some of the most well-known examples are classic Bluetooth (BT), Bluetooth Low Energy (BLE), Near Field Communication (NFC), and Wi-Fi Direct (or Wi-Fi Peer-to-Peer). ZigBee and Z-Wave, on the other hand, are low-energy technologies that are typically used for enabling communication in home automation systems. In addition, wireless sensor networks (WSN) are constantly being developed for various purposes [6, 79, 103]. Recent examples include vehicular networks [71] for use in self-driving cars, and underground networks [5] with applications in agriculture.

Fixed-price mobile broadband connectivity and WLAN connectivity available in places where people spend most of their time have changed the ways people interact. However, device-to-device networking will become increasingly important not only because it enables faster communication, but also because it can make communication more secure. In

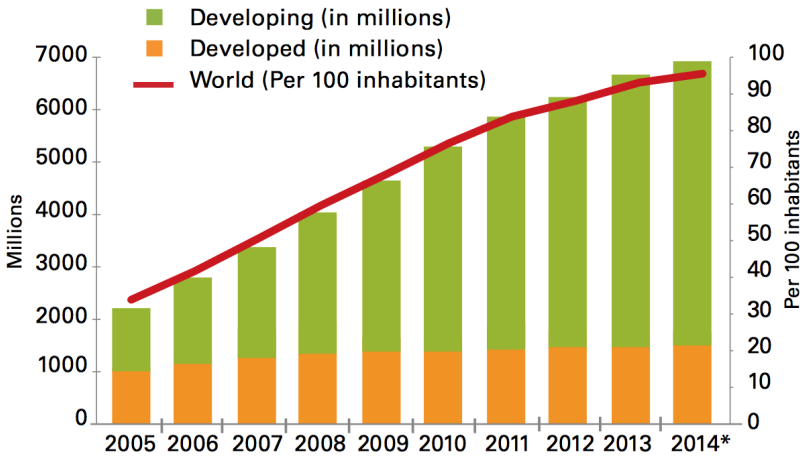


Figure 2.3: Mobile-cellular subscriptions, total and per 100 inhabitants [18].

addition, because the public Internet may sometimes be a hostile place, not all locations offer proper Internet connectivity [111]. According to Benincasa *et al.*, the next generation of mobile applications will operate in highly dynamic environments such as heterogeneous wireless networks [11]. This claim is easy to adopt considering that within the coming decades we will be surrounded so many different types of physical objects that it simply would not make sense to communicate everything through Internet. Conversely, not all the entities even have support for Internet connectivity. In 5G, the network infrastructures may already automatically support utilizing heterogeneous local area networks and simultaneous use of multiple connections.

Seamless Usage of Multiple Devices

Application Programming Interfaces (API), together with the pervasive Internet, have enabled more seamless use of multiple devices. In many ways various APIs are now defining how software is implemented as many software implementations are based on synchronizing data over Internet-based services. Today hardware manufacturers offer their own cloud services for keeping information synchronized across the different entities of individual users. For supporting interoperability between devices from different vendors, manufacturer-independent services have also been introduced. Typically however, these services mainly focus on data interoperability between entities.

Lately, however, device vendors have started to utilize other networking technologies for supporting interoperability between their ecosystem devices. For example, Apple's Continuity technology aims at enabling one to continue the same task when the user switches from one device to another. Thus, when two devices are near each other, a device can suggest opening the same currently open document or website on another device. Although this example is mostly limited to synchronizing data, it still shows how network technologies can interoperate: Bluetooth technology is used for transferring information about proximity, and Wi-Fi is used for transferring the actual data.

Connectivity Supporting New Types of Multi-Device Interactions

Entities with connectivity and different *input* and *output components* can interact with their surroundings. As an example, sensors enable measuring data from their environment and their users. On the other hand, screens, speakers, actuators, etc. enable outputs to the physical space, as well as to their users. While interacting with people and physical space happens with these input and output components in the physical dimension, interactions with other entities typically require connectivity and takes place in the cyber dimension. This also includes relaying the results of processing and input operations from one entity to another. The entities' components (input and output), and their other abilities to store and process data, are called *resources*.

The ability of these physical objects to interact with people and their surroundings enables building new types of interaction capabilities. As an example, touch screens revolutionized the way we interact with many entities today. However, devices are still typically used one entity at a time. When the surroundings become flooded with a wide variety of entities, the number of possibilities for interaction between and among them will be nearly endless. By connecting and harnessing multiple entities they can all act as one [84], or, allow many users to participate in the same interaction. Moreover, as the entities have different resources, they can participate in the interactions in different *roles*, and the interactions can utilize only some of their resources. For example, while playing a game at home, it makes sense to select the big screen TV, and still use the home theater receiver for the audio, unless it has been reserved for other purposes, like playing music at the same time. Harnessing, however, requires clever ways of recommending and configuring entities to their roles in the interactions based on their resources (e.g. [98]).

Smart home and home automation systems are timely examples of more advanced controlling of physical entities. Whereas smart home systems mainly offer *remote interfaces*, home automation systems offer tools for automating the control of these entities based on the user's actions. Smart home systems, however, are not a new topic but have already been studied for years, and for this reason vendor-neutral systems have been developed. Some of these older approaches include for example Wink (`http://www.wink.com/`) and Smart Things (`http://www.smartthings.com/`). The idea in all of these systems is essentially the same: home electronics are connected to a centralized gateway that mediates messages from the input devices (typically a mobile device) to them. Part of the control can be automated with mobile apps that these systems offer.

Recently, some mobile platform vendors have started offering more advanced software development kits (SDKs) with *communication frameworks* that can be used for building more advanced applications that communicate with other entities. Naturally, however, all of the entities need to have a similar framework and support the same protocol in order to communicate with each other. One of the most recent examples is Apple's HomeKit approach (`https://developer.apple.com/homekit/`) which enables the user to control their HomeKit-compatible electronics via Apple ecosystem's devices. For example, iPhone and Apple TV enable commanding the electronics with voice modality by their Siri assistants. Also, Google is developing its Weave protocol for controlling its Brillo and other Internet of Things devices (`https://developers.google.com/brillo/`).

2.1.3 Social Interaction Dimension

The social dimension encompasses the technical enablers for human-to-human social interaction. Traditionally, the interactions have taken place within the physical world.

However, people are social creatures by their nature, and thus from the beginning of the computing era computers have been utilized for communicating with other people [138]. When the public Internet enabled remote communication, it created new *cyberspaces* where new types of interactions and collaboration in real-time were made possible. Moreover, mobile devices have allowed computing to become detached from fixed locations, enabling constant connectivity.

Living in Two Worlds

Ubiquitous Internet connectivity has now totally changed the way people interact. Even the traditional mobile technologies are being replaced by new ones. A survey made by the Finnish Communications Regulatory Authority shows that the usage of traditional SMS messages and phone calls have started to decrease because of the emerging social technologies [134]. Social media, instant messaging, dating services, online collaboration tools, and other types of mobile and Internet applications are basic examples of these *virtual world* technologies.

Constant connectivity, however, has its costs; it has become clear that being constantly connected and online is changing the way we socialize, react, and relate ourselves to our surroundings. For instance, our ability to focus and concentrate has changed, and we have become more detached from our surroundings [14]. Thus, it can be said that in many cases the technology can actually separate people from other people and from their surroundings; it is common to see people gazing their mobiles, possibly communicating with someone remotely rather than face-to-face with people nearby.

As pointed out by the physical dimension, computing devices have typically been highly personal, and used solely by their owners. As with highly personal devices and their usage, digital life has become equally private. As of yet, no major social media service has been designed to support and to be used together with other people nearby. Hence, interactions in the two worlds remain separate, and consequently, it can be said that at present people live and socialize in two different worlds, physical and virtual, simultaneously.

Human Involvement

One of the goals of social technologies is also human involvement in computing. The entities can become social by understanding their owners and other humans more deeply. Whereas this might sound like a future technology, already today mobile phones, networking technology, and Web services gather tons of information about people's behavior. This information is then used for suggesting products or activities, typically for commercial purposes (e.g. advertising). Data mining is typically done by performing data analysis on big data that can be gathered based on network traffic or users' physical locations. This collection of such private data has raised concerns in many people and governments. At the same time though people, voluntarily or through ignorance, are giving up their privacy in exchange for free services.

Despite the fact that these social media services have their flaws and have changed the way we communicate, they can offer improved access to social knowledge [116]. This information can then be utilized in many contexts, and within other dimensions. For example, data about social relationships can offer essential information related to security, like improving trust and privacy. In social media people share their activities with a certain group of people that they trust and with whom are willing to give up their privacy. In many cases the sharing happens among closest friends, but not necessarily with the

entire social network. Considering that users have entrusted a certain group of people, it could be assumed that the same item could be shared among that group in face-to-face situations as well. However, considering the highly personal nature of our current digital life, this may not apply in all the cases as people communicate in different ways in cyberspace compared with physically co-located situations.

Socially Appearing Entities

From a software development perspective, connectivity or computing capabilities alone do not make an object social. It is the program controlling an object that has the greatest potential to make an object appear more lively or smart for the user.

Although the social technologies of today primarily support socializing among humans, the entities are getting more and more autonomous and intelligent. The digital assistants are an example how the entities can already appear in a more social way for the user. The most recent examples of these assistants are Facebook's M, Microsoft's Cortana, Google Now, and Apple's Siri. The latter one works across Apple ecosystem devices, as shown in Figure 2.4. Although some of these assistants can suggest activities for the user based on previous activities or locations, typically the assistants only offer a new kind of interface. Nevertheless, these digital assistants do not yet offer advanced interactions or applications for multiple co-located people and devices.

Figure 2.4: Siri on Apple Watch.

Another example of socially appearing entities is the emerging social robotics technology (e.g. `https://www.aldebaran.com/`). The vast development of artificial intelligence and controlling of embodiments may allow these types of autonomously moving social entities to walk among humans relatively soon. The artificial intelligence research of social robotics looks into how robots could become self-aware. Furthermore, research has been carried out for generating personalities for these robots [35].

From the interaction and social appearance perspective, the physical dimension becomes important. For example, with assistants the feeling of social appearance comes from the voice modality, which is the natural way for people to communicate with each other. However, the flaw of the assistants seems to be that they don't take the user into account fully in a social sense. For example, from a social interaction perspective, the assistants focus on serving only one user at a time, instead of considering other nearby humans or entities.

Ownership

When the paradigm changes and computing becomes more ambient and ubiquitous, we will also need to rethink the concept of ownership. For example, mobile phones and wearable devices are highly personal, but a tablet can be shared among a family. Another example of such device is the new Apple TV that the whole family shares and also uses together. Whereas programming applications for these devices (e.g. for personal mobile phone and shared tablet) may not be that different from a technical perspective, the applications should be totally different from social and user experience perspectives. In the future, more of these types of semi-public and public devices are likely to be seen. Then, from an interaction perspective, this would mean that the people are allowed to use, interact, and borrow resources from the available entities in their surroundings, thus making the computing more social.

2.2 Related Research Areas

There are many research areas around the topics of this doctoral thesis. Some of these research areas overlap, have similar goals, and the differences are minor. Therefore, these areas are hard to distinguish from one another. The following gives an overview of some of these areas, and describes how these areas are related to each other as well as to interactions in general.

Social Computing

In 1994 Schuler gave a definition for social computing: "Social computing describes any type of computing application in which software serves as an intermediary or a focus for social relation" [114]. In 2007 Wang *et al.*, however, extended the definition of social computing as "Computational facilitation of social studies and human social dynamics as well as the design and use of ICT technologies that consider social context" [138]. Thus this definition also contains theoretical underpinnings for supporting social aspects, as depicted in Figure 2.5 from their publication. Moreover, Wang *et al.* claim that the idea of social computing can be tracked back to Vannevar Bush's paper "As We May Think" in the 1940's [138]. In the article Bush introduced a device named memex, a communication and memory device.

Despite these rather broad definitions, however, social computing nowadays has an established meaning, referring to all types of social communication and interaction that is somehow computer mediated [99]. Nowadays, typical examples include social media and instant messaging. Moreover, many *computer-supported cooperative work (CSCW)* applications are often considered as social computing applications as they aim at supporting the communication and interaction between multiple users to achieve a common goal [43]. Typical examples include cloud services like Google Docs and Microsoft Office 365.

In traditional social computing, the support for the physical dimension is weak, as it typically lacks support for face-to-face situations and co-located cooperation. Considering, for example Facebook or Twitter, the most commonly used social applications, they almost completely lack support for physical presence or location. In Facebook, users can check-in to places and tag who else is present in their status updates. Similarly, Twitter allows defining a "context" for a tweet with hashtags. Thus, these social media services currently neglect the physical dimension.

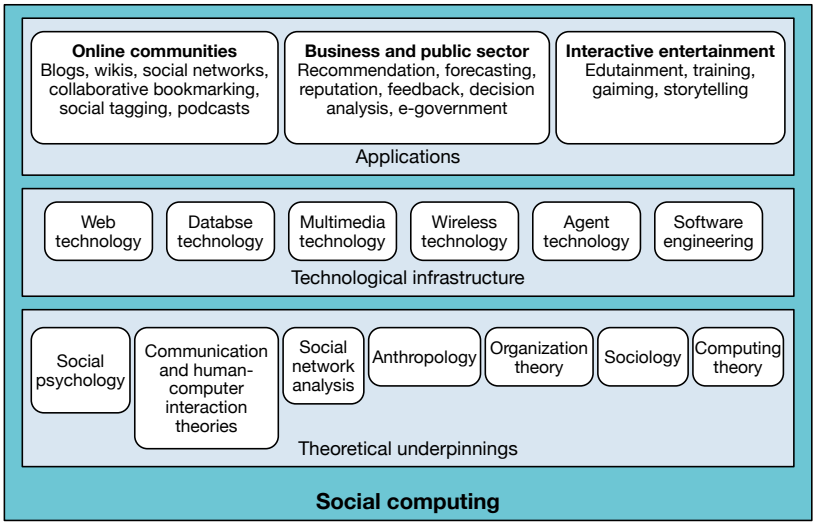


Figure 2.5: Wang *et al.* definition of Social Computing research from 2007 [138].

With more and more interactions taking place right next to users, the physical dimension and proximity are becoming ever more important domains in social computing research. Thus the real strength of social computing, the technical underpinnings, can also offer valuable insights for developing these interactions. Moreover, whereas everything is becoming more easy to program, this leaves more room for interdisciplinary research. Today, some emerging technologies, like social robots for instance, already serve as examples of more physical social computing technology that involves interdisciplinary research.

Ubiquitous Computing

In 1991 Weiser envisioned in his article "The Computer for the 21st Century" [142] how computers will work in the next millennium, and called this paradigm *Ubiquitous Computing (UbiComp)*. The article introduced *pads*, *tabs* and *boards*, which are computers of different sizes and the basic building blocks for ubiquitous computing. The goal in ubiquitous computing is to embed and make the technology disappear into the environment by hiding several, probably hundreds, of these and other types of devices into our surroundings, where they then would seamlessly communicate with each other as well as with humans.

Pervasive Computing is a term that is often used interchangeably with ubiquitous computing, and the difference between the two is subtle. One way of distinguishing them is related to Weiser's vision of things needing to disappear into our surroundings in such a way that we use them without thinking. Thus, ubiquitous computing is also often referred to with the term *ambient intelligence*. Pervasive computing, on the other hand, aims at making the information on anything available everywhere, which typically requires having a distributed set of pervasive devices like smart phones and wearable devices, for instance. Ambient intelligence and embedded devices typically have much more limited user interfaces than mobile devices though, and thus the UbiComp research often studies the appearance (or disappearance) of the entities, in addition to novel human-computer

interaction techniques. Kindberg and Fox mention three important goals for ubiquitous systems [75]. Firstly, UbiComp software must deliver functionality for our daily activities. Secondly, this delivery must be done on failure-prone hardware that has limited resources. Finally, UbiComp systems must be able to operate in extreme conditions.

At present, due to the wide availability of mobile devices and ubiquitous Internet connectivity, the computing landscape looks more pervasive than ubiquitous. However, the vast development in physical and cyber dimensions offer opportunities to promote more ubiquitous computing aspects. Nevertheless, ubiquitous and pervasive computing are both broad research areas that share many research domains.

Pervasive service compositions research studies how entities of different types can offer services, and how these services can be combined into more meaningful compositions in an automated way. Typically, pervasive service compositions systems follow a service oriented architecture (SOA) architectural pattern, where the application components provide their services described so that other entities can find and utilize them. As the task of composing pervasive services is challenging, Brønsted *et al.* have defined four main goals for such systems in [17]. These four goals give an idea about the main research topics of the pervasive service composition domain. The first goal is context awareness, which means that a pervasive system should be able to detect and react to the changes that take place within the computing environment. Moreover, the system should also support taking context into account while composing the services, or handling contingencies. The second goal is to be able to manage contingencies. As contingencies are inevitable, the system needs to detect when something is not working as expected, and then offer support for replacing the services with other services as needed in an automated way, or support notifying the users. The third goal is leveraging heterogeneous devices, which essentially means that the system should support utilization of different types of entities by distributing responsibilities in the composition based on the entities capabilities. As automated solutions don't work in many cases, the fourth goal is to empower users to recompose and configure the compositions manually.

Proactive computing is a form of ubiquitous computing, where the goal is to gather information regarding people to analyze and anticipate their needs [130]. Indeed, this kind of data mining is currently already being done in the background, without people even realizing this. For instance, data is collected on people's Web usage. On the other hand, many telecommunication operators are already gathering vast amounts of data about users' contexts and from their network usage. Similarly, many cloud service providers analyze users' content and usage. Typically, this data is then analyzed, and user profiles composed based on the results. Often the composed profiles are utilized for commercial purposes (e.g. advertising). Slowly, however, proactive computing has begun to emerge in other types of applications, like the digital assistants discussed above.

The goal is to gather even more precise information about users' activities, and for this purpose ubiquitous technology can be utilized. Sensors and other types of physical entities, for instance, now offer an efficient way of gathering vast amounts of new types and more precise information about people that can possibly be quite intimate. Naturally, this raises many concerns about privacy, and raises the question of who owns this data? Despite these privacy issues, proactive computing can offer many benefits for improving computing. Moreover, the privacy and data ownership issues can be improved by studying new types of technical solutions.

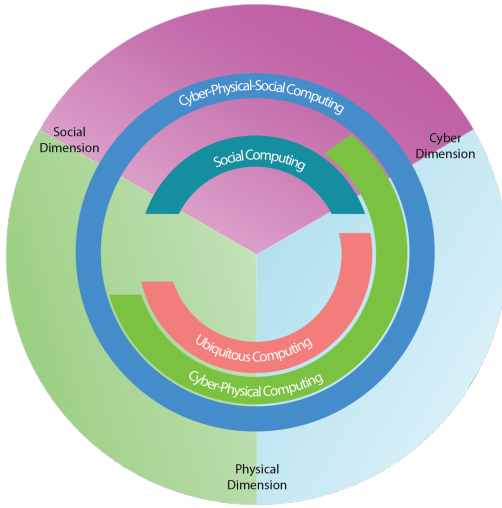


Figure 2.6: Computing concepts placed in the interactions dimensions.

Cyber-Physical Computing

Cyber-Physical Computing aims to bridge the gap between the physical and computing worlds [115, 139]. For this purpose cyber-physical systems (CPS) typically offer tools for integrating physical processes into computational processes [80]. Moreover, within their feedback loops, computational processes can affect back to the physical processes. With this integration, the two worlds can then collaborate and adjust the processes, possibly in real-time.

Today, in the physical dimension, there are plenty of sensors and devices equipped with input components for measuring physical processes. Typical examples include health and wellness devices. However, the cyber dimension gives the possibility to combine several sensors, and measure vast amounts of data from larger scale physical processes and geographical areas. For example, processes related to agriculture, traffic, transportation, civil infrastructure, weather, and climate can be measured and used as data sources. With the sensors continually becoming cheaper, it is possible to harness new processes.

The pure raw data itself, however, is only rarely useful, and thus it needs to be processed to become something more meaningful. Often this process is depicted with the DIKW Pyramid, where the idea is that the data first becomes information, then knowledge, and finally wisdom [116]. The first step to take is typically to pre-process the data for instance by filtering, cleaning, or correcting some fault values. Information is then inferred from the data by for example structuring, organizing, classifying and clustering the data, in such a way that it becomes something more useful and relevant for the context. Becoming knowledge depends on the context. The idea is to extract interesting and relevant features from the information over time, possibly from multiple sources, and understanding how the variables affect the process. Finally, wisdom can be achieved over a longer period of time with a deeper understanding of why a process works as it does.

Physical-Cyber-Social (PCS) computing is an emerging paradigm which adds a third dimension to this equation [116]. The social dimension offers valuable information

about humans' backgrounds and social behavior that can then be considered in these computational processes. The sources for such data typically include social networking services and online communities from several different areas. Sheth *et al.* describe an example scenario related to healthcare. In this scenario the idea is that the humans' health and wellness are being measured with several devices and profiles are composed, not only by these measurements, but also based on the available background information of these people, such as age, gender, race, living environment, and so on. Thus this information helps to define the context for the measurement results, and compare the results to those of similar people. The goal of PCS computing is to tie these worlds more deeply together so that the processes will support each other. From the interaction perspective, this brings the cyber world and physical world closer to each other. According to Nielsen *et al.* [97], a lot of sensor data can enable new kinds of research to better understand the behavior of the people in various kinds of environments.

Summary

To summarize, compared to UbiComp the focus in Cyber-Physical Computing is more on data mining and analytics. Also, the scale of the performed data analysis is typically larger in CPS than in UbiComp. In UbiComp, on the other hand, the focus is often more on the functional and interaction perspectives than in the data analysis itself. Moreover, according to Kindberg and Fox [75], the world should consist of several UbiComp systems rather than one system, and thus the boundaries of UbiComp and cyber-physical systems are different. Whereas a CPS may produce data from a wide range of geographical areas, a UbiComp system should follow the boundaries of the physical world. Sheth's visions [117] and [116], on the other hand, are close to the original vision of Weiser's ubiquitous computing [142], but the perspective is more on the computing taking place in the background which then can support the interactions. Hence PCS computing recalls Tennenhouse's definition of proactive computing [130]. Nevertheless, these paradigms are making way for research to become more interdisciplinary.

2.3 Visions and Approaches

The following section will give a short introduction to the three concepts, *Social Devices*, *Internet of People*, and *Mobile Content as a Service*, reported and described with more details in Publications [IV], [I], and [VI]. Here, the concepts are described from the perspective of how they are related to the interaction dimensions, and how they contribute towards improving interactions within the computing environment.

2.3.1 Social Devices

This chapter discussed how nowadays an increasing number of interactions between people take place in the virtual world, through several Internet-based services (Figure 2.7, *social – cyber dimensions*). At the same time, an increasing number of objects from the physical world are being connected to the cyber world (Figure 2.7, *cyber – physical dimensions*). This enables these entities to communicate with each other, gather information about physical world processes, as well as control some of these processes.

Social Devices is a concept that aims to improve social interactions that take place in the physical world, with multiple devices and people co-located (Figure 2.7, *physical – social dimensions*). In Social Devices, humans together with the entities form a novel

socio-digital system where both can participate into the interactions that are initiated *proactively*. The aim is that the Social Devices operate with the user in their surroundings, facilitating and augmenting their daily activities and routines.

Improving the co-located and social interactions requires considering that all the interaction dimensions are equally important, as the enablers from each dimension support the interactions, but from different perspectives. The word *social* in the concept can essentially refer to three points:

1. The entities should be able to share their own resources and utilize resources of other entities that are around them.
2. Leveraging the resources should happen in such a way that it supports social interactions between the people and entities that are nearby.
3. An entity can appear in social and humane way for humans.

In the concept, all the different physical entities, from single sensors to servers, can be regarded as social devices. On the other hand, a set of entities, like a sensor network or home automation system, can be regarded as a single social device, depending on the purpose. Thus their physical and social appearance can vary a lot. This affects the utilization of the resources of these entities: some of the entities appear in social way for the humans, whereas some stay more in the background, offering information and resources for processing or storing data. Some social devices are highly *personal*, used solely by their owners, whereas others are shared devices and remain *impersonal*. Moreover, some social devices are *mobile*, being carried everywhere by humans or moving autonomously for instance, and some social devices are *stationary*. More details about the concept can be read from Publication [IV].

As interactions between the entities, as well as between the humans and the entities, are the core of Social Devices concept, a programming model has been designed for implementing such interactions – *Action-Oriented Programming (AcOP) model* offers developer-friendly abstractions that help in defining and programming the social interactions within contemporary computing environments.

Research Projects

Originally the concept was developed as a joint research work of Tampere University of Technology (TUT), Nokia Research Center (NRC), and Aalto University during 2011 – 2012 in the *Cloud Software Program* (`http://www.cloudsoftwareprogram.org`).

After the Cloud Software Program concluded, the research work was continued by TUT in an Academy of Finland funded project (#264422), named *Co-Located User Interaction with Social Mobile Devices*, *CoSMo* (`http://www.cs.tut.fi/ihte/projects/CoSMo/`) during the years 2013 – 2015. Some of the work was also conducted within another Academy of Finland funded project (#283276), named *Device-to-Device Security*.

Research and Implementation

In the Cloud SW program each partner of the Social Devices team focused on different aspects. The Aalto team mainly focused on recommendation and configuration aspects. The Nokia team mainly worked on designing the original scheduling and triggering related

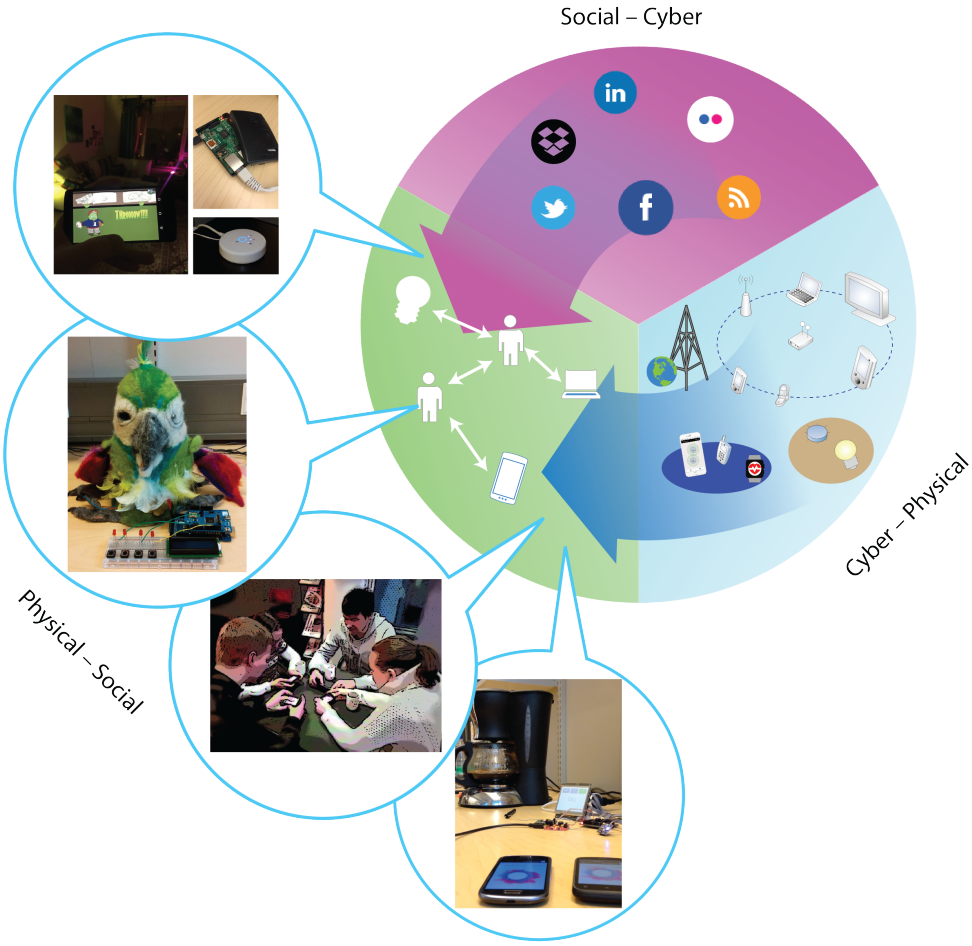


Figure 2.7: In programmable interactions, the focus is on the Physical–Social sector.

aspects. TUT focused on entity coordination and implementing the development and runtime environments for the programming model that was designed together with the partners. The concept was designed in collaboration, and resulted in a proof of concept implementation named *Social Devices Platform (SDP)* that was introduced together with the concept in Publication [IV]. SDP consisted of three cloud services, respective to the three partners, and was developed while having in mind the possibility of it becoming a real product with Nokia aboard.

Starting from 2013, however, the implementation of the concept has been developed solely by the author of this thesis. In the CoSMo project, the whole concept was re-implemented. The new implementation, named *Orchestrator.js (OJS)*, is targeted to be more lightweight and streamlined to better support prototyping of Social Devices applications. At the same time with the reimplementation, the original AcOP model was developed further. OJS

and the refined AcOP model have been reported in Publication [II]. The solution is open source, and an instance is running in Amazon Cloud (`http://orchestratorjs.org`).

The CoSMo research project aimed at studying novel technologies from social interaction perspectives. Some of the most central questions of this project included:

How can devices initiate social interaction with co-located people? Technology has enabled people to interact with like-minded people in the digital world. Could technology also be used for helping people with similar interests to connect in the physical world? Devices could, for example, proactively try to initiate an interaction between people by encouraging one person to speak to another in a conference, or suggest a date for two singles with matching profiles.

How to enrich social interactions in co-located situations? Technology has enabled the production of significant amounts of digital content. This content is already used for enriching social interactions, and sharing life events in social media. Could this content be shared in co-located situations as well? For example, devices could suggest sharing a photo album while friends are gathered.

How to facilitate interactions between people and technology? The ever-growing number of physical entities requires ever more attention and time in configuring and managing the technology. But could the technology actually work for their users, rather than forcing users to adapt or change their behavior? For example, a car navigation system and a mobile phone could automatically negotiate a route based on a calendar entry, and then inform the user in a social and user-friendly way.

How to bring people together? Technology can help automate many tasks in people's daily lives. At the same time, however, technology seems to also isolate people from other people nearby. Could technology instead be used to bring people in the vicinity together instead of pushing them away from each other? For instance, a coffee machine could invite people at the office to have coffee together.

Architecture of Social Devices

The architecture of Social Devices has changed over years of research. Although the centralized cloud has yet to play an important role in the current implementation, attempts have been made to reduce its role. The aim is to enable devices to interact directly, more independently of Internet connectivity – The devices themselves would then be part of the decentralized clouds where the tasks are distributed to the entities in the people's surroundings. The present implementation has been described in Publication [II], but it has been complemented with more direct device-to-device coordination as described in Publication [III]. In the future, the goal is to utilize the centralized cloud service primarily as a development and deployment platform, and to offer a linking service for information how the entities can connect to each other.

2.3.2 Towards an Internet of People

An Internet of People is a vision of a more human-centric Internet of Things. For this purpose, the vision declares a manifesto of how the Internet of Things can become the Internet of People by improving interactions. Thus, the Internet of People approach aims to contribute to the same aspects as Social Devices (Figure 2.7, *physical – social dimensions*). In this thesis the IoP manifesto has an important role of defining the fundamental principles of programmable interactions.

Research Collaboration

The IoP vision has been designed in collaboration with a Spanish research group, during the author's research exchange visit. The Spanish team comes from University of Extremadura and from University of Malaga, and has studied a concept named *People as a Service (PeaaS)*. This concept aims at deeper integration of people into computing. Throughout their research work, the author of this thesis and the group shared a similar ideology of how the Internet of Things should be, but their research was conducted from different angles. Thus the author made a research exchange visit to meet this group, which then resulted in the IoP manifesto.

Reference Architecture

In addition to the manifesto, the Publication [I] describes a reference architecture for the Internet of People. This architecture is a consolidated work of Social Devices and the PeaaS model, and their implementations, Orchestrator.js and nimBees platform, respectively. The benefits of this architecture are explained with user-centric scenarios in the Publication [I].

2.3.3 Mobile Content as a Service

Mobile Content as a Service is an approach designed to meet the growing demand of managing digital content. Today, the computing environment contains plenty of entities that can produce, store, and utilize digital content. However, the challenge is that the content is typically scattered among the entities, and it is the user's responsibility to keep track of what content there is and where it is located. Whereas cloud services now offer solutions for synchronizing content between entities, they typically require uploading everything to the cloud service. The Mobile Content as a Service approach, however, differs in its idea as only metadata of the content is uploaded to the centralized server, and the essence of the content is exchanged directly between the entities whenever possible, and when not possible, Internet connectivity is used. Thus, the Mobile Content as a Service approach mainly improves the interoperability between the entities, (Figure 2.7, *cyber – physical dimensions*). The approach has been depicted in Figure 2.8.

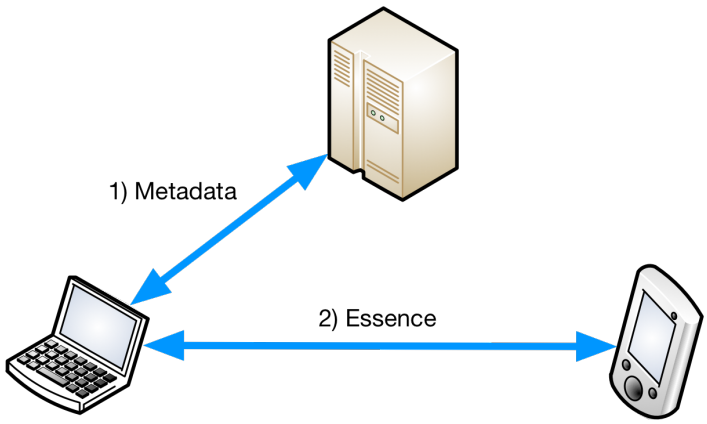


Figure 2.8: Mobile Content as a Service approach.

Research and Implementation

Research around this approach first started in 2009 in collaboration with Nokia Research Center and Tampere University of Technology. Later, when Cloud SW Program started on 2010, Aalto University also joined the research team. The concept was developed by this team until 2011, but was later continued by others [100].

The original implementation is named VisualREST, referring to the REST technology that was used. The author of this thesis has implemented around half of this system, and the implementation has been described in the author's M.Sc. thesis [86], and in Publication [VII]. Aalto University was developing client applications for mobile devices that utilized the VisualREST system.

Chapter 3

Fundamentals of Programmable Interactions

Interactions play a key role in today's computing environment. Ever more interactions take place in our proximity, where we co-exist with diverse sets of entities. Simply co-existing, however, is not an option. Already too much of people's time and attention is taken while they are slavishly using their devices. The problem is that interactions between computers come from different dimensions than the interactions between humans. Thus the interactions are fundamentally different. In many ways, human interactions with computers feel forced and unnatural.

This current situation has much room for improvement. As technology today is an inseparable part of our lives, it is essential to make interactions as human-friendly as possible. Programmable interactions are defined by the following four fundamental principles:

P.1. **Be Social**. Interactions in which entities and humans participate must be social. In particular, interactions should allow for heterogeneity by supporting the different types of entities that people use, and let them interact with each other and with people more socially than does the IoT. Entities will have to be aware of their current computing environments and be able to automatically adapt their own and other entities' social behavior. Users need to be empowered to adjust their preferences and policies about when and with whom their entities are socializing, and with which interaction modalities.

P.2. **Be Personalized**. Interactions between entities must be personalized to users' sociological profiles and environment, allowing for contingencies and providing a transparent mechanism for this customization. The interactions must consider the sociological profiles of all participating people. Again, users must be empowered to adjust their preferences to control how others use their profile.

P.3. **Be Proactive**. The triggering of interactions must truly be proactive, not manually commanded by the user. Today, most scenarios where multiple entities participate only consider remote interfaces for managing connected entities. But ever more entities online mean more distractions and work in managing

them. The interactions should allow for entity heterogeneity so that they can all interact more proactively. Users must be empowered to adjust their preferences to control how proactive their surrounding entities are. Note that letting entities act proactively could entail security risks that must be analyzed and mitigated, keeping users informed of any established proactivity policy.

P.4. **Be Predictable**. Interactions must be predictable – that is, they should be triggered according to a predictable environment that the user has previously identified, and for which a specific behavior has been defined. Users must be empowered to identify and tag that environment, specify the expected behavior of the entities involved, and set the privacy policies for sharing their information by being advised of what information they're sharing and with whom. Given that complete predictability of interactions is hard to achieve, the user must always understand how the interaction can be stopped immediately, and also how to prevent this misbehavior in the future.

These principles originate from the *Internet of People manifesto* from Publication [I]. Minor changes, however, have been made here to align the terminology with the thesis. The manifesto was declared by two research teams facing similar challenges, but from different perspectives. The remainder of this chapter gives the motivation for interactions based on these principles, and explains what challenges are related to following these principles. For further motivation, the Publication [I] gives examples and discusses the principles.

3.1 Towards a Social Computing Environment

The first principle of programmable interactions is to be social (P.1). In essence this means that the interactions must leverage multiple entities rather than simply one entity and one human at a time. This also implies that other than the user's own entities must be enabled to participate in the interactions. Thus, the interactions should not be limited to any specific set of devices as is typical in the ecosystems of today. Also, the interactions should not only be enabled in predefined locations like in smart homes. Instead, being social means that the interactions must be enabled to take place freely, in an ad hoc manner in any location, and with any entity that the user specifies. Moreover, from the input and output perspective, entities should not only interact silently and in the background as they now do. Rather, the entities should interact with humans in more social ways, and with interaction modalities that are more natural for humans.

Being social helps humans integrate with computing as they can be more aware and more in control of interactions. This further helps to build better interactions between people, and makes interacting with multiple entities more seamless. Utilizing and mixing the various resources of each device can offer better modalities for the input and output in different computing environments. More human-friendly modalities can then facilitate or even encourage co-located people to socialize, while using technology is demanding less attention. Moreover, the collaboration of the entities would support the collaboration of people, and for this reason entities of multiple users need to be enabled to interact.

Following the principle of being social (P.1) requires that the entities are operating in a computing environment where they are aware of each other. This further addresses three

major challenges from the social dimension perspective, and thus here the term *socially computing environment* is used. These challenges include:

C.1. Awareness of entities and humans. Entities and humans must be able to detect and identify other entities and humans nearby.

C.2. Awareness of entities' interaction capabilities. In order to socialize, entities and humans need to become aware of each other's resources and reveal their interaction capabilities.

C.3. Awareness of social relationships and ownership. The relationships of entities should be considered in interactions. Especially important are relationships between their owners.

The Role of Companion Devices

The principle of being social(P.1) for an entity requires being aware of other entities that are present (C.1), as well as indicating its own presence and interaction capabilities for others (C.2). Devices and humans, however, have different methods for detecting nearby entities. Whereas humans have natural aids for detecting things in our surroundings (our senses), entities are much more limited with these, and special technical enablers are required.

Social Devices and Internet of People concepts utilize mobile devices as both virtual representatives and extensions of their owners. Whereas some other types of devices (e.g. wearables, smart watches) can also be used, these concepts exploit the fact that at present mobile devices are the devices that are most commonly with their owners, and contain much personal and intimate information about them.

In Social Devices and IoP these special entities are named *companion devices*. The idea is that as the companion devices are carried by their owners wherever they go, and have excellent connectivity support, they can represent their owners for other entities, and via other entities also for other users. Thus by utilizing the cyber dimension, the companion devices extend and bridge the users' abilities to interact with the entities in the surrounding physical space (C.2). Especially useful, this is in interactions with many IoT smart objects that do not necessarily have any kind of user interface. The role of the companion devices has been depicted in Figure 3.1.

Supporting Social Behavior with Interaction Modalities

Interaction modalities define how people and devices interact, and for this reason have an important effect on how we and other people surrounding us react to interactions, and to computing in general. To avoid living in the two worlds, virtual and physical, at the same time requires integrating the user's digital life into the physical world. An example of this approach is the current generation of smart watches that show notifications on their screens, aiming to free the user from picking up their phone so often. These devices can also be used for reacting to these notifications, but the interaction is still fairly limited, and at least not yet anything social (P.1) – glancing at the watches may actually even detach the user from ongoing face-to-face interaction. Another issue of these devices is their interfaces: The screens are small, and whereas the voice can be a natural modality

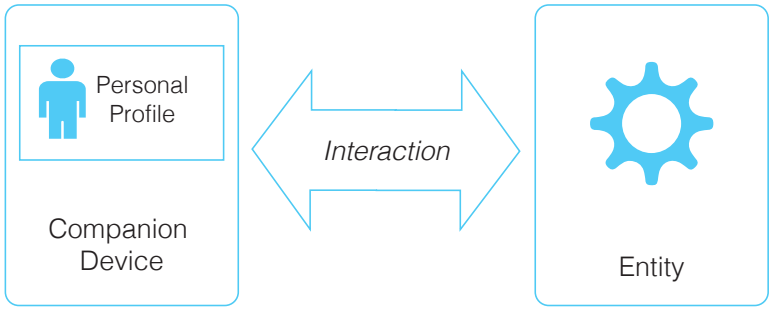


Figure 3.1: In programmable interactions companion devices represent their users for others.

for communicating with others, talking to your own wrist is not yet socially acceptable. Obviously this can change in the future when people become more accustomed to this kind of interacting, similarly to how we are now accustomed to people using their hands-free sets. But even using hands-free today may sometimes be confusing to others. Thus the change may take time. However, there are many contexts where the voice can actually be the best interaction modality. For example, while driving a car or motor bike, the best option to interact with your devices would probably be voice input and output.

Some approaches have also been studied for improving others awareness of one's social behavior and activities. For instance, Social Displays [65] is an experimental concept for showing other people in the same space how you are currently interacting with your devices. The idea is then to initiate interactions with like-minded people based on similar interests and activities, but the actual initiated interactions are then up to the users.

In Social Devices one of the goals is to enable designing capabilities that can actually initiate, and support interactions with co-located people. The capabilities are directly linked to the interaction modalities, as the idea is to utilize the modalities that best suit each social environment. Some examples include content sharing capabilities and different types of game capabilities. For instance, Social Devices *CarGame* (`https://youtu.be/T3sL3JYjCEM`) is designed for stirring up the atmosphere while friends are having a party. The main modality for humans to interact with the game is voice: the cars gain more speed from the volume of the users' voice. Naturally, this type of interaction or modality does not fit in all situations, but at least based on our experiences, they become more acceptable when people are using them together. Furthermore, as users should be empowered to define how, when and with whom are they are willing to interact (P.1), companion devices must also enable adjusting these preferences. Social Devices implementations, for instance, have responded to this challenge by allowing the user to enable and disable interaction capabilities separately for each entity (C.2). Then people who are not willing to participate in these kinds of interactions can simply leave them disabled on their companion devices that are used for representing their owners. At the same time, supporting experiences with like-minded people become easier, which further supports following the principle of personalized interactions (P.2).

Although companion devices are typically carried in a pocket or bag, they are not always at hand. This forces the user to pick up the device countless times a day, and with more entities online there is the danger that it become a remote control for everything. Moreover, although large, high-resolution touch screens now offer decent interfaces for controlling things, it is rarely an appealing interface for any long lasting activity. For

this reason, the role of the companion device should be less active from a human-entity interaction perspective, but *active from an entity-entity interaction perspective*; companion devices should be considered as first class citizens for representing their owners, but as active interfaces only in situations where its modalities either fit best for the interaction, or when no better alternatives are available. In other cases, the surrounding entities should be utilized as interfaces.

Alternatives for Detecting Co-Located Entities

Currently the most-used technology for location aware systems are satellite based positioning systems, like Global Positioning System. GPS, however, only works outdoors. Moreover, it is not targeted for detecting other entities in the proximity, but simply determining the entity's own location on Earth. Indeed, entities may communicate and exchange their location information in order to become aware each other, but naturally then the entities first need to be able to communicate. For more precise indoor positioning, approaches such as Wi-Fi base station-based position determination [2] can be used.

The Social Devices concept is especially designed to support interactions that take place in an ad hoc manner in co-located and face-to-face situations, where many people and different types of devices exist. For this reason we have studied how nearby entities can be detected, and how this information can be maintained and utilized. As described already in the introduction of first proof of concept system in Publication [IV], the detection of nearby social devices has mainly been based on *Bluetooth service discovery* technology. The idea has mainly remained the same: the companion devices map the received signals with identifiers to the entities, and the received *signal strength indicator values (RSSI) to physical distance*. This information is then used for composing a proximity graph of the entities. Furthermore, a new wireless device-to-device technology, named Wi-Fi Direct or Wi-Fi Peer-To-Peer (`http://www.wi-fi.org/`), offers service discovery that is similar to Bluetooth. Although, at present this technology has limited support, it has begun to emerge among Android devices. Thus in the future, this technology can be used to complement the companion device entity discovery for also detecting devices that have no Bluetooth support.

Another alternative for companion devices to detect *the presence of humans* is, for instance, *active badges* that were described already by Weiser in his early vision of Ubiquitous computing [142]. Originally these badges were based on infrared technology, but today there are multiple options that can be used, such as NFC [89], QR codes, and beacons of different types (e.g. `http://estimote.com` and `https://passkit.com/`). The difference is that despite of their name, the badges are fairly passive devices and their capabilities for interacting with the user are limited. In contrast, the companion devices offer an important interface to their owners, and also for interacting with the user in many situations, as at present there aren't many better options are available yet. In the future, when the number of entities capable of more active interactions grows in our surroundings, the role of the companion device from human-entity interaction perspective will become less important, and then maybe can be replaced with less active technology, like active badges, for instance. Until then though, companion devices will be the primary tools for interacting with entities.

Describing the Relationships of the Entities

Another major challenge for the social computing environment is to describe the relationships between entities with different properties (C.3). For this purpose various social media services offer valuable information about the relationships of people, and this information can also be linked also to the relationships of entities [20, 39, 57]. Additionally, the relationships can be described by maintaining information about how often the entities come into contact, and how long they stay within each other's proximity. Furthermore, this history data can include information about what interactions we have previously performed with some entity for proposing similar interactions with that entity in the future, and information about canceled interactions for avoiding unwanted interactions. Social Devices, for example, maintain history data about triggered and cancelled interactions, and their parameters.

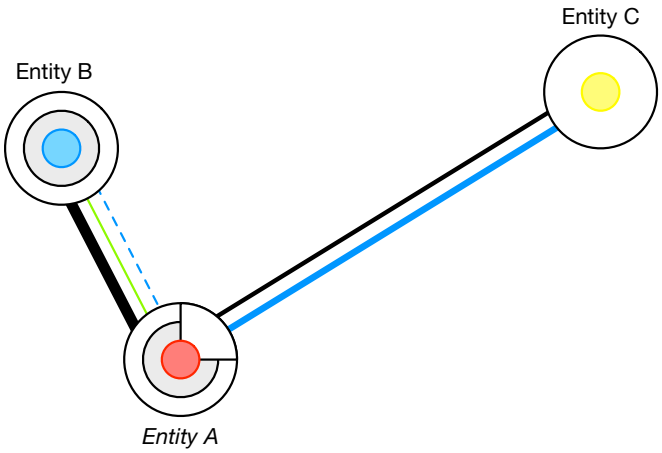


Figure 3.2: Socially computing environment from *Entity A*'s perspective

Together these two tasks are the key for defining the social closeness or awareness of the entities. Being socially aware of nearby entities leads to a more human-friendly computing environment. This awareness then helps to build better co-located interactions between the most useful entities. For example awareness defines what entities we can utilize, and what type of interactions we can perform with them. Figure 3.2 illustrates a socially aware computing environment as a graph. In this weighted graph the *nodes represent the entities*, and the *black edges represent the distance between the entities*, and their weight describes how often the entities interact with each other. The *colored edges are properties that illustrate relationships* in social media services. Similar to the black edges these edges could also be weighted based on how often we interact with the owner of the entity in social media. The dotted edge represents that we are only following the owner in social media and thus the connection is only one way. Later this information can then be utilized while deciding what interactions should be triggered and with whom. As an example, in [1] we have presented an idea of sharing photos published in Flickr (`http://flickr.com`) social media service while we are gathered together with our Facebook friends. Section 4.2 presents an idea of utilizing this relationship information for improving the interactions by forming connections between the entities in proactive manner.

3.2 Issues of Continuously Detecting Proximity Set

The third principle of programmable interactions is to be proactive (P.3). At present most interactions between humans and computers are initiated by the users. This makes the interactions asymmetric and one-way only, meaning that the user must manually give commands to the computer which then performs the tasks on behalf of the user. At the same time, we have become more active users of computing, as we have from the beginning of the computing era. Now we are living in the beginning of an era where more and more entities are getting connected and becoming capable of doing some kind of computing. However, having more entities entails more distractions and effort in managing them as they require our input.

For this reason, interactions should be triggered in an automated way, without user interaction. This means that entities also need to become enabled to initiate the interactions or otherwise these entities will simply add extra steps to our daily lives instead of making them easier. Thus empowering the entities to initiate the interactions must be done in such a way that it enables two-way communication and supports the principle social interaction (P.1).

Typically, triggering is based on detecting changes that happen in the computing environment, such as the entity's location. From the social dimension perspective, however, the main challenge in supporting the proactive triggering of co-located and social interactions is to be able to detect the changes in the socially aware computing environment:

C.4. Continuous detection of changes in proximity set. To proactively trigger interactions, entities must be able to detect when other entities move in and out of their proxemics. This detection should work independent of the current mode or state of any of the entities.

Issues of Utilizing Bluetooth Technologies to Support Proactive Triggering

Social Devices research was started in 2011 and at that time Nokia N900 phones with MeeGo operating systems and Bluetooth version 2.1 were used as a development platform. As the idea is to support interactions with any entities and not just with previously known ones, this sets special requirements for utilizing Bluetooth technology. For example, information about the connection status cannot be used as it would require first establishing and maintaining the connections. This is the typical approach taken in many of the recently emerged proximity-based keychains that alert when the user goes far enough away from keys, and the connection drops. For this reason we ended up using Bluetooth service discovery for detecting the entities as was mentioned above. Classic Bluetooth service discovery, however, has had, and still has several issues that prevent using it for triggering interactions *proactively* for co-located entities. In Publication [III] we reported some of these, which include:

- Traditional service discovery approaches can be too slow for triggering some social interactions [8, 72].
- Making the device discoverable may require manual efforts from the user, and the device may be visible only for a short period of time, or while the Bluetooth settings panel is running in the foreground.
- Constantly performing service discovery quickly drains the battery.

- Constantly performing service discovery may cause the Internet connection to break when Bluetooth and WLAN interfere.

Figure 3.3: Social Devices proximity graph in Orchestrator.js system.

One of the main issues of classic Bluetooth was power consumption. For more power efficient connectivity, Nokia started developing *Low End Extension* for Bluetooth that was later names as *Bluetooth Low Energy (BLE)* [56]. Today BLE is often marketed with the name of Bluetooth Smart, and it has become an essential part of Bluetooth since version 4. Hence, the power consumption of advertising one's presence is not an issue anymore: Coin-sized devices are capable of advertising their services for a year. However, there still is another issue related to power consumption: Constantly doing the service discovery drains the battery in a few hours, as it did with classic Bluetooth. This issue, however, can be mitigated by optimizing in which situations and how often the discovery needs to be done frequently. Another issue related to BLE service discovery is that the devices tend to hide their real identities of the devices, but this issue can also be managed via a cloud service as described later in Section 4.2.

From the perspective of co-located and social interactions, there are two major issues at the moment that limit or even prevent utilizing Bluetooth Low Energy based service discovery and advertising technologies in a *continuous way* for detecting the changes in the proximity set (C.4). With Android, the support for BLE advertising is still mostly missing due to the limitations in the hardware of many current devices. For instance, LG's Nexus 5 Android reference phone cannot support BLE advertising, albeit it can be utilized for detecting other entities capable of advertising, such as beacons or iOS devices. At present Social Devices utilizes both, BLE and classic Bluetooth, and maps the results together in a cloud-based service.

The second major issue is that although iOS has had support for BLE already since version 5 and iPhone 4S, the implementation is yet very limited in detecting nearby entities in an ad hoc and proactive way. For instance, iOS at present (version 8) only allows broadcasting one's identifier while the application is running in the foreground; when an application advertising a device's services is sent to background, the discovering device stops detecting the real identifier of the advertising device, and only gets randomly changing identifiers. This efficiently prevents detecting iOS devices that are not running

the application actively in foreground, when the phone is locked and in a pocket for example. Clearly Apple plans on utilizing BLE advertising with their iBeacon branded technology that was introduced in their Worldwide Developers Conference (WWDC) 2013 for location-based interactions. However, it seems that their priority is that the device is able to detect other entities, rather than being detectable by other entities.

Both of these issues are in conflict with the principles of being social (P.1) and being proactive (P.3), as they effectively prevent truly proactive triggering of social interactions with others. It will be interesting to see how Android will manage identities when it gets support for advertising. For now, the best option to circumvent these interoperability issues is for the user to carry a portable beacon that can constantly advertise its presence.

Despite the issues of Bluetooth service discovery, we still consider it the most promising technology for continuous detection of the entities to support the proactive triggering of social and co-located interactions: It is wireless technology, and at least in theory allows performing discovery without any manual effort from users. The range is also well-suited for this task, and the signal strength can even be used for detecting the distance between the entities. Moreover, recent contributions from the major mobile device manufactures, like Apple's (`https://developer.apple.com/ibeacon/`), and Google (`https://developers.google.com/beacons/`), are proving that Bluetooth technology is being developed for detecting the device's location more precisely, and may help to overcome the issues related to detecting the companion and other devices.

3.3 Engineering the Social Dimension

The second principle for programmable interactions is for them to be personalized (P.2). This means that the computing environment should adapt its behavior based on the user's preferences, instead of forcing the user to adapt their own behavior. Hence the preferences and digital life should follow the user seamlessly to any location. To support the principle of social interaction, also sharing the experiences with other people in the same space, and taking into account their preferences (P.1). This requires that the digital life brought into physical life needs to merge and adapt to the life of other people that are in proximity, and participating in the same interactions.

Section 3.1 discussed how social media information about users' social relationships can be linked to describe the nearby entities. However, our different devices, and cloud and social media services have a lot of other important information and digital content that can be used for supporting personalized interactions (P.2). The actual challenge then is in harvesting this information to be exploited for triggering the interactions, and then to be accessed within the interactions. Although the focus of this thesis is not in mining information from social media as such (e.g. social mining [109]), the following gives a short overview of our idea of providing a more complete profile of the user, which in Publication [I] we call as sociological profile. From the programmable interaction perspective, utilizing the user's digital life for personalization purposes addresses the following challenges:

C.5. Support for accessing virtual world from the physical world. The computing environment must support exploiting the user's cyberspace information in the interactions that take place in physical space.

C.6. Support for detecting changes in virtual world. The changes in cyberspaces (e.g. in social media) must be detectable in order to trigger interactions in physical space, and thus bridging the two spaces.

C.7. Support for accessing preferences of co-located people. Access to the preferences and content of all the co-located users should be enabled, so that those can be taken into account as far as possible while triggering and developing interactions for multiple humans in the same space.

From Related Work to Collaboration: Towards Sociological Profiles

Indeed, Social Devices has had a concept of a companion device from the beginning. This special entity was mainly used for detecting users' presence in a space, as described above. As truly personalized experiences, however, require detailed information about the user (C.5), the plan was to compose more detailed profiles of the users. Thus we started collaborating with a Spanish research group developing a concept named *People as a Service* [44]. *PeaaS* is a concept for offering personal profiles as a service from user's mobile devices. The concept is based on four main pillars:

1. Mobile devices as interfaces to the people.
2. Sociological virtual profiles.
3. Sociological profiles as a service.
4. User privacy.

Based on the two first pillars, and the discussion above, it can be agreed that the two concepts consider the role of the mobile devices almost identically. According to the second pillar, however, PeaaS aims at a deeper understanding of the user than Social Devices at present. Moreover, the third pillar especially tells the difference between the concepts: Whereas in PeaaS the idea is to provide the profile as a service directly from the user's device, in Social Devices this profile was originally implemented as a cloud service. Naturally, the approach of PeaaS further supports the fourth pillar, user privacy. At the same moment while sensitive data leaves the user's hands, the ability to control this data is lost forever. That is, the user can no longer be sure how this data will be used, and who eventually will gain access to it. For this reason, the intimate data about the user should always be kept secure, and in the user's possession. The end result of our collaboration was a consolidated work, the Internet of People model, where Social Devices and PeaaS complemented each other. In this approach PeaaS's *Sociological profile component* is being used as described in Publication [I].

With this profile component, the idea is that it contains basic information about the user, such as age and gender, as well as some fairly static information, like personal preferences and interests. More dynamic data is then gathered with the companion device's sensors, similar to how we have done in Social Devices implementations. The difference now is that very sensitive data is always maintained on the user's companion device, and never leaves there, at least not without user's consent. The idea is that only user-authorized applications are allowed to query the data from the profile. The centralized solutions, however, have many benefits as data from different sources can more easily be gathered together for doing computations. Moreover, the capacity of the mobile devices for storing

data is limited, and for analyzing very large data sets (Big Data) requires resources that exceed the resources of mobile devices. For this reason, some less intimate data can, by the user's consent, be relayed to a cloud-based *State Registry*.

A similar approach has also been taken, for example, by Huang *et al.* in their *Mobile as a Representer model* [59]. The *MaaR model* is an approach towards user-centric mobile cloud computing. The idea of mobile phone as a representative of its user is similar to the original implementation of Social Devices, but different from the Internet of People model. As a distinction to both, Social Devices and IoP, in MaaR the user's virtual life is separated from the physical life, and this entity or profile is then queried when information about the user is needed. Hence, the approach resembles typical examples in Social computing and Proactive computing.

Social Media and Personal Profiles for Composing a Sociological Profile

From the early ages of Internet computing, when the (World Wide) Web first became publicly available, people used it for creating home sites containing personal information about themselves. Today, several services enable known and unknown people to create profiles, and then to communicate and socialize remotely. Compared to the regular home sites, these services contain similar, or even more personal information about the people. Furthermore, these services, such as Twitter and Facebook, are actively used by sharing status updates and Web pages, and by indicating that a person "likes" some certain brand or other person's status update. This information becomes valuable, and can be exploited in the programmable interactions, for example by generating natural language from the users' posts to the social media, as studied by Szoniecky *et al.* in [125]. Especially important, this information is used for finding common interests within a group of co-located people (C.7).

The advantage of these services compared to the classic websites is that they typically offer APIs for accessing the data, and that the data is in a structured format that can be easily parsed by the applications. As an example, Figure 3.4 shows what data is available via Facebook's *Graph API* (`https://developers.facebook.com/docs/graph-api`). As was discussed above, the relationships of the users help also identify the entities, and help consider how we should interact with them. By linking these profiles to the sociological profile component, some of the basic information of the user can be automatically imported. With the companion device, this information then always travels with the user, and can be accessed and utilized by surrounding entities in programmable interactions (C.5).

Today's mobile devices enable rich sources of sensory data [31, 57], and can be used, for example, for collecting different types of health and wellness data about the user, as depicted in Figure 3.5. The wellness and health data are perfect examples how this type of sensory data now typically gets stored into silos, or otherwise (for example because of their changing formats) are hard to get accessed by third party applications, and hence cannot be utilized by programmable interactions. Fortunately, services like W2E now offer access to this kind of data in unified formats and via RESTful APIs (`https://www.w2e.fi`). These types of health profiles as a part of the sociological profile component can then be used for improving our wellbeing, for example, by suggesting we exercise to improve our psychological wellbeing [51]. Moreover, making exercising a more social event may help to motivate some people. As wearable technology evolves, maintaining and analyzing the input from these devices can be used for proactively adjusting our surroundings to fit us best, for example by adjusting temperature and lightning to our working environment to

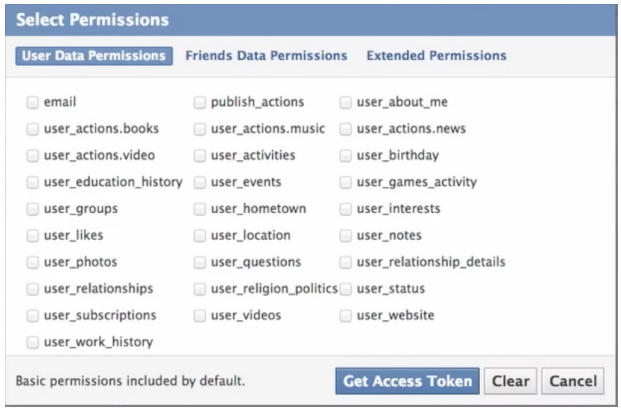


Figure 3.4: The data available of the user from Facebook via Graph API.

keep us better focused, and different ones to keep us comfortable while we are driving our car or reading a book at home (P.3).

Although the goal is still distant, the idea of the sociological profile component is that eventually it could reflect the companion device owner's personality and even *mood* for humans and other entities nearby as we presented in Publication [I]. This would then help in defining how social we feel by taking into account the user's current preferences (P.1), and also make the proactive interaction triggering more precise, and thus more predictable (P.4). Google has also recently studied and patented a high-level approach for collecting information in social media and cloud services to reflect their users' personalities in social robotics [35]. However, the patent has also been criticized as, on one hand, because Google gets most of its revenues from advertisements and, on the other, these types of

Figure 3.5: Screen captures of Apple's Health app for iOS, and Activity app for watchOS/iOS.

Figure 3.6: In PhotoSharing the user browses through a photo album shared in social media, and manually picks photos to share with co-located people.

patents may prevent others from working on technical implementations [25]. In addition to social mining [109], many have also previously studied how people's Web usage can be mined for composing profiles about their activities [48, 101]. These studies will eventually give valuable input when we aim to understand the user's behavior more deeply.

Digital Content in Physical Space

The lives of people have become increasingly digitalized, and digital content is now involved in most of our daily interactions with computing. Content is used while collaborating and socializing with others, which typically requires sharing that content with others. In many cases the sharing takes place via remote services. Take Facebook for example: many people now post their holiday photos as status updates. Also, remote collaboration is supported with tools like Google Docs, or Microsoft's Office 365. However, while we are co-located, collaboration is simply somehow expected to happen. While the number of entities in our surroundings grows, and the content is already in digital format, it simply makes sense to also enable collaboration while we are co-located with other people. For this reason, the presented *Mobile Content as a Service* approach enables uniform access to all of the user's content stored in social media, cloud services, or in any of the user's devices (C.5). Moreover, the service offers notifications whenever new content is published, or old content changes any of these media. As the presence of the user can now be detected by other entities, this enables whole new kinds of sharing experiences with the co-located people, as well as exploiting content in new ways by the surrounding entities. For instance, photo sharing can take place while friends meet in a cafeteria as presented in Figure 3.6. Or, at a party, all the people's musical taste could be taken into account based on their profiles, and the music then accessed directly via their devices.

3.4 Predictable versus Proactive Behavior

The fourth principle of programmable interactions is to be predictable (P.4). Naturally, users must be able to trust the interactions that take place. Typically this requires that interactions get triggered according to users' expectations. In the same vein, the users

should be able to trust that any sensitive data is shared only with entities they trust, and by taking the social environment into account. Hence even interactions that are targeted to surprise must carefully consider to not exceed the predictability too much. Moreover, interactions with strangers should at first be conservative and, only if the interactions continue, then become bolder.

As can be noted, being predictable is highly demanding principle. This mainly results from the principle of being proactive (P.3), but being social (P.1) and personalized (P.2) also have both negative and positive influences on predictability. Consequently, it is the combination of these three other principles which makes it challenging to follow the principle of being predictable: Being social means leveraging and empowering as many entities to interact as possible, and also possibly with social interaction modalities. Being personalized means that users may be sharing sensitive information with these entities. When such interactions take place in proactive way, the changes are that they get triggered in the wrong environments and the entities behave unpredictably.

Yet, by properly taking into account the three other principles, this also supports the principle being predictable: The awareness of the entities' social relationships and their owners' preferences help define more precisely how the entities behave, and with whom they share and what. Furthermore, with a proper perception of the entities entering and leaving the proximity also then helps to make the triggering more predictable. Thus, in spite of the challenges listed already, the following challenges should also be considered for supporting more predictable interactions:

C.8. Interaction perception. Users should have perception of the ongoing interactions that involve sensitive data, or require user's attention or input.

C.9. Information perception. Users should be able to define what information they share and with whom.

C.10. Environment perception. Users should be able to identify different computing environments, and then enable and disable certain types of interactions in these environments.

C.11. Empowerment perception. Users must be empowered to stop an interaction immediately, and teach their devices to avoid misbehavior in the future.

User's Perception of Ongoing Interactions

Countless interactions can run simultaneously in today's computing environment. Many of these interactions take place proactively, and run silently in the background. This means that the user cannot be aware of all ongoing interactions. However, from a predictability perspective, it is essential that the user is enabled to be aware of interactions that involve sensitive data, may require user input, or may affect to the physical world in such a way that it endanger humans or entities (C.8). In other words, the user should be prepared to react, or at least be warned if any sensitive interactions are about to take place. The difficulty, however, is the dynamic and heterogeneous nature of the computing environment – many of the entities do not have proper UIs and, for instance, the companion device is not always at hand.

With Social Devices we implemented a mechanism to enable the interaction developers to *set a warning in the beginning of the interactions* that they considered requiring one. In a way, this resembled a ring or SMS tone of the current smartphones, and if the phone was in a silent mode, it gave vibrations with a pattern to distinguish it from a phone call. The user was able to adjust which tone was used, or if there was even need for such a warning tone. This method could also be developed even further by enabling the user to set warnings for only interactions that utilize some specific capability or modality of the entities, such as voice for example. Naturally, these types of tools can improve predictability, but the downside is that these also add extra noise.

Protecting User's Privacy with Proximity-Based Interactions

In [89], Marquardt and Greenberg list one of the main design challenges for proximity and orientation-aware computing environments managing users' privacy and ensuring security. As the users' information gets presented via embedded and other types of interfaces, the users' presence can be used for authentication. However, this does not prevent privacy issues. [89] describes how systems could use RFID technology for defining how far the user is from the interface, and then access for different levels could be granted. The idea utilizes Edward Hall's theory of proxemics; based on the distance of other people and their identities, different levels information could be shown or hidden via the embedded interfaces [10, 47], as illustrated in Figure 3.7. According to [89], these levels could also be used for preventing unwanted interactions in the first place with *proxemic-based safeguards* that only allow sensitive operations to happen when the user is very close. Additionally, [89] suggests utilizing different proxemic dimensions (*position, orientation, movement, identity, and location*) for ensuring what information should be available.

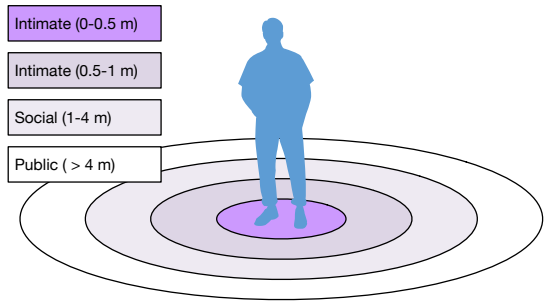


Figure 3.7: Edward Hall's theory of proxemic zones and how they correlate to social zones.

We have also experimented with similar support for detecting distance between entities, but the implementation has been based on Bluetooth technology instead. Figure 3.8 depicts a proximity graph based on measured RSSI values. The different levels in the figure suggest what type of interactions (and thus what content) could be shared while these zones overlap (C.9). Moreover, the figure depicts how a wizard asks the user for parameters when the user starts an application. The wizard can, for instance, be used for defining how far the user must be from an entity in order to trigger a specific interaction between them. Thus, the user can define how socially the entities are allowed to interact. The wizard also helps the user to identify the current environment by suggesting the nearby and previously used entities for different roles in the interactions (C.10).

Other persons from our CoSMo research group have studied similar approaches of mapping the physical distance to different proxemic levels and then, based on this, revealing different social media and other content for co-located people with their CueSense application [66]. The results of this study should be considered while developing proximity-based interactions that share content from social media.

Indeed, *position*, *orientation*, and *movement*, would be interesting dimensions from the privacy perspective. However, whereas Marquardt and Greenberg in [89] were able to test their system in a fixed space, using similar approaches in a dynamic space, and with ad hoc interactions that can be defined in different ways beforehand, would not work for couple of reasons: Firstly, as was already discussed above, even detecting the companion devices and other entities have some of major issues. Secondly, in addition the issues mentioned above, we have also encountered issues in detecting stable RSSI values in a space where multiple Bluetooth enabled entities are present. Whereas this can be improved, for instance, by filtering clearly false values and by calculating moving average values (as we deal with the issue presently), this makes detection slower. This further has an influence on detecting position, as the position needs to be calculated based on movement. Based on our experiences, even detecting *the topology of the entities* from stable RSSI values can be challenging as the end result can be mirrored. For this reason, in our CarGame interaction, where each device's screen shows only a part of a race track, the user must manually set the topology of the participating entities as illustrated in Figure 3.9.

Control Over the Entities

Complete predictability is hard to achieve, especially when the whole idea is novel, and the support for these types of interactions is still in an early stage. Hence many interactions

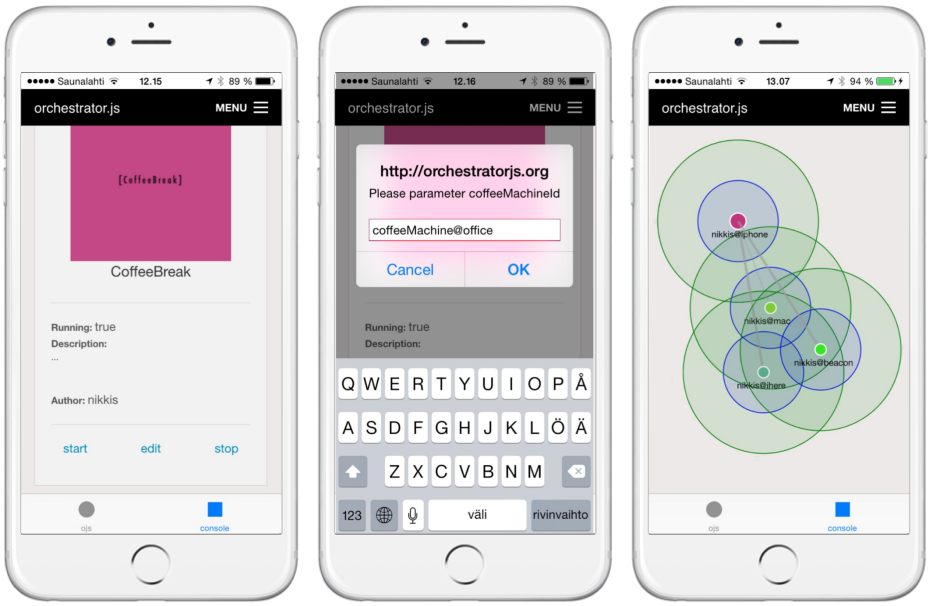


Figure 3.8: Starting an app on Orchestrator.js Framework for iOS.

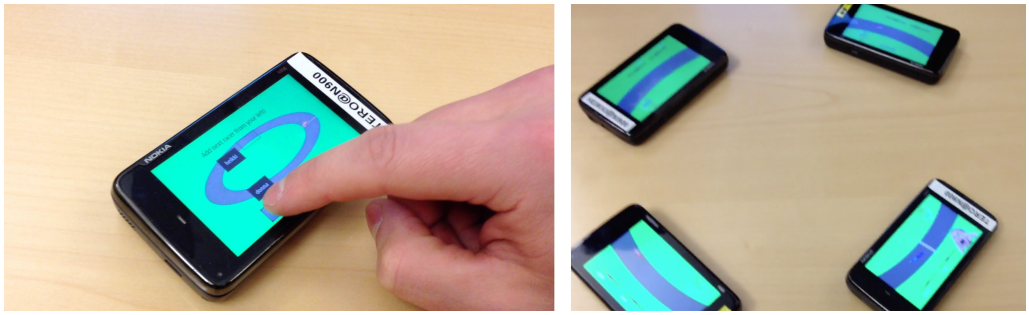


Figure 3.9: User is setting the topology of entities in CarGame manually.

will get triggered by interpreting the current environment incorrectly, for instance due to false input data. For this reason, it is especially important to offer an *easy way for users to cancel any interaction*. As many of the entities do not have an interface, the actual difficulty is in commanding the interactions. From a trust and predictability perspective it is important that the user feels to be in control of the entities, and is empowered to stop any interaction immediately (C.11).

For this purpose, in a perfect setting, the user could simply perform an inverse action for *undoing* activities as proposed by Marquardt and Greenberg [89]. Unfortunately, detecting the changes in the different proxemic dimensions is highly demanding, and therefor other means are required. Again, the companion device can serve as a tool for canceling interactions, as its job is to represent the user. In the Social Devices framework we implemented a *kill switch* for any ongoing interaction that the user was participating in: By shaking the phone, it sends an authorized command for the interaction coordinator to immediately end the interactions on behalf of the user. This kill method also closed any screens and audio players on all the entities related to the killed interactions. The problem of this method was that the user had to take the companion device in hand, and then start shaking it, which took too much time. For this reason, giving such commands should be more nimble. As an example, Microsoft has patented a method for controlling audio signals by whacking a mobile device [85]. This type of a method could enable giving commands and feedback to the system more instantly, and without taking the companion device out of the pocket for instance.

With similar gestures the user could also give parameters for the kill command, for example, to prevent the interaction from being triggered for the next two hours. Moreover, after a couple of seconds killing the interaction, the user could send feedback to teach the system and improve the triggering in the future. For example, one whack could then mean that it was triggered in a wrong context, and two consecutive whacks that it was a wrong interaction for the user in the first place, and should not get triggered anymore.

In general, the issue with a co-located but distributed system like this is that also the control is distributed, or, at least many of the entities may be operating in an autonomous way. Thus some operations may then be hard to end. Unfortunately, there are not many other options for supporting predictability from the control point of view than simply ensuring that the communication enables an immediate way of discontinuing the interactions and, in case of connectivity loss, ensures a proper error handling for ending the ongoing interactions.

Chapter 4

Seamless Communication and Social Sharing of Resources

At present the Internet covers nearly all areas of human habitation. This makes it the primary platform for communication. However, the ever-increasing diversity in the physical dimension means growing heterogeneity in the cyber dimension. Thus we are now living in a world of heterogeneous networks. At the same time there is a growing demand for collaboration of entities which bids for a common language and platform for seamless interactions. Indeed, plenty of technologies have been developed to enable Internet-based communication. These protocols do excellent jobs in supporting remote activities. However, in computing environment today a large part of communications will take place right next to us. Thus we become constantly connected to the entities in our surroundings, and for this reason more direct communication between these entities is required. Direct communication can minimize the lag, and help to improve other elements of communication, such as security, and help to support proactivity of interactions.

Although many connectivity types now support direct entity-to-entity communication, these solutions are underrepresented. Forming connections between devices can be a cumbersome process. Moreover, direct communication is missing similar developer-friendly protocols that exist for Internet-based communication. The protocols are low level, meaning that a lot of responsibility is left for developers. This makes it hard to utilize the heterogeneous networks, and communications need to be implemented separately in an application-specific fashion.

In spite of the missing models for device-to-device communication, communication between two similar entities can still be implemented with some effort. However, when heterogeneity grows, so does complexity. Whereas manufacturer support and ecosystems may offer their solutions for the communication between different entities, communication with outsiders still need to be done with open standards. Thus, with a diverse set of entities it becomes essential to consider the interoperability of the networks as well as interoperability of the communication protocols to implement seamless interactions with any entity.

For the above reasons, utilizing heterogeneous networks to support interactions is a challenging task; Direct device-to-device communication technologies only work with some entities, and are applicable only for some types of interaction. This leaves plenty of gaps that Internet-based communication can help to bridge. This chapter addresses challenges for reconciling Internet-based communication and direct device-to-device communication.

In this thesis the combination of the two is simply called *heterogenous networking*. Whereas at present the role of the Internet-based communication through a centralized service is major, developing the communication platform can be considered as a continuous process which aims to minimize the role of Internet-based communication – The goal is to decentralize the cloud and distribute tasks among the entities. The chapter summarizes the research carried out in the listed publications for improving interoperability of different communication technologies, and for implementing communication for interactions.

4.1 Alternatives for Communication

Entity-to-entity communication can be implemented either directly between two devices, or through a remote service. At present, however, direct device-to-device communication alone cannot be applied for all interactions due to inadequate hardware and software support. For this reason, the Internet is still the main communication platform, and in the future will continue to bridge the gaps that device-to-device communication cannot cover. As there are many ways to implement communication, the term Internet-based communication is used in this thesis as an umbrella for all communication that takes place via Internet, typically via remote communication service. The following concentrates on giving an overview of the most commonly used protocols and architectural styles for implementing Internet-based communication.

Internet Protocol, IP, forms the basis of the entire Internet, and therefor is essential the communication. On top of IP, Transmission Control Protocol, TCP, is typically used as it offers a reliable means of relaying data between communicating entities. Another commonly used protocol is User Datagram Protocol, UDP, which includes benefits of lighter-weight communication compared to TCP due to its asynchronous nature.

Protocols for Web Services

Representational State Transfer [33] architectural style, commonly known as *REST*, has been the de facto way of implementing Web services and especially their application programming interfaces, API. It has replaced XML-based Simple Object Access Protocol, *SOAP*, [137] communication in many cases, as it offers API abstractions on a level that is intuitive for developers to understand if the developer knows the following four main principles of REST: The first principle suggests that all the resources need to have a unique identifier. The second principle, statelessness, prescribes that a server cannot save resource states. Instead, the state needs to be transferred to the client as the name representational state transfer suggests. The third principle, connectedness, proposes that the resources need to be connected to each other, for instance, by offering links to other resources. The fourth principle requires that all resources need to have a uniform interface through which they can be referenced. As these are design principles, they leave room for interpretation, and thus many services are actually called RESTful Web services, since they are not following these principles by the book [105].

Although REST is not tied to any communication protocol, another likely reason for its success is that Hypertext Transfer Protocol, HTTP, protocol offers a good means for fulfilling all the design principles: the resources can have uniform and hierarchical interfaces where all the resources have unique URIs, and which other resources can use to fulfill the requirement of connectedness. The resource states can be transferred inside the request and response bodies. Together with the uniform interface HTTP methods GET, POST, PUT, and DELETE form the concept of RESTful API that nicely maps to the

CRUD operations (create, read, update, delete) familiar from SQL databases, and offer the possibility to negotiate a suitable presentation for each resource.

Bridging Networks

Lately, REST APIs have also started to emerge in embedded and smart home electronics. For example, Phillips HUE lights bridge offers a RESTful API (`http://www.developers.meethue.com/`) for controlling smart light bulbs as shown by the following example HTTP request:

```
PUT /api/newdeveloper/lights/1/state HTTP/1.1
Host: 192.168.1.68:80
Accept: application/json
Content-Length: 45
Connection: keep-alive

{"on":true, "sat":255, "bri":255,"hue":10000}
```

Although the example above is simple (switches on a smart light bulb number one), it shows how in smart home electronics such bridging can help the entities to communicate: a LAN/WAN-connected bridge unit mediates messaging to the bulbs via ZigBee networking technology, which otherwise has poor support in today's devices. In distributed communication approaches, a remote communication service can mediate messages to entities that do not support direct device-to-device communication.

Looking more closely, the example also shows another important point: the HTTP request took place within the LAN, as the bridge allows communication from WAN only for applications that have separately been registered. This means that within LAN, applications need to discover the bridge. For this task the bridge uses Simple Service Discovery Protocol, *SSDP*, that is commonly utilized by Universal Plug and Play, *UPnP*, (`http://www.upnp.org`) devices. Different implementation for essentially the same idea of detecting the entities is called by many names, such as linking service [102], or lookup and discovery infrastructure [45]. In decentralized communication approaches, similar service is needed for offering information about the entities, and how these entities can be connected.

Two-Way Communication

The problem with HTTP-based communication, and hence typically with RESTful interfaces, is that the communication needs to be initiated by the client, and the server cannot really push data to its clients. This forces the client to poll the server at regular intervals to receive updates. This defect makes it hard to utilize REST in programmable interactions, since each entity should be able to initiate communication with others (P.1). However, despite the limitations of REST and HTTP, it has still been utilized for implementing communication for heterogeneous networks. For example, Guinard has introduced the Web of Things (WoT) platform [45], which allows defining and running IoT mashups within a Web browser, and the actual communication with the entities takes place via a RESTful cloud service. In WoT, the Web-based clients send HTTP requests to the RESTful service, which further translates the messages and forwards them over TCP/IP.

Comet [23] is an umbrella term for a set of technologies that enable Web servers to push data to Web browsers on top of HTTP protocol. In Comet, the client maintains a long-lived HTTP connection to the server, which returns a response immediately after new data is available for the client. REST and Comet-based communication was used for implementing communication for an SDP proof of concept implementation for Social Devices. In SDP, the cloud sent remote commands to the entities via Comet-based message bus [36] implementations. However, maintaining a Comet-based connection unnecessarily consumed the resources of the entities, and the connection was hard to maintain by platforms with limited resources (e.g. Arduino) as described by Kelloniemi [74].

Other solutions for complementing HTTP-based REST to enable server push have been proposed, for example, by Stirbu and Aaltonen with their REST Observer model where the changes are streamed through long-lived HTTP connections to clients without dropping the connection in between [123]. The need for server push in HTTP has also been noted by the working group of the upcoming Hypertext Transfer Protocol Version 2, *HTTP/2*, [61] will allow pushing data from server to clients.

Message Passing

Message passing is a common architectural style for implementing asynchronous two-way communication between entities. An example following this style is Extensible Messaging and Presence Protocol, *XMPP*, that is an open protocol for near real-time and request–response services streaming XML messages [62]. XMPP and its extensions provide support for establishing presence, authentication, one-to-one and one-to-many messages and Publish-Subscribe [93]. XMPP technologies have been used for building systems for instant messaging and lightweight middleware [110]. The connections in XMPP are built on TCP/IP protocol, and it enables bi-directional communication between the entities. XMPP follows client-server architecture where the clients communicate with a server component named provider. The providers, on the other hand, can communicate with each other, relaying the messages with each other and then to their clients. As a result, the scalability of XMPP architecture is very good. XMPP, however, has been criticized for the overhead in the communication due to its XML-based messages. The XML-based messages are typically also hard to parse by entities with limited resources. Moreover, XMPP requires setting up and maintaining its own messaging server which the clients need to register.

Protocols for Constrained Devices

MQTT is an open message passing protocol (`http://mqtt.org`). The communication follows a Publish-Subscribe design pattern [30] in which the clients send messages to server with a certain topic, and the server then passes the messages over to the clients who have subscribed to those topics. MQTT is well-suited especially for one to many, and asynchronous messaging. The protocol has become common especially in machine-to-machine messaging due to its lightweight and extensive client support. Although MQTT is a very lightweight protocol, it uses TCP as the underlying protocol which means that even the most constrained devices need to maintain a connection constantly.

Constrained Application Protocol, *CoAP*, [21] is another protocol targeted to constrained devices. CoAP uses UDP protocol which makes the communication more lightweight compared to MQTT. It offers the same method as HTTP (GET, PUT, POST, and

DELETE) for handling the resources in a RESTful way, and for this reason CoAP can also be mapped to interoperate with RESTful Web services.

Sockets

For a long time, one of the most efficient protocols for transmitting messages between entities has been sockets. A socket is an abstraction that describes an endpoint of communication between processes. This means that a process can transmit messages to another process that has bound its socket to some agreed port. Typically, however, the term socket relates to the Berkley socket, which is a standard communication library commonly supported by all modern operating systems, and which offers a standardized API for application programs to utilize TCP and UDP networking. Sockets offer efficient ways to implement application-specific communication protocols as the overhead is quite minimal. On the other hand, the low-level socket API typically requires adding application-specific layers to enable easier usage. [22]

WebSockets [60] offer long-desired server push features for Web applications. The WebSocket protocol itself is completely separated from HTTP, except the need for connection initialization with a handshaking HTTP request. Otherwise the communication happens on top of persistent TCP/IP connections that offer near real-time communication and streaming features similar to XMPP and Comet-based protocols. In contrast to Comet, WebSocket is a standardized protocol [60] and also the Web browser API is under standardization by W3C [136], which indeed is the biggest advantage of WebSocket in comparison to Comet and long-polling solutions. Moreover, the bi-directional connection allows Web applications to communicate more intensively and directly with servers which, on the other hand, decreases the need for constantly loading entire HTML pages. The protocol also supports sending and receiving messages simultaneously, which may not be possible with all the Comet and long-polling solutions. Although WebSockets are intended to be used in Web browsers, many open source projects have implemented WebSocket protocol as a library for different platforms and programming languages. This, in a way, brings Web applications and traditional native desktop/mobile applications closer to each other, as now the server can offer only one API for both of the application types. However, the communication is not as efficient as with plain TCP sockets.

Event-based Communication

Socket.IO (`http://socket.io`) is new unofficial standardized [119] protocol for event-based communication between client and server. The communication mainly relies on WebSockets. However, the aim of Socket.IO is to offer multi-transport support, meaning that protocols like xhr-polling, xhr-multipart, flash socket and jsonp-polling can also be supported by the server. This is useful, as the application programmers do not have to focus on the underlying protocol since relaying the messages is done similarly with each protocol. Other features are: disconnection detection, reconnection support with message cache, multiple sockets (endpoints in other words) under the same namespace, and optional acknowledgements of the relied messages. The connection is initialized in a similar way to WebSocket by making a handshaking HTTP request to the Socket.IO server. As a response, the client receives session-id and a list of supported transport protocols. After this, the client can initialize transport connection by using one of the supported protocols (like WebSocket) and the session-id in the URI. When the transport connection has been established, the client can connect to a socket. Socket.IO protocol contains the seven predefined message types which are presented in Table 4.1.

Table 4.1: Socket.IO message types (version 4).

0 – Connect	The client connected to the endpoint.
1 – Disconnect	The client disconnect from the endpoint or from the whole socket.
2 – Event	Event received.
3 – ACK	Acknowledgment messages inform that a sent event has been received.
4 – Error	Contains information about contingencies.
5 – Binary event	Contains non-JSON data.
6 – Binary ack	Similar to ACK but the arguments contain binary data.

All the messages are based on JSON, and thus lightweight and simple to parse on most platforms. The message type 2 allows developers to define their own event types easily. Events messages contain a JSON array with a list of arguments of which the first one is the name of the event. The problem with non-standardized protocols is that they may be changed, as has also happened to Socket.IO over the years.

4.2 Heterogeneous Networking in a Socio-Digital System

The today's computing environment offers various connectivity types that can offer substantial improvements, and make communications less dependent on Internet connectivity. Different technologies have their characteristics and are designed to be utilized for different purposes. However, progress does not come without early stiffness, and despite the many options for implementing communication, these new technologies have not been put into wider use. Utilizing new technologies is not always straightforward, and issues come up especially when implementing communication between different hardware platforms.

For leveraging the full potential of the cyber dimension, the interoperation of different networking technologies must be supported. Interoperability, however, requires a common platform for bridging the gap between different communication technologies. In device-to-device communication, the entities typically initiate and establish the connections to other entities, and for this reason require information about other entities and how these entities can be connected. For accessing such information, a common connectivity is required. On the other hand, many entities do not support direct device-to-device communication, and Internet connectivity is essential for leveraging these entities. Thus, although the computing environment today is rich in communication technologies, Internet connectivity is actually becoming ever more important as a common denominator.

In this thesis, the term *heterogeneous networking* is used for describing communication infrastructure where the Internet-based communication assists interconnecting the entities, as well as supports mediating messages between them. Socio-digital system, on the other hand, refers to a system like Social Devices where the entities are interacting with each other and humans. The following lists challenges for implementing a communication framework for communication in such system:

C.12. Support for unique entity identifiers. The communication framework must allow referring to all entities with unique identifiers and in a uniform way. Although communication protocols have identifiers, their formats vary and they are not necessary unique in the scope of all networks. For this purpose, the

communication framework must unify the way how different entities are referred regardless of the connectivity or protocol used.

C.13. Support for connectivity. The communication framework must support both indirect and direct device-to-device communication. Thus the framework must offer support for the entities to initiate the communication and set up ad hoc networks when required, but also support mediating the messages between entities through the cloud whenever that is preferred or required.

C.14. Support for protocols. The communication framework must support the most commonly used communication protocols to leverage the most diverse set of entities as possible. Moreover, the communication framework must allow the addition of new communication protocols for extending support and leveraging new entities.

C.15. Support for uniform messaging. The communication framework must allow mediating messages in as unified a way as possible regardless of the underlying network and communication protocol. Furthermore, the communication framework should not be limited for implementing only synchronous (e.g. remote method invocation) or asynchronous (e.g. event-based) communication, but allow them both.

C.16. Support for ecosystem interoperability. The communication framework should prevent silos and support interoperability by allowing communication between ecosystems as well as between other middlewares and services.

C.17. Support for proactive connectivity. The communication framework should support proactivity by connecting to the entities beforehand. This helps the interactions to take place more instantly, and allows discovery of what options for interactions are available.

C.18. Support for security and privacy. The communication framework should support protecting users' security and privacy while connecting to the entities. The trust aspect must be considered while making the decision which entities it is safe to connect and the user should be made aware while connecting to unknown entities. Moreover, the user must be empowered to disconnect from unwanted entities at anytime.

C.19. Support for reconfigurability. Despite the challenge of (C.17), the communication framework must offer an easy way for reconfiguring connections, and allow developers to choose the best connectivity and protocol for their purposes. Despite the challenge of (C.18), users must be able to connect to any entity they wish.

The above list represents key challenges related to communication for supporting programmable interactions. However, implementing a communication framework that can respond to even these challenges is highly demanding. The following introduces how interactions with digital content and physical entities can be supported.

4.2.1 Framework for Social and Proactive Ad Hoc Interactions

The communication in Social Devices has essentially been Internet-based as reported in Publications [IV] and [II], but from the beginning we have also studied and experimented with how to reconcile direct device-to-device communication to indirect Internet-based communication as has been reported in Publication [III]. Bluetooth technology has always been one of the key building blocks of Social Devices for detecting proximity. Thus we have naturally studied how it could be utilized for communication as well. Publication [III] depicts a communication framework where the devices can connect to each other directly with Bluetooth, but also communicate remotely with other entities with Internet-based communication service. The approach has later been developed further as reported in [87].

The framework offers a *uniform communication layer*, which enables mediating events and synchronous method calls between the entities, despite whether they are directly connected via Bluetooth Low Energy (BLE), or indirectly connected via the Internet (C.15). A remote service maintains a registry of the entities, *State Registry*, which allows referring to all entities with unique identifiers which are in a human-readable format: <user name> @ <device name> (C.12). The user name must be unique within the scope of the whole system, and the device name unique in the scope of the user name. This practice, however, may become impractical as the number of entities grows.

The State Registry maintains information about the entities' connectivity support, like Bluetooth MAC address and BLE service identifiers (C.13). The communication framework then utilizes these identifiers for discovering entities and establishing connections between them. On the device-end, the framework maintains information about established connections, and then mediates messages by preferring direct device-to-device connection. As the communication framework stores the information about the entities locally, connections can also be established while the entities have no Internet connection available. The State Registry allows storing any type of meta-information about the entities, and thus can be extended to support other direct device-to-device connectivity types, such as Wi-Fi Direct for instance (C.14). Naturally, however, this requires implementing support on the device end of the framework for each platform that is able to support the new connectivity type.

Supporting the Interactions with Socio-Digital Topology

The Social Devices and Internet of People concepts are both targeted to support interactions which take place directly in our surroundings. Together they set four principles for interactions: be social (P.1), be personalized (P.2), be proactive (P.3), and be predictable (P.4). These principles have an influence on further communication.

As humans are creatures of habit, the social dimension may offer valuable input based on users' social relationships and habits. For example, it could be argued that people are more likely to interact with familiar entities, referring to the entities that belong to themselves or their friends and family. Furthermore, it could be argued that people are more likely to continue interacting with entities with which they have previously allowed interactions to take place, instead of disconnecting or stopping an interaction. However, the case may not be that straightforward, and the willingness to interact may actually depend on the use case. For example, people who are traveling may actually be more willing to interact with unknown entities.

Nevertheless, social dimension data can be used as a basis for establishing connections proactively (C.17). The interactions can then take place instantly as time is not wasted

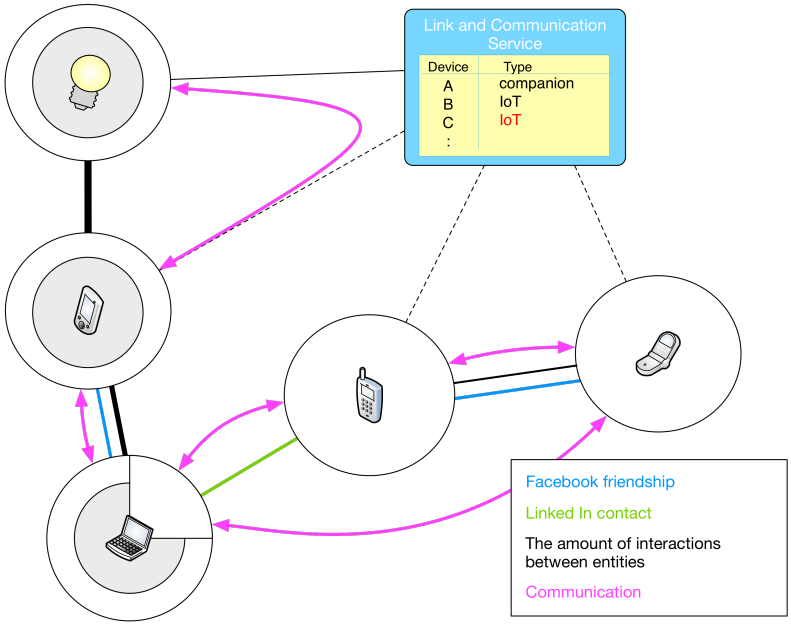


Figure 4.1: Communication topology in a Socio-Digital System.

on discovering and establishing the connections while an interaction has been triggered. As was reported in Publication [III], time wasted on forming device-to-device connections is one of the major defects in proactive interactions compared to Internet-based communication. The same information from the social dimension can also be used for considering to which entity it is safe to connect (C.18). For example, users may typically trust their friend's entities. On the other hand, exploring the history data may offer valuable information about unknown entities. For instance, if many have blocked connecting to an entity it probably should not be trusted.

In this thesis the term *socio-digital topology* is used for a group of co-located entities that are interconnected based on social relationship and history data. One of the ideas is to estimate to which entities the users are going to interact and connect proactively. These proactive connection establishments also support social interactions.

Towards Proactive Ad Hoc Networks

Forming socio-digital topologies is the responsibility of the communication framework, and so it must consider the peculiarities of each connectivity type. As an example, in Bluetooth Low Energy, discovering and establishing connections is based on universally unique identifiers (UUID) [95]. However, some implementations of BLE tend to hide the identities of unknown entities which makes it more complicated to identify nearby entities. BLE allows defining its own services with specific identifiers which can then be discovered by others. Thus, in essence, the nearby entities need to know who they are looking for (UUID of the discoverable entity) in order to connect specifically to that entity. Conversely, the entities can first connect to an entity and then ask them to identify themselves, as BLE allows connecting without pairing. However, this may expose them to threats.

The State Registry offers valuable assistance for this type of paring process by applying information available from the social dimension for forming the connections. We presented this type of approach in Publication [III]. In the implementation each entity capable of connecting directly to other entities downloaded a list of interesting entities. Moreover, later the implementation was complemented to stay up to date by requesting updates at regular intervals when an Internet connection was available. The list contains identifiers of the user's own devices which are then used for discovering the entities. Although the list only contains user's own devices, it could easily be extended with identifiers of the user's friend's and family's devices, as well as with information about which devices the connections should be maintained to support the challenge of proactive interactions (C.17).

In our earlier experiment with classic Bluetooth socket-based communication, one of the difficulties again was forming the connection between two phones as this type of connectivity requires pairing the devices. The State Registry can store information about which devices have been paired, and later use this information while selecting the connectivity type. The paired devices also hint that that they can be trusted in other types of communication as well, like the remote communication via Internet.

Similar to Bluetooth, Wi-Fi Direct connectivity also requires that devices have accepted the communication and formed a group. As the pairing seems to be the key issue in device-to-device communication, the State Registry could help to suggest which devices should be paired, and use the Internet-based communication between the entities that only rarely need to communicate.

Distinctions to Opportunistic Mobile Social Networks

Typically ad hoc networks formed based on social network information are called Opportunistic Mobile Social Networks, often referred as Mobile Social Networks, or simply MSNs [46, 73, 144]. However, there are essentially three distinctions between the implemented framework and MSNs. Firstly, MSNs are delay-tolerant networks and not targeted for rapid and continuous end-to-end networking. Programmable interactions, on the other hand, require as direct end-to-end connections between the entities as possible to support as fast communication as possible for interaction purposes. Moreover, the communication in MSN follows the principle of store-carry-forward [73], meaning that a message is traveling with the users and via their devices until the message reach the target device, which requires advanced routing [144].

Secondly, the idea of programmable interactions is not to form social networks as such, and thus communication is not as coupled to human social networks as it is with MSN [73]. In a socio-digital system like Social Devices, the relationship in social media is not a precondition for forming the connection, but is rather considered as a recommendation for establishing the connection proactively to support more instant social interactions.

Finally, in the framework implemented for the programmable interactions the communication is rather Internet-based, and communication with remote and previously unknown entities is common. Also, whereas the communication MSN is typically based on advanced mobile technology, such as smartphones [73], the idea in the implemented framework is to form ad hoc topologies with any entities in our surroundings to support programmable interactions. Nevertheless, MSNs are closely related to the implemented framewrok, and the research done for forming connections in opportunistic MSN is very relevant and can benefit programmable interactions, especially its proactive aspects (P.3).

4.2.2 Bridging Content from Cyberspace to Physical World

Today ecosystems of different vendors typically enable accessing digital content within their own ecosystem. On the other hand, other cloud services offer solutions to synchronize data in between the entities in a more vendor-neutral way. Nevertheless, both solutions require copying and uploading all the content to a remote cloud service. Moreover, accessing this content from another entity requires downloading the content. Thus, seamless access always requires a fast Internet connection. This may seem absurd, especially if the two entities are right next to each other as they often are in today's computing environment.

Accessing digital content in a seamless way differs from the communication that aims at coordination of the entities within heterogeneous networks. Essentially the difference is due to the size of the messages. Another difference is that the content needs to be stored, unlike the messages. Despite these differences, however, there are many confluences and heterogeneous networks can also help accessing the distributed digital content.

Publication [VI] depicts a blueprint architecture that is based on the VisualREST distributed content management system. The approach takes advantage of the nature of the content which consists of two separate parts: metadata, and essence [118]. Mauthe and Thomas describe the essence as: "the raw programme material itself, represented by picture, sound, text, video, etc." [90]. Hence, the essence is the core part of the content that is typically perceived as the content itself. From a distributed content management perspective, however, the distinction of the essence and metadata is important. The metadata is used for describing the essence and its representations, like length of a video, or size of a photo. In VisualREST, the metadata is also used for describing the existence of a content item, including its exact physical location, owner, access rights, and so on.

Metadata is the Key

Figure 4.2 illustrates how VisualREST enables devices, middlewares, services, and applications to access content stored within users' devices, cloud services, and social media in a uniform and homogenized way through a RESTful interface (C.12). In VisualREST, only the metadata of the content is stored in the centralized service, and the essence of the content stays in its original location until it needs to be accessed. As the problem with RESTful HTTP-based Web services is that they require the client to initialize the communication to the server, we have studied how Web services can be complemented with architectural styles [88]. VisualREST was complemented with message passing architectural style for sending remote commands to the client, which then utilizes the RESTful API to communicate back to server. The communication in message passing was based on XMPP protocol, but other message passing protocols could as well be utilized for this purpose (C.14). Later on, Publish-Subscribe design pattern was also applied to enable sending one-to-many notifications for the entities about changes in the content (C.14).

One of the key concepts of VisualREST is *containers*, which are like buckets of metadata coming from a certain source, and for a certain purpose. Each user may have several containers for different purposes. Originally, a container did correspond to a single device. However, later it was considered a better solution that a device could hold multiple containers for different purposes. The metadata of a container can come from a *container program*, which is a daemon process observing for new and changed content and running in the background on the user's device. It is the responsibility of these processes to extract as much metadata as possible from the essence and its source.

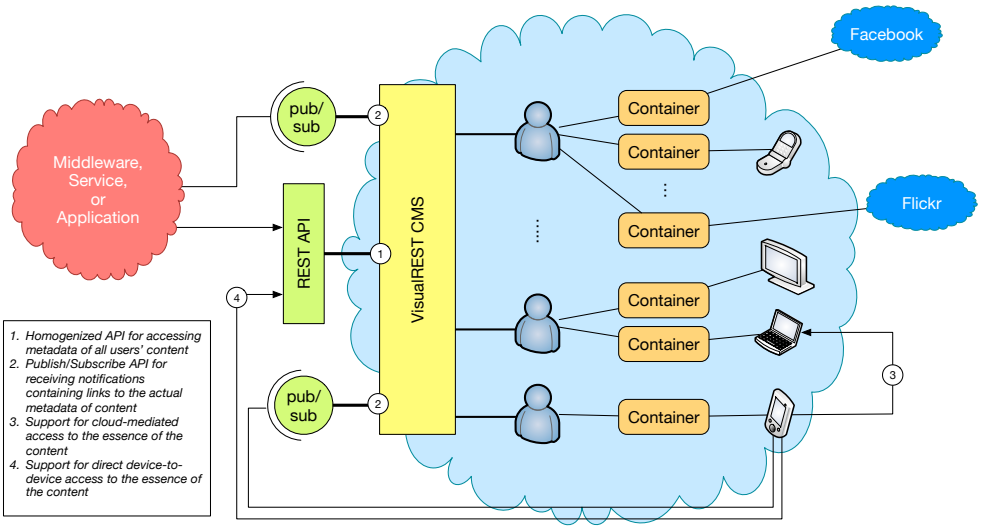


Figure 4.2: Users' content as a Service.

An example container program for Android automatically attached all the information about the device's context as metadata to the created or changed content. For example, when users created or edited any document, it was automatically tagged with the place where it was created. Moreover, a client named Context Camera was later created, that allowed users to predefine a *context* for the content in a half-automated way to attach as rich metadata as possible to the images taken. This metadata included physical place with city names and a set of tags. Afterwards, the context enabled an easy way to find content and share it with others, and to receive notifications when new content was updated to the defined context.

As a lot of user's content today is also stored in different cloud services and social media, VisualREST was complemented with features that allow users to link their cloud services, email, and social media accounts to be utilized through a uniform interface in a controlled manner [100]. Essentially the idea is that a daemon process observes social media, cloud services, and email accounts, and extracts meta data of the content. Thus VisualREST also helps improve interoperability of the cloud services (C.16).

In VisualREST, the server's role is to manage access rights to the content, which is done with *user groups*. One can create as many groups as is necessary, and then add other VisualREST users to these groups. The aforementioned concept of contexts not only automate the processes for adding metadata, but also adding the access rights: Users can create contexts and then post content to a specific context. This adds or changes the metadata of the content accordingly. Moreover, the contexts can also be used for observing when new content matching a predefined query gets added or updated. After detecting changes, the *context query* pushes a notification about the event to a related publish/subscribe node. This makes it unnecessary to poll the REST API constantly for updates. The notifications only contain links to the actual metadata, and REST API is then used for getting the new or updated metadata as well as the essence.

Accessing the Essence

There are various options for establishing the connection for accessing the essence of the content. The key is the information available from the metadata of the content as it contains the identifiers and the physical location of the content, which means that the two entities can negotiate the best connectivity type for accessing the data. As an example, Wi-Fi Direct enables fast transfer rates directly between entities that are co-located and support that connectivity. In the first implementation of VisualREST, the devices hosted their own Web servers for providing content. A disadvantage of this approach was that firewalls were blocking the communication from WAN (i.e. between entities that were not operating in the same LAN). However, this approach could still be utilized in co-located situations where the entities are connected to the same LAN. For remote access, VisualREST enables the entities to mediate the content via the Internet service by utilizing HTTP protocol. This also allows lightweight Web-based clients to access the content and supports interoperability (C.16). For example, VisualREST's potential to support building Web mashups was studied in [91].

If a VisualREST client maintains the metadata of the interesting content, it can naturally also operate and access the essence with direct device-to-device communication while no Internet connection is available. An example of such a client is named MIST developed in Aalto University [104]. This client enabled Nokia Maemo phones to mount the VisualREST content to the phone's local file system with FUSE (`http://fuse.sourceforge.net`). Thus the main idea resembled cloud storage services like Dropbox (`http://dropbox.com`), for instance. However, unlike with Dropbox and other similar services, the essence was not copied and synchronized to the local filesystem. Instead, only the essence that the user actually accessed was copied to the filesystem. This saved a lot of resources, as users nowadays can have terabytes of content, and additionally access to other users' contents as well. The client also supported mounting more complicated file structures defined with context queries: Whereas the typical cloud synchronization services enable sharing folders with other users, the query results of VisualREST may come from multiple users and from varying paths.

Rich metadata can also help to make interactions with content more proactive (P.3), personalized (P.2), and predictable (P.4). Metadata helps to utilize the content and detect in which social environment the content can be shared. For instance, a connection can be proactively established (C.17) as is typically done with opportunistic mobile social networks [46, 82], and even be proactively transferred between the entities based on the social dimension information (e.g. if we know who was present while the content was first created or later on modified). This further improves the proactivity of the interactions (P.3). The metadata may also set boundaries in which the social environment the content can be shared, and what connections should be established (C.18), which can further improve predictability (P.4). We may also have already shared the content with someone by email, for instance, and later in a meeting a fresh new version the same content inside a work email container can be updated for the participants. As another example, the content inside a Facebook friends container can be used while we are co-located and our devices are interacting with the same people with whom the content was shared in Facebook.

Chapter 5

Empowering Developers for Programmable Interactions

We are heading towards a world where silicon is everywhere. Hundreds of millions of computing devices can be found in any place. Mobile devices are relatively few compared with embedded and emerging IoT devices. Many of these provide their application programming interfaces so that the technically-skilled can develop their own applications for controlling their physical space. If one cannot find an appropriate device, gadget, actuator, or smart home component from a shop, it is simple to put ready-made pieces together. Platforms like Arduino, Raspberry Pi, .NET Gadgeteer, together with hundreds of sensors and servo motors offer unlimited possibilities. These platforms are especially suitable for experimenting and building prototypes [55, 135], and together with the emerging 3D printers can help encapsulating the hardware into refined, usable objects [112]. Prototypes with potential can then later be turned into commercial products by hardware manufacturers. Eventually, however, one will be able to print complete devices at home, which will revolutionize the way hardware is produced and lead to even more massive numbers of physical objects.

Programming physical devices today is an enjoyable task, thanks to the great IDEs and SDKs. However, when it comes to building multi-device applications, developers are often forced to invent their own concepts and abstractions: current tools lack support for making applications where devices of different types can co-operate towards a common goal. Moreover, abstractions that are essential building blocks of designing interactions are missing. Instead, developers have their own concepts and implementations for coordination logic, communication, passing messages, calling remote commands, distributed application logic, and so on. The lack of a common language makes it harder to communicate one's ideas and more deeply understand how a multi-device application should actually be built or which part of the system should be doing what. According to Lee, for example, we will have to rebuild computing and networking abstractions in order to understand the full potential of cyber-physical systems [80]. Thus, more concrete abstractions are needed for defining applications for heterogeneous cyber-physical environments, which is the basis for today's computing.

Indeed, with the current communication-oriented abstractions it can be straightforward to get a few entities to interoperate. However, the diversity of devices in cyber-physical applications is typically high and continues to grow. Therefore, applications are in danger of becoming too complex to program and manage. Lee claims that the abstractions will

have to embrace physical dynamics and computation in a unified way [80]. Consequently, it can be argued that a full programming model with proper abstractions for handling all the complexity is needed.

This chapter introduces the research carried out for developing new programming model for the modern computing environment, which we have named as the Action-Oriented Programming – *AcOP* offers high-level abstractions for software developers to build and understand interactions between multiple computers operating in the modern computing environment, taking also the human and social aspects into account. It is a tool that enables developers to focus on the functionality and development process itself, and not to reinvent concepts and abstractions. From an interaction perspective, the runtime and coordination model are essential for operating in the cyber-physical environment, and hence needs to be kept in mind by software developers. Although in AcOP the coordination of the entities is built into the programming model, the coordination is also always visible to the developers when that is required.

5.1 Motivation for the Approach

Over the years many approaches have been taken for offering tools that help programming distributed applications. Java remote method invocation (RMI) is probably the most well-known attempt. RMI offers a mechanism for calling a method of another Java program running remotely [22]. The issue with RMI was that it was tied to the Java programming language, and thus to its virtual machine environment. Another well-known attempt is CORBA (Common Object Request Broker Architecture), which has the advantage of language independency [22]. In CORBA, the remote interfaces are defined with an interface definition language named CORBA IDL, which remote clients can then use for invoking method calls similar to Java RMI [22]. Whereas CORBA as well as Java RMI offer mechanisms for describing what kind of tasks the entities are capable of performing, and then calling their interfaces remotely, the solutions still lack support for defining the interactions themselves. Proper abstractions for defining how the different entities should interact and when these interactions should take place are missing.

Ideally, from interaction and interoperability perspectives, operating systems and network infrastructures would have built-in, open-standard support for providing the resources of the computing environment as understandable and programmable components. Such components could then be utilized for defining the interactions between other entities nearby, and possibly, also with humans. Yet this type of support is missing [78], and the closest approach is services provided by some networked devices. Consequently, many hardware manufactures focus on implementing their own ecosystems where their devices interoperate. Also, some more vendor-neutral approaches exist: UPnP (Universal Plug and Play) is an initiative from industry to improve interoperability between devices from different vendors by providing services [58]. For instance, devices may offer their audio services with Bluetooth Advanced Audio Distribution Profile (A2DP) [58], or video beaming with DLNA [133].

In general, the problem of services is that they are too generic concepts for defining the actual interactions; These services are essentially targeted for improving interoperability, not for defining social interactions (P.1) where multiple entities participate and co-operate. Moreover, the services provided by the devices do not really describe how and for which purpose the resources will be used, which does not support the principle of predictable interactions (P.4). Furthermore, the generic nature of these services may also leave the

platform open for possible security threats, since these services do not clearly define for what and in which context they can be used.

Different solutions have also been proposed for defining higher-level interoperability between independently operating services. In such service-oriented architectures (SOA) the services are typically described with service description languages, like WSDL (Web service description language), and mechanisms like universal description, discovery, and integration (UDDI) services can be used for discovering what services are available. The key idea with comprehensively described services is that those can be composed and integrated with programming frameworks and higher-level tools [22]. For example, Web services business process execution language (WS-BPEL) can be used for composing several services [140]. Moreover, graphical business process model and notation (BPMN) can be used for defining BPEL compositions without writing code [19].

The rich service descriptions can also be a burden as each service has their own APIs and ways of handling operations [96]. Thus, REST design principles and APIs have become the de facto ways for implementing communications with Web services. Moreover, considering that many entities have very limited resources, handling rich XML-based service descriptions might become too heavy. Consequently, other types of protocols have emerged, like CoAP, which follows many REST design principles.

As today's mobile devices have quite adequate computing power, many have studied their capabilities for providing SOA services [13, 63, 64]. Typically, compositions that also employ mobile and other types of devices are called pervasive service compositions [17]. For instance, IBM has taken a graphical approach similar to BPMN and BPEL with their NodeRED platform (`http://nodered.org`) which is targeted to wiring together Internet of Things devices and cloud-based services. However, instead of XML, NodeRED compositions are defined in JSON. Another example of such a graphical tool is Octoblu (`http://octoblu.com`), which is one of the most recent comers to IoT platform markets, and is offering a Web-based wiring tool for building IoT services.

The problem with tools designed for building service compositions, as well as with home automation in general, is that they overlook the social dimension and interactions with the user. Often these systems are based on recipes, or (work) flows, which mean that the user is left to be more of an observer than an active participant. Moreover, often such approaches work one way only, meaning that the user is left without ways of controlling and interacting with the flow, and many times also completely without feedback.

All of the approaches above have their limitations, and thus do not fit well in defining social and co-located interactions. Often the solutions are too heavy, too complex, tied to specific technology, lack support for dynamic ad hoc interactions, and above all, miss proper abstractions. Especially, they all focus on machine-to-machine interactions, but seem to forget the social dimension. These approaches tend to expect that the devices interact in an automated way, and that users are willing to only observe rather than participate themselves. This maintains the gap between computing and humans.

5.2 Action-Oriented Programming Model

Enabling programmable interactions is a demanding task; thus a programming model is needed that fully considers the physical, cyber, and social dimensions. To understand and support the interactions, it is vital that all three of these dimensions are considered equally important. The following addresses a set of challenges for a programming model that

follows the main principles of programmable interactions: be social (P.1), be personalized (P.2), be proactive (P.3), and be predictable (P.4).

C.20. Support for defining interactions. The programming model needs to enable defining interactions between all types of entities, how these entities behave and co-operate. Furthermore, controlling the entities should be independent of the used connectivity type.

C.21. Support for composing meaningful interaction abstractions. The programming model needs to enable composing entity resources to meaningful abstractions that are clear for both developers and end-users. Based on these abstractions, the programming model needs to support describing and controlling the entities in a unified way, and support leveraging their different resources.

C.22. Support for anticipating sensations of the world and reacting upon. The programming model needs to support detecting and reacting to changes in the computing environment. Moreover, the programming model must offer support for users' preferences while defining what type of interactions take place, in which environment, and how the entities behave.

C.23. Support for empowering users. Humans should be an integral part of today's computing environment. For this reason, user experiences should be considered already by the programming model and its abstractions. The programming model should empower the user to intervene in the interactions whenever that is desirable, and always empower the user to be in control over the system.

C.24. Support for detecting contingencies. In a dynamic, distributed environment contingencies are common. The programming model should support managing these, and support recovery from such situations. If the system cannot manage a critical contingency, the programming model needs to support acknowledging the right person.

C.25. Support for distributing tasks. The boundaries of platforms and ecosystems often limit building applications for distributed environments. For this reason, the programming model needs be able to distribute tasks and take advantage of the modern computing environment. Thus, instead of the platform or ecosystem, it is the user who draws the borders that determine where the entities are allowed operate.

Responding to these challenges, the programming model requires clear concepts that abstract the complexity of today's computing environment. The following describes an Action-Oriented Programming (AcOP) model.

As the name of the programming model implies, *actions* are its centerpiece. The two other main concepts, *capabilities* and *apps*, are designed to support the actions. In short, actions and capabilities offer abstractions for programming interactions in a unified way, and the apps an abstraction for defining when an interaction should take place. In this

section the main concepts of AcOP are defined and the choices for developing the model further rationalized. Later in the chapter, the runtime of AcOP is explained and its limitations are addressed.

5.2.1 Actions

The first challenge for a programming model is to support describing the interaction of entities independent of how these entities are connected (C.20). Although the interaction should take place proactively (P.3), these interactions between entities still need to be predefined. Thus, defining the interaction for co-located entities resembles writing a scene for a movie; each entity should have a specific *role to play*. In Action-Oriented Programming these scenes are modeled with a concept-denoted action. Actions can take place in an ad hoc manner with any entities that are suitable for the roles defined by the action. The *participants* of an action interact with each other based on predefined logic, and the interactions with humans take place via the participating devices (P.1).

The Story of Actions

Action-Oriented Programming came into being while we were developing a model for programming interactions in the Social Devices concept. The notion of an action is rooted in the DisCo method [68], which is a formal specification method for reactive and distributed systems based on joint action theory [9]. Unlike in DisCo, however, the actions in AcOP do not have a formal meaning, but instead executable semantics. Moreover, the participants in DisCo actions are typically processes, whereas in AcOP actions the participants are typically, but not always, devices of different types. AcOP actions were first introduced together with the Social Devices concept in Publication [IV], then the idea of Action-Oriented Programming was generalized for pervasive computing in [1], and finally the current concepts of AcOP were introduced in Publication [II].

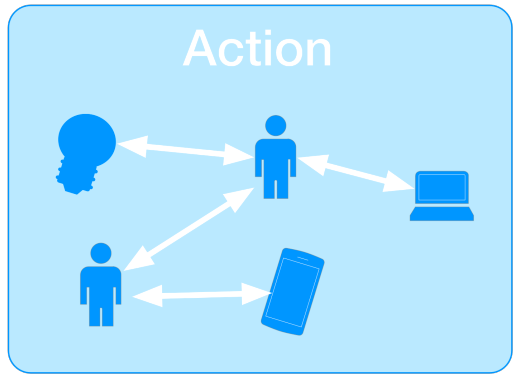


Figure 5.1: Action is a programmable unit for defining joint behavior for a set of co-located entities and people.

Joint Behavior

The idea of Action-Oriented Programming is to offer an abstract model for defining programmable interactions. Thus AcOP is not tied to any specific hardware platform or connectivity type. Whereas many UbiComp systems draw boundaries in the physical

world between different systems and environments [75], AcOP actions are designed to take place freely in any location. Hence, the actions fit well for defining the interactions for cyber-physical systems where diversity and scale are large, and boundaries less limiting. Naturally, however, the developers can define strict boundaries for their AcOP actions, and develop interactions that only take place in specific environments.

Figure 5.1 depicts how actions are abstractions which encapsulate a joint behavior of multiple entities and their users. This kind of joint behavior can be, for example, a photo sharing session between multiple entities, or a multiplayer game that involves and interacts with the environment. An action could also be used for controlling IoT objects. As an example, in a room where gaming takes place, the actions can be used for turning off the lights, or adjusting their color. Thus, the actions support interacting with the environment and adapting it based on the ongoing interaction. This kind of behavior could be implemented as one action. However, it is preferable to split the behavior into several smaller actions that can then be reused in different contexts.

Precondition, Body, and Handlers

Actions consist of two main parts, precondition and body. Additionally, actions can be complemented with handlers for contingencies and input.

A precondition of an action is used for defining and guarding when the interaction is allowed to take place (P.4). A precondition should be implemented as a simple Boolean expression which evaluates to true if the environment is correct for an action. Originally in Social Devices, preconditions were used for constantly evaluating if actions can take place. However, this kind of usage may lead to performance issues, and thus is not recommended. Instead, the precondition should be executed just before the body part, to simply make sure that the state of the devices and the environment is correct for running the action body part. Thus, the role of the precondition is to act as a guard that ensures that the entities are operating in the correct environment and that their joint state is suitable for performing the action.

The body part of an action is used for defining the actual interactions (C.20). The joint behavior of multiple entities is defined with the help of device capabilities. Although Action-Oriented Programming is not tied to any specific programming language, it is preferable to use a real programming language for defining the body. Using a proper programming language allows defining a more sophisticated behavior of the entities compared, for instance, to many service composition languages. As an example, coordination logic for multiplayer games has been defined with an action body in Publication [V].

Contingency handlers offer an understandable way of managing the contingencies without deep knowledge about the remote entity itself (C.24). The idea is that the contingencies, like exceptions or notifications about connection drops are relayed for the coordinator which then invokes a method that tries to recover from the contingency. This mechanism, however, is far from the reliability required from highly critical applications such as healthcare and automotive systems. Thus, this aspect requires more research and testing with less critical applications.

Input handlers may be added to any action, and they work similar to contingency handlers. In the future such handlers could be utilized to support proxemic dimensions [89]. The handlers would then be used for coordinating the action during its runtime based on user presence (C.23). Another application for the handlers could be, for instance, when a

new entity is detected it could join to an already running action, or an entity could be removed in a controlled way from a running action when it indicates willingness to leave.

5.2.2 Capabilities

The second challenge addressed for the programming model of a computing environment that is getting ever more diverse is how to describe and control the entities in a unified way, and yet emphasize and take advantage of the different resources each entity has to offer (C.21). The Action-Oriented Programming model offers a concept named capability for encapsulating a set of resources into a concrete and understandable abstraction that can be programmed. Figure 5.2 depicts how an entity encapsulates the resources, and how it appears to AcOP. Furthermore, the figure implies that these capabilities may change the state of the entity, and that the changes can be reported for the computing environment.

Actions and Roles

The capabilities are used for describing the entities, while alotting to them the roles of an action. While developing an AcOP action, it is enough for a developer to know that if an entity was selected for a specific role, the entity is then capable of performing a certain task. Thus, the developers are freed to focus on the actual interaction development process itself, without the need to focus on how the different resources of each device could be harnessed for the purposes of the interaction (C.20).

From the end-user point of view, the capabilities offer an understandable way of defining what kind of tasks their devices are allowed to perform; end-users should be enabled to install and switch on and off capabilities based on their personal preferences. Similarly, end-users are able to understand what kind of actions an entity can then participate in, as the action roles require certain capabilities (C.21).

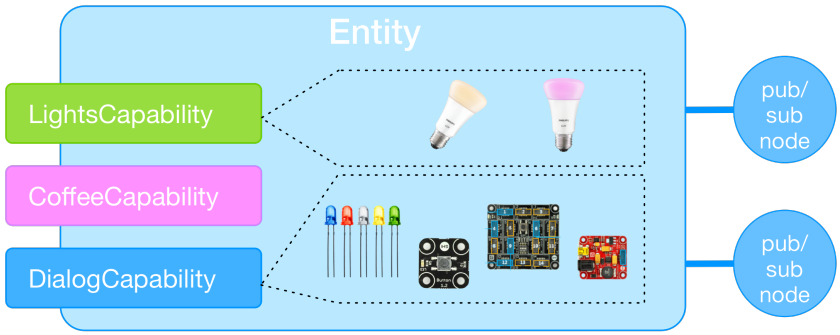


Figure 5.2: Describing an entity with capabilities helps both the developer and the end-user to understand to what kind of interaction the entity can participate.

Appearance

The devices in today's computing environment may be very different by their appearances, and their resources vary accordingly. Some devices are stationary and integrated tightly into their surroundings. Other devices are mobile and their users carry or wear them all

the time. Some devices can be specialized in presenting content, while others may be able to store large amounts of digital content. New types of devices are continually emerging.

Moreover, even a single device may consist of a set of different sensors, actuators, screens, as well as other input and output modalities. Utilizing such a large set of different types of resources can indeed be confusing, especially for developers who have mainly been developing applications that operate within strict boundaries, or co-operate by synchronizing data over cloud-based services. However, concrete understandable units with clear purposes can help to better understand the possibilities of today's computing environment, as well as help to develop applications that contain a large set of different types of entities.

Hooking into Hardware

Whereas the AcOP aims at being independent of any platform or programming language, the concept of capability has two sides in this regard: The capabilities operate closely within the interface between hardware and software, offering hooks to the resources. Thus they may become tied to the platform they operate and its programming language respectively. From the interaction development point of view, however, the capabilities appear the same and in a programming language-independent way. The actual interactions are programmed with the *methods of the capabilities* that are always the same despite the underlying hardware and resources – These methods allow considering the physical objects and their resources as objects familiar from object-oriented programming. From a technical point of view, the capabilities and their methods could be compared to CORBA or Java RMI, for instance, which allow calling a method of an object remotely [22]. However, the abstraction level is intended to be higher, and less technical to better support programming the interactions. Also AcOP is not programming language-specific as RMI is, but an abstract programming model. Thus, on an action-end, the capability needs to appear independent of hardware platform and programming language (C.21).

An entity is essentially allowed to implement the behavior defined by the capability in any way it considers best, and utilize whatever technologies that are considered the best. Hence the same capability may appear differently on separate entities, although the main idea must stay the same. As an example, a dialog capability could be implemented using a touch screen as a graphical user interface, or with physical buttons, LED lights, and handwritten labels. Naturally, however, this capability could then only participate into certain interactions, which is fine since many entities have a certain dedicated purpose anyway. Some advanced capability implementation, on the other hand, could tend to adapt based on the environment in which it is operating. For example, while driving a car, the output could be in audio format and the input based on simple voice commands.

Configuration

As the implementation can vary, it naturally has an influence on the quality of service. One solution for ensuring quality would be to utilize recommendation systems: each capability would then have a quality attribute which would be used while selecting the best candidates to a specific role [98]. Also, history information could be used for this selection task: if a user has not given any negative feedback on a specific device playing a specific role, it could be assumed that the device fits that role. The user should also be able to prefer or enforce a certain device to a certain role. The most important aspect here, however, is that the user always stays in control and is empowered to change the

behavior later on with his or her actions (C.23). This means that the developer must consider the capability also from an end-user perspective. Each capability should have a descriptive name and a clear purpose which end-users can understand – With the capabilities the end-users control to which actions the entity can participate and what roles an entity can play (P.4).

5.2.3 Apps

According to the third challenge, an application running in a heterogeneous multi-device environment should be able to detect the changes in the environment, and decide how to react to these changes (C.22). Apps in AcOP are designed for this purpose. An AcOP app differs from a traditional mobile or Web application in many ways, and above all, is more of an abstraction for controlling the interactions. Unlike typical apps, AcOP apps do not really have any borders as they are not limited to operating only with and on the current platform, but rather enable employing a heterogeneous set of entities to co-operate (P.1). AcOP apps define the environment where the interaction should take place, and define participating entities.

To have more control, the users need to be able to start and stop apps (P.4). Moreover, while starting an AcOP app it should ask users to define their preferences (C.23). Hence, instead of the platform or ecosystem, it is the user who sets the limits of which environment the app can operate (C.25)

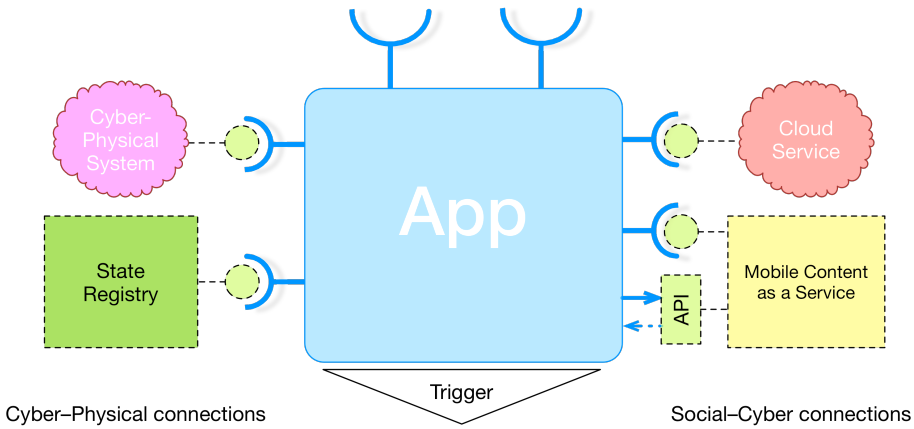


Figure 5.3: Apps have the important task of observing the computing environment, and then based on the changes, schedule an action for a set of entities.

Connections to the Computing Environment

Programmable interactions should be proactive (P.3), personalized (P.2), social (P.1), and yet predictable (P.4). For this reason, a running AcOP app needs to be able to detect changes in the user's environment, and then based on these changes trigger interaction among the surrounding entities, and possibly with some remote devices or services to support more personalized interactions, for instance (P.2). Figure 5.3 depicts an app and its connections to the computing environment. Detection of changes typically follows an *observer design pattern* [36] in which an object (app instance) observes its dependents and

then notifies the object when the state changes. The actual observing can be implemented, for example, by polling remote entities at regular intervals. However, a much more elegant and efficient solution is to use, for example, a *publish-subscribe design pattern*. This type of system is often called a *distributed event-based system* [22].

Towards Lightweight and Flexible Scheduling

Scheduling interactions is a highly challenging task with many aspects. These aspects include: detecting the changes in the computing environment; finding a suitable set of entities; finding an action for the participating entities; allocating the entities; and finally triggering the action at the right moment.

Originally the scheduling of interactions in AcOP was based on a more complex triggering system. Selecting entities for the roles of an action, or the other way around, selecting the roles for currently available entities was done with a service named Configurator as a Service, *CaaS* [98]. However, this type of heavyweight configuration is not always needed, and thus a new concept named app was developed for scheduling the actions. This lightweight solution was introduced in Publication [II]. For more advanced selection of entities and their roles, CaaS can still be utilized as described above. Moreover, if the capabilities are described with quality attributes, using a configurator would then be the preferable tool for the configuration.

AcOP apps do not limit in any way how the entities should be selected for their roles – different types of mechanisms can be used for this task. For instance, apps that are intended for controlling smart home electronics must allow users to specifically predefine devices for the roles. On the other hand, some apps can be limited to trigger interactions to only familiar people, or with people that are friends in social media. Other apps can be more ad hoc by nature, enabling interactions with any entity while the environment dynamically changes. Some apps may simply trigger interactions based on time, date, or location like an alarm clock or a reminder. Also, observing third-party services can be carried out with apps, and when new content gets shared in social media (e.g. a photo album in Flickr), the app could later trigger an interaction to share this content while the user is co-located with other people. Nevertheless, the app developer has an important task of making the scheduling as predictable as possible (P.4), but yet support personalized (P.2) interactions to take place proactively (P.3) and socially (P.1).

The actions and apps should also allow new entities to join already running actions, especially for those types of actions that take a long time to execute. For instance, when new people arrive to a party, a music-playing app could detect this and add them as participants to a running action which picks songs from the participants' playlists to be played through stereos. Naturally, also leaving from such long running actions must then be allowed without needing to stop the execution. Joining and removing entities then also needs to be taken into account by the developer of the action.

5.3 Runtime Implementation for Action-Oriented Programming Model

A complete programming middleware needs a runtime environment implementation which abstracts the heterogeneity of the computing environment [22]. Action-Oriented Programming itself is an abstract model as was described above. The following describes the approaches studied to implement the underlying infrastructure and runtime environment.

Based on the discussion in Publication [III], the following identifies the most important elements of coordination, and addresses challenges for them. The runtime environment acts as a middleman between the communication framework and the programming model, and thus these challenges are linked to each other as well as to some other challenges listed in this thesis.

C.26. Diversity. The runtime implementation should support taking advantage of the diverse and heterogeneous set of entities in the computing environment.

C.27. Seamlessness. The runtime implementation should take advantage of the uniform communication between the entities (C.15), and enable the applications to take advantage of this support, and allow implementing applications whose components can operate seamlessly.

C.28. Scalability. The runtime implementation should not limit scalability of the system, but instead allow the delegation of tasks and allocation of entities within the computing environment.

C.29. Deployability. The deployability of applications directly affects many other elements of the coordination. Thus, the runtime implementation must make sure that the application components can be deployed to the entities.

C.30. Performance. The runtime implementation should offer fast enough device coordination and support for minimizing lag, for example by taking advantage of the heterogeneous networking and connectivity support (C.13, C.19).

C.31. Proactivity. Proactivity is one of the key principles of programmable interactions. Thus, the runtime implementation ensures that interactions can be triggered in a truly proactive way.

5.3.1 Coordination Model and Runtime

The Coordination model is "the glue that binds separate activities into an ensemble", according to Gelernter and Carriero [38]. In the context of AcOP, the coordination model refers to which actions are performed, as well as coordinating the entities and the operations they execute. Thus in AcOP, the coordination takes place on two levels: app and action. AcOP components can only be utilized by other AcOP components, and there is a hierarchy defining which components can utilize which. Moreover, while running the interactions, the components are coupled to each other, and the entities allocated to certain roles. At present, each device can only participate in one action at a time.

Application Boundaries

In the modern computing environment, where interconnected entities of different types co-exist, the boundaries become less strict, and to some extent lose their meaning. This diversity should be taken as an opportunity, as it allows sharing big tasks as well as allocating entities to perform some of the tasks [141]. On the other hand, some tasks should be offloaded from device to the Internet cloud [77]. The roles should be selected

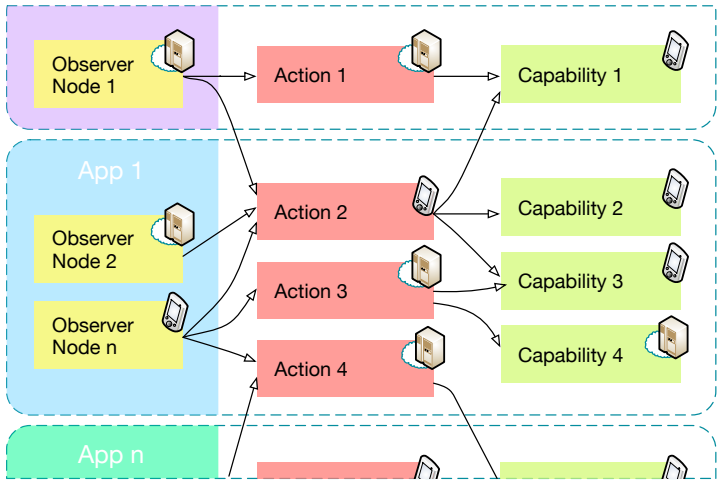


Figure 5.4: Runtime of AcOP: the application boundaries are artificial, any observer can schedule any action, and any action can call any capability.

based on the nature of the task, as well as based on the resources and states of the entities. Also, the location of the data should be considered while offloading the tasks.

The abstract and modular AcOP model enables the delegation of tasks, the runtime allocation of entities, and their assignment to roles. Each of the three concepts can, in theory, run on any entity of the computing environment (C.26), as depicted in Figure 5.4. For example, an action that requires fast coordination speeds could be deployed to a device that would then connect directly to the other participating devices, and coordinate them (C.30). On the other hand, actions that mostly employ entities that only have Internet connectivity should be coordinated from the Internet cloud. Moreover, the distribution of tasks can also support scalability (C.28), and even privacy. An app observing highly sensitive or vast amounts of data could run on the producing entity to avoid unnecessary transfer of data. Also, the capabilities do not always need to be executed on the device; for example, a capability controlling smart home electronics systems can be executed on the server-side, or on an entity that does not directly implement the functionality (e.g. on smart phone). Although the idea is that the developer could choose on which entity each component runs, in practice the type of the entity often limits what components can be deployed in them (C.29), and how seamlessly (C.27) the components can communicate

Coordination Model

Communication and coordination often go hand in hand. Connectivity types affect the network topologies, and hence limit how the entities can be coordinated. Moreover, the features of the supported communication protocols can limit the coordination. As an example, for a long time users of traditional desktop and mobile applications actually have been the coordinators of the operations as HTTP-based communication needs to be initiated by the client. The cyber dimension, however, has developed greatly, enabling new ways to support entity coordination. The entities may, for instance, be directly connected to each other forming *mesh topologies* [120] in which they then coordinate each other. Or, the topology may yet be the same *star topology* [120] as in Internet-based

cloud services, but due to the bi-directional communication protocols, the entities from both sides are able to coordinate each other. Today, entities support heterogeneous connectivity types, meaning that some of them can connect directly to other entities, and some can communicate via Internet. Thus the network is often based on *extended star topology* [120]. The topologies are depicted in Figure 5.5.

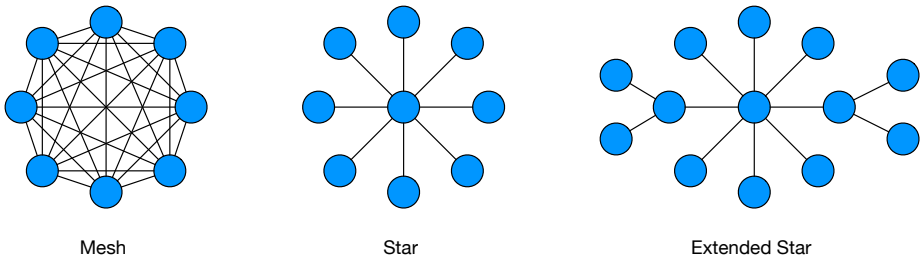


Figure 5.5: Entity topologies.

Communication turns into coordination when the purpose is to control the entities. The messages then contain an operation or data that the recipient is expected to perform or react to. The coordination can be implemented by writing data into a shared memory, or by sending data directly between the entities. Whereas the asynchronous and synchronous communication both may set their own challenges, the difference from the coordination point of view is not only in the communication itself, but also in the execution of the operations. If the sender waits until the recipient has performed the operation, the coordination is then *synchronous*; otherwise the coordination is *asynchronous*.

A common way of implementing asynchronous operations between entities (e.g. IoT objects) nowadays is based on relaying events. Whereas the entities can react to these events, and then emit events back, the coordination logic is still distributed, which makes it challenging for the developer to implement complex interactions. For example, often the developer must ensure of that the remote entity actually received the event, and then properly reacted to it. Moreover, the handlers reacting to the events need to be implemented and deployed in the different entities, and possibly updated if the data in the event changes. Thus, the event-based or asynchronous coordination in general, may not be the most approachable way of implementing interactions that require joint behavior of multiple entities. Asynchronous coordination, on the other hand, enables performing operations in parallel, and due to this efficiency fits well for many purposes, like relaying and processing contextual data in cyber-physical systems [129].

In synchronous coordination, on the other hand, the sender halts while the recipient has finished the operation. Although the control travels along with the operations, it returns to the sender. This makes it possible to wrap the complex messaging and communication into *developer-friendly method calls*. Moreover, if all the coordination logic is kept on one entity, it is intuitive for the developer to implement programs where *physical objects can be treated similarly as any other objects in object-oriented programming languages*. Whereas the deployability (C.29) is typically better compared to distributed coordination logic, the disadvantage is that this single coordinating entity may become a bottleneck and a single point of failure for the coordination. If the connectivity is slow, or there are many entities to coordinate, this may lead to slow coordination speed (C.30) or scalability issues (C.28).

Execution Model

As both coordination approaches have their advantages and disadvantages, they should be used for the purpose that they are best suited. For the same reason the underlying communication framework must support both types of messaging (C.15). The runtime implementation of the AcOP model utilizes both asynchronous and synchronous coordination. Figure 5.6 depicts the typical phases of executing an AcOP action.

Linda is a decoupled coordination language and model where processes write and read tuples into shared tuple space [4, 38]. Linda allows the processes to run on the same physical computer, or on separate entities. In AcOP a similar coordination approach has been taken for relaying entity state values asynchronously, as each entity can write its state into a shared *State Registry* component. Moreover, the registry compares the old and new values, and in case these values are different, it creates and emits events that are then relayed following a publish-subscribe design pattern. These events are then used for proactively triggering interactions (C.31). At present, and in the included publications, the State Registry component has been implemented as a cloud-based service, but each entity is, however, allowed to maintain their own registries as well.

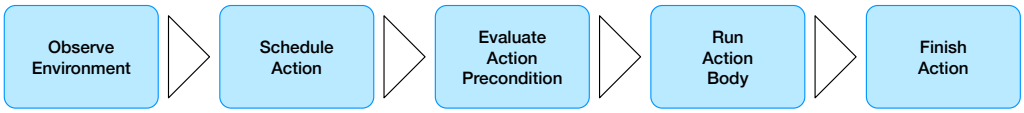


Figure 5.6: Typical phases of running an AcOP action.

Operations and device coordination in AcOP are based on scheduling actions for execution in a proactive fashion. The execution model of actions behaves like a single guarded loop in Dijkstra's guarded command language (GCL) [26]. In GCL, the basic building blocks are guarded commands which is a statement list prefixed with Boolean expressions. The statement list is eligible for execution only if the Boolean expression evaluates to true. Similarly, an action can be executed if the precondition is true, and the statement list corresponds to the body of an action. However, unlike in AcOP, the execution model in GCL is non-deterministic.

The coordination of the actual interactions takes place in a synchronized manner. While an action invokes a capability's method on some entity, the action process goes into a state where it waits for the response. The response can either be a response to the method call, or a contingency event, in case something went wrong on the remote entity end while it was executing the capability's method. Moreover, multiple capability methods can be invoked in parallel, and the execution continues only when all of the entities have returned a response event, or some of the entities returned a contingency event. In case of a contingency, the handler needs to recover from the issue and then return the value that was possibly expected from the original method call.

5.3.2 JavaScript as a Coordination Language

The key purpose of Action-Oriented Programming is to make co-operation and interactions between entities programmable with clear abstractions and a proper programming language. Today's computing environment consists of countless different types of hardware platforms which are programmed using various languages. To enable programming interactions between the entities in a unified way but yet allow the definition of complex

application logics, the abstractions of AcOP need to be implemented with a proper programming language. For this purpose, any modern programming language can be used. The selected language becomes the main programming language for implementing the actual interactions, and thus is called a *coordination language*.

The programming language needs to be selected carefully, as some elements of the coordination become tied to the limitations of this selection. As an example, deployability is one of the key elements as it describes the process of deploying the coordination logic to the entities responsible of coordinating others, and their ability to interpret the coordination logic (C.29). Deployability, on the other hand, has further influences to other elements, such as seamlessness (C.27), referring to how well the heterogeneous networks and devices with different connectivity can be utilized. During the years, we have studied the elements of coordination and experimented with different platforms and programming languages. The results of four different solutions have been reported in Publication [III].

Lessons Learned from Coordination of Devices

The original idea was to design our own programming language for defining AcOP actions. However, this would have required implementing a compiler or interpreter for each platform intended to be used for executing the coordination logic, which would have affected the deployability element (C.29). Moreover, this would have required the developers to learn a new programming language in a relatively challenging distributed and heterogeneous computing environment. For this reason, it was considered a better solution to use an existing programming language that developers are already familiar with. The designed programming language had already started to resemble Python programming language, and eventually became replaced by it. Python is a weakly-typed dynamic scripting language that offered the required flexibility for device coordination [113]. In order to use scripting language for coordination, it needs to offer features for interoperation with external components [113]. For this purpose, the meta-classes or meta-protocols of Python enabled extending the language with support for interoperation with remote entities. These were used for generating communication stubs, which are a common way of implementing procedures that handle the communication in the background, but hide this from developers and make the procedures appear local [22]. Examples of Python-based actions can be found from Publications [IV] and [V].

Running Python-based actions was first done on a cloud-based centralized server and the coordination required proper Internet connectivity. However, Internet cloud -based coordination has it flaws as has been discussed in this thesis. On the other hand, mobile phones today have excellent computing power and connectivity support, which makes it tempting to try utilizing them as coordinators for other entities. Due to this, an experimental personal area network (PAN) -based coordination entity-to-entity middleware was implemented with classic Bluetooth 4 (BT4) and Android platform. As Android and other mobile devices do not perform well in running Python code, Android's Java classes were used for defining the coordination logic. The actions were then deployed from a Internet cloud -based repository to the devices as Android application packages (*.apk*): The apk files were downloaded by the middleware framework on the device end, and then a class loader was used for dynamically loading the coordination logic. Naturally, the experiment was too Android-specific to be used with other platforms, and the deployability of the coordination logic was poor (C.29). However, the experiment successfully proved that PAN-based coordination dramatically improves the coordination

speed element (C.30), and helped to identify other challenges of using Bluetooth for device coordination.

Dynamic Coordination Approach

To improve the deployability element (C.29), a new approach was taken. At present, JavaScript seems to be the most platform-independent programming language. Its origins are in Netscape Web browser from Sun Microsystems, but for improving the interoperability with other browsers, the language was used for defining ECMAScript Language Specification [29]. Whereas today there are many dialects of ECMAScript, the language itself is typically referred to as JavaScript. For decades, it has been the programming language of the Web front-ends (Web browsers) [24], and recently it has also started becoming popular for implementing back-ends (Web application servers) due to Node.js platform (`https://nodejs.org/`). Moreover, iOS has had support for running JavaScript in a native app since version 7 with JavaScriptCore.Framework, and the work is ongoing to get similar native SDK support for Android as well. At present, some embedded hardware platforms have emerged that are also capable of interpreting JavaScript (e.g. Espruino (`http://www.espruino.com`), Tessel (`http://www.tessel.io`), and IOP (`http://iop.io`)). Consequently, JavaScript may finally fulfill Java's promise of write once, run everywhere [83]. Theoretically this kind of support would mean that the same code could be executed on both front and back-ends. Unfortunately, the code defined for a front-end in many cases is bound to different libraries than the code defined for back-end. Fortunately, however, defining the coordination logic typically does not require using additional libraries, but instead needs to access the hardware, and with proper abstractions JavaScript might perfectly fit for coordination purposes on all the entities.

Based on these properties of JavaScript, it was assumed that JavaScript would be the best solution and least platform-dependent programming language for coordination purposes. To test JavaScript's feasibility for this task in practice, a proof of concept implementation was first made with the Node.js platform. As JavaScript excelled in the initial task of coordination, a completely new AcOP middleware named Orchestrator.js (OJS) was implemented afterwards for conducting further studies. Today OJS is a complete middleware, meaning that it offers tools for defining all the abstractions of AcOP, and then running them inside the cloud. In OJS the AcOP concepts of app and action are defined with JavaScript, and as JavaScript is utilized as a coordination language, the capability stubs are also generated for JavaScript. Essentially, this means that the action-based coordination logic can now be executed on both Internet cloud and device ends. However, this requires that the coordinating device is capable of running JavaScript and supports direct entity-to-entity connectivity. An example AcOP action defined in JavaScript as a Node.js module is shown in Figure 5.7.

To experiment running the same JavaScript-defined AcOP actions on both server and device end, a new coordination framework was implemented for iOS devices. For deploying the body, precondition, and contingency methods defined in Node.js modules, these methods were exported since devices typically cannot utilize (require) Node.js modules directly*. Ensuring that each coordinator always had the latest definition of each method downloaded was done by notifying the coordinators with MD5 checksum [106] calculated from the methods' contents. This approach turned out to be a decent solution for

*Similar native support for modules may soon be available in client-side JavaScript as defined in the newly released ECMAScript 2015 standard [29].

```
module.exports = {

  body: function ( coffeeMachine, companionDevice ) {
    companionDevice.user.data[ 'askedAboutCoffee' ] = true;
    var m = 'Morning, ' + companionDevice.ownerName + '! ' +
            coffeeMachine.name +
            ' asks, if you want to have a nice cup of coffee?';
    companionDevice.talkingCapability.say( m );
    var response = companionDevice.dialogCapability.show( m, ['YES','NO'],60 );
    if( response != 'YES' )
      return;
    coffeeMachine.coffeeCapability.makeCoffee();
  },

  precondition: function ( coffeeMachine, companionDevice ) {
    return ( coffeeMachine.coffeeCapability &&
             companionDevice.talkingCapability &&
             companionDevice.dialogCapability &&
             coffeeMachine.data[ 'state' ] == 'LOADED' &&
             !companionDevice.user.data [ 'askedAboutCoffee' ] );
  },

  contingency: function ( information ) {

    // examine which kind of contingency did occur
    if( information.isError() && !coffeeMachine.isConnected() ) {
      // e.g. if a device lost it's connection, inform the user
      companionDevice.notifyCapability.show( information.getReason() );
      return;

    } else {
      this.kill();
    }
  }
}
```

Figure 5.7: An AcOP action which defines interactions between user (via companion device), and a coffee machine.

deploying the coordination logic, and improved the deployability element substantially compared to the first entity-to-entity experiment. As a result, the experiment proved that JavaScript appears to be the best solution for a platform-independent coordination language, since it allows defining complex coordination logics.

Benefits of JavaScript

Mikkonen and Taivalsaari have extensively studied [92, 126] JavaScript and other Web technologies, and claim that Web applications and the underlying technologies have many benefits over native applications. For instance, the Web enables easy delivering of the applications globally without installation, and the updates can instantly and automatically be delivered [70, 126]. Moreover, the basic Web technologies work on most platforms, and thus the applications are platform-independent [126]. The drawbacks of using Web technology have been the dependency of Internet connectivity, as well as the limited access to the hardware. However, with proper abstractions, hooks can be offered for accessing the hardware as described above. Furthermore, the actions are downloaded and stored locally and updated only when needed and Internet connectivity is available. Thus it is possible to gain the best from the both worlds.

JavaScript has also been studied in the context of *agents*, and *liquid software* [49], by Triglianos and Pautasso [131], as well as by Mikkonen and Taivalsaari [128]. The idea of deploying JavaScript-based actions on the entities where they make the most use are

reminiscent of the concept of liquid software applications. However, in liquid software the running applications are seamlessly transferred with their current states between the entities. In the current implementation of AcOP, on the other hand, the actions run on the same entity from beginning to end. In the future, however, it would be possible to study how the AcOP model could be applied in a liquid software context.

5.3.3 Support for Implementing Scheduling Policies

AcOP scheduling of interactions proactively is based on the concept of the app. An instance of an app essentially has two goals: 1) observe the computing environment, and 2) schedule the interactions for sets of entities (P.1 and P.3)

From Observations to Meaningful Abstractions

An important facet of scheduling is the way observations about the state changes are handled. Many of the current IoT and cyber-physical systems approach state management by publishing and consuming the data in a distributed manner [50]. As vast amounts of sensory observations are often rapidly produced [34], communicating and processing them can take a lot of resources [52, 53]. Moreover, the semantics of the sensory observation differ from one manufacturer to another, and abstractions are needed [50]. Commonly the issues of analyzing such data are known as "the Vs": volume, velocity, variety, and veracity [34].

The AcOP runtime implementation does not limit what data can be relayed and used for scheduling. However, using raw and unprocessed data quickly affects the performance of the system. Moreover, raw data itself is rarely useful, especially from a scheduling perspective [37]. A more efficient way of handling state changes and sensory observations is to perform preprocessing already at the device end, where the data is first produced, and then generate events based on changed values. This helps abstracting the observations, and further defining scheduling for the interactions in an understandable way (P.4). At present, AcOP runtime offers support for two types of higher abstraction level events (C.22).

Framework events are generated by the runtime implementation running on the device end. Examples of such events are the proximity data and graph. At present, OJS allows reporting the proximity data in various formats for the State Registry component. The devices can report classic Bluetooth MAC addresses with RSSI (received signal strength indicator) values, or Bluetooth LE service UUIDs with RSSIs or device identities with values with Beacon style proximity indicators. The changed values are then reported as proximity data events. These events, however, may be hard to utilize in the scheduling. For this reason, the State Registry component combines the reported values into a graph where the levels of proximity follow a convention similar to Beacons, and hence can be IMMEDIATE, NEAR, or FAR. Thus the preferable way is that the entity reports its whole, already processed, proximity graph as this reduces the processing on the State Registry. When the graph changes substantially, a new graph event gets published. The significance of the change can be described with a deviation value that indicates how much the proximity graph changed compared to the previous graph. Other similar higher level abstractions that could be reported in an automated way are, for example, battery level low (compared to battery level 13%), missed phone call, or unread SMS message.

Capability events are defined by the AcOP developers on the device end. In general, as the context where the sensor component is used sets the meaning for the observation,

generating the higher level abstractions in automated way is challenging. Whereas capabilities are used for abstracting the way entities are coordinated, they can also be used for producing abstract events. The capability then contains a public method that generates events by examining the state changes on the platform. This state method is then called by the framework at regular intervals, and the capability events published and mediated for other entities. Additionally, the entities may also publish events directly by calling the framework's method. As examples of capability events, consider the coffeeReady and coffeeMachineLoaded events that were presented in Publication [II].

In the future, generating the abstract events could be taken further by extending the device end frameworks. For instance, Henson *et al.* [52] have studied how abstractions could be generated for the purposes of Physical-Cyber-Social computing [116]. As an example use case, Henson *et al.* describe how a computing system could help medical doctors to diagnose illnesses based on symptoms. This approach is based on Parsimonious Covering Theory (PCT), which is an abductive logic framework taking some background knowledge (e.g. the symptoms of flu and cold) and a set of observations as input, and giving a set of possible explanations as an output. A similar approach could be utilized in Social Devices for generating events that hint at the user's willingness for certain types of social interactions. The observation input would then be certain state values of the user's companion device (e.g. social relationship between the nearby entities, and distance to these devices), and the background knowledge input would be the typical triggering contexts for certain interactions. Moreover, if users were able to teach their devices by giving feedback, the scheduling could become more accurate, and interactions more personalized over time (C.11). The abstract event could also help the user to better understand what information is gathered, and how this information is being used (P.4).

Implementing Scheduling Policies

Scheduling an interaction should be based on a policy or a strategy that in a controlled manner defines when the interaction should take place (P.4). For defining scheduling policies, the runtime implementation of AcOP allows implementing the apps in JavaScript, as depicted in Figure 5.8. Unlike actions, AcOP apps are not tied into any specific programming language or the same language as the actions. In fact, the apps can operate independently on different entities, and can be implemented with the native language of that entity. The benefit of an app implemented with a native language of a platform is that it can directly access data that is not necessarily produced and published by the framework or capability events. Naturally, however, these types of apps are then more tightly coupled to run only on the specific platforms. Thus, in most cases, it is recommended to use a language that is as platform-independent as possible to enable running the app on as many platforms as possible.

Like with actions, JavaScript opens the possibility to deploy and also run the apps on many types of entities. The benefit of running the app on the device end is that the observing of the platform's state can be done directly, without utilizing the network. This further helps taking scalability (C.28) and privacy into account while defining the scheduling policies, since possibly sensitive data does not have to be transferred or stored in any remote entity, but it can be processed and acted upon based on user's preferences directly on the location where it was first produced. Moreover, running all the components on the devices makes the system less dependent on the Internet connection, and therefore supports defining scheduling strategies also for interactions that take place in locations that lack proper Internet connectivity.

```
var httprequest = require( 'request' );
var tools       = require( 'tools.js' );
var pubsub      = tools.pubsub();
var Action      = tools.Action;

module.exports = {
  // set by the user while the app starts
  settings: [ companionDeviceId, location, distance, minTime, maxTime ],

  logic: function() {

    pubsub.on( 'location',
      // observer code omitted for brevity (20 loc)
    } );

    pubsub.on( 'coffeeReady',
      function( cxtDataVal, device ) {
        if ( device.user.data[ 'askedAboutCoffee' ]
          || settings.minTime < tools.now() || settings.maxTime > tools.now()
          || settings.distance > tools.distance( cxtDataVal, settings.location ) )
          return;

          var action = Action( 'InviteForCoffee',
                               settings.coffeeMachineId,
                               deviceIdentity );
          action.trigger();

    } );

  }
};
```

Figure 5.8: An example AcOP app defining a very simple scheduling policy for an action.

The challenge of this approach is that the app implementations often utilize libraries for accessing the different observable services, and the libraries are not necessarily supported or work in the same way in server-side and client-side JavaScript. In entity coordination with actions, this is a less common issue as the coordination logic is typically based on basic programming syntax. However, by abstracting the most commonly used libraries and methods (e.g. HTTP requests and Socket.IO pub/sub events), the deployability is possible to achieve to some extent. In an experimental implementation for iOS, the communication framework publishes the events also locally at the same time as it mediates them for other entities. Hence, the JavaScript based apps could run locally and have immediate access to the events.

The example observer in Figure 5.8 points out that implementing the scheduling policy becomes more straightforward when the abstraction level of the event is correct. Despite the abstract events, however, the policy can be more advanced and based on several events reported by different entities. For instance, multiple entities can mutually agree on triggering an interaction by mediating the same event (e.g. publicly or among friends) that describes their willingness to participate in an action (e.g. game).

Moreover, with proper connections to cloud and social media services, it becomes possible to share items from virtual world also locally when people meet in the physical world (P.1). For instance, by utilizing information from social media (status updates, likes, and hashtags), it could be possible to define policies for triggering social interactions. As the framework maintains social proximity graph and generates events about the changes, it can be observed when predefined layers overlap (people with certain social relationships are within certain proximity from each other) and then trigger an interaction based on their interests or shared information (P.2) and (P.3).

As digital content is an essential part of modern daily life, the Mobile Content as a Service approach offers a way to also bring the content as part of the programmable interactions (P.2). The VisualREST service for instance offers publish/subscribe events on new and changed content stored in *containers* within a mobile cloud, and this information can be used for scheduling the interactions that involve digital content. Furthermore, with the concept of *context*, VisualREST content can be shared within a certain environment (e.g. when certain people are present, or the location is correct), which further helps define the scheduling policies.

5.3.4 Interaction Capability Frameworks

Ideally, from the interaction perspective, operating systems and network infrastructures would have built-in, open-standard support for providing the resources of the computing environment as easily understandable and programmable objects that could be utilized for defining interactions with other entities. Yet, however, this type of support is missing from current platforms, and for this reason AcOP runtime requires frameworks on the device-end for sharing resources and taking care of communication. The framework can be implemented with various technologies, but the goal is always the same: Offer the resources of the different physical objects in such a way that they can be considered as programmable JavaScript objects in AcOP actions.

Cross-platform SDKs

Cross-platform SDKs have as a goal to develop once, run everywhere. The first implementation of the capability framework was Python-based, enabling capability development with the Qt cross-platform framework and toolkit (`http://www.qt.io`), its PySide Python binding (`https://wiki.qt.io/PySide`) to be exact. The advantage was that most capabilities did run on the Maemo/MeeGo mobile platforms, as well as on Linux, OS X, and Windows platforms. This portability was mainly due to the fact the MeeGo platform was Linux-based and Nokia was strongly pushing this compatibility forward. Nevertheless, the approach worked well on devices at the time. However, later not many other mobile device platforms have been able to run Python applications. Thus other options were studied in the later research.

Today, cross-platform SDKs may help while developing capabilities for mobile platforms. They can be especially useful while implementing graphical user interfaces for multiple platforms simultaneously. For instance, Qt has been developed further and now offers better portability between the main mobile platforms. However, some issues may still occur while trying to access some specific resource of some specific platform, especially if that resource is new or works differently on the platform than in other platforms. Although cross-platform SDKs like Qt offer support for many embedded environments, the support for many commonly used hardware platforms, like Arduino and .NET Gadgeteer, is still missing. As the idea of the capabilities themselves is to take advantage of the different resources each device type has to offer, this aspect must be taken into account.

Native SDKs

Native SDKs refer to the original tools offered by the vendor for implementing application software for the platform. These tools typically offer the best and most up-to-date support for accessing the hardware and other resources of the system. After the Qt/Python-based multi-platform framework, the focus of supporting capability development has mainly

been on the native SDKs of each platform. The frameworks have been implemented for Android, iOS, and Windows Phone 7 mobile platforms, as well as for more constrained .NET Gadgeteer and Arduino hardware platforms.

Naturally, the disadvantage of these tools is that the capabilities must typically be implemented separately for each platform. However, the SDKs of today have excellent libraries, and thus implementing a capability does not typically take too much effort, especially when the capability has already been designed and implemented for some other platform. Moreover, the developer typically has direct access to the resources without a third party, and plenty of examples in the Internet that help in the development process.

One of the main issues of capability development has always been the deployment element: How to deploy the capability implementation to a device? (C.29) With the Qt/Python-based framework this was possible by downloading and importing a capability module during the runtime. A similar approach was experimented with in Android framework, but issues did occur as reported in Publication [III]. Moreover, the business decisions of some platform vendors also prevent importing new code during runtime. Naturally, this would open the platform to many security threats, and hence the decision is easy to understand from a security point of view.

At present all the capability implementations are packed inside of one mobile phone app, and new capabilities can only be added by updating the app. However, not all the entities need the entire collection of capability implementations, as they unnecessarily consume space even when not in use. Another solution would be to implement each capability as a separate app, or create bundles of capabilities by packaging certain capabilities into a couple of mobile applications. However, this approach would face technical issues related to the current mobile platform implementations as they are not targeted to run multiple applications in parallel. In the future it might be possible that these types of limitations will no longer be valid, and even safe mechanisms could be offered by vendors. In some sense, the concept of a capability could even replace the concept of a mobile application in the long run, when computing becomes more ubiquitous.

Web technologies

Web technologies can offer truly open standards that help implementing portable applications that work out of the box on multiple platforms [126]. To gain the aforementioned benefits of Web technologies, an experimental Web-based framework was implemented. Not only would this approach have enabled implementing some capabilities at once for multiple platforms, it also would have enabled deploying and updating the capabilities automatically. Unfortunately, it turned out that the Web technologies were too limited in accessing hardware as required by many capabilities. Moreover, despite the open nature of Web technologies, other types of limitations also emerged. Take *Talking*-capability as an example, which is one of the most characteristic examples of Social Devices (P.1): In the Web-based framework, this capability was implemented in the same way as for the Qt-based framework where a speech service was used for translating text to speech, and then an audio player for playing the produced sound file. With the Web-based framework the capability simply did not work as expected on any iOS devices as the iOS platform stubbornly prevents auto-playing in any HTML 5 audio element, and always requires user-generated input. As of yet, no workaround has been found to circumvent this issue on any browser on iOS. On Android browsers however, auto-play worked as expected.

Hybrid application SDKs and approaches

Hybrid application SDKs have emerged due to the issues with the plain Web technologies. These approaches try to gain the best of the both worlds by combining native and open Web technologies, with the goal of requiring as little effort as possible to implement the same application for multiple platforms. Some examples of such technologies are Apache Cordova (`http://cordova.apache.org`), NativeScript (`https://www.nativescript.org`), and React Native (`https://facebook.github.io/react-native/`). Typically, each of these approaches require some effort for customizing the code separately for each platform, as they all have some of their own principles (e.g. usability guidelines). Moreover, similarly to all the other than native SDKs, these tools are typically only supported by the main mobile device platforms.

Another type of hybrid approach has also been taken by utilizing Node.js server-side JavaScript runtime for a capability framework. As Node.js can operate on Mac OS X, Windows, and Linux platforms, including Raspberry Pi, the capabilities for this framework can be implemented with either server-side or client-side JavaScript. The advantage of this approach is that the server-side JavaScript together with third-party libraries can offer deeper access to the platform resources as well as controlling IoT objects. Moreover, as the underlying platform remains the same on both sides, also exactly the same apps and actions can directly run on the device-end framework as on the server without the issues mentioned above. Naturally, utilizing client-side JavaScript for implementing the capabilities for this framework suffers from the same issues as mentioned above.

Web technologies, on the other hand, evolve fast and the support for accessing hardware improves continuously (`http://caniuse.com`). Already now, Web APIs on many platforms offer similar access to the resources as the native SDKs. As the responsive layout frameworks make it trivial to implement user interfaces that scale on any screen sizes, and the efficient communication protocols offer near real time communication, it makes it worthwhile to consider them as an option for implementing some capabilities. In the near future these technologies may indeed become the preferred ways of implementing the capabilities for many of the platforms. Furthermore, Web technologies could then help to overcome the issues with deployability that have always been one of the main problems in AcOP capability development (C.29). Ideally, a user could make any device with a Web browser a social device simply by logging in with username, device name, and password. The browser would then fetch a virtual social device configuration, with capabilities already defined.

Nevertheless, all the above are valid approaches, and can be used for implementing the capabilities. However, their limitations must already be kept in mind while considering the purpose and the appearance of the entity before actually designing and implementing. Consequently, as the discussion above reveals, the capability development also craves open technologies and standards, similar to action and apps.

5.4 Evolution of Action-Oriented Programming

Enabling programmable interactions is a continuous process. Action-Oriented Programming has evolved during the year when it has been used in its real context, by developers with different level technical skill to implement software for various purposes. In most of these uses and implementation processes the author of this thesis has had some kind of role of an advisor, while he has at the same time studied the feasibility of AcOP for

enabling the programmable interactions. AcOP has also been improved within these processes by fixing bugs when those have emerged and by implementing new features when those have been requested by the developers. Research method where also the researcher is somehow involved is typically called as action research. The following summarizes how AcOP has evolved while conducting action research.

Social Applications for Social Devices

The suitability of Action-Oriented Programming and Orchestrator.js middleware for implementing social applications has been studied and evaluated by Benítez in his MSc thesis. [12]. The evaluation is very detailed, going through all the phases from installing Orchestrator.js middleware and using the tools, to introducing some ready applications and their usage. In his thesis, Benítez concludes that Action-Oriented Programming offers good abstractions for programming interactions between the devices to support developing social applications (C.20). According to Benítez, also the Web-based IDE and tools become handy, since the developer don't necessarily have to install any tools on the computer to start developing with Action-Oriented Programming.

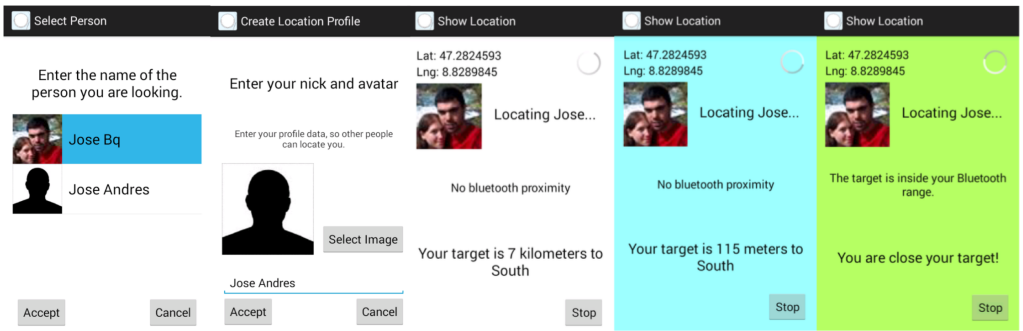


Figure 5.9: Screenshots of HaveYouMet social application implemented with Orchestrator.js middleware [12].

One drawback that Benítez found was the lack of proper documentation about how to use Orchestrator.js middleware. For this reason, he wrote a detailed tutorial that introduces how social applications can be build using Orchestrator.js middleware and Action-Oriented Programming. The tutorial, which is now online (`http://orchestratorjs.org/tutorial.pdf`), contains five example implementations. One of them is *HaveYouMet* (see screenshots in Figure 5.9) where the idea is that the devices can help new people to find each other. Another example is *FollowMe*, where the devices first interact by exchanging some pieces of information that their owners have specified about their personal interests. With this information, the devices then try to find matches and help their owners to connect and interact within the cyber world.

The suitability of Action-Oriented Programming for implementing similar types of social applications where the cyber world and physical world processes become bridged (C.5) was estimated also in another experiment. In [1] we introduce a *PhotoSharing* application which was implemented with AcOP by a MSc student in couple of days. The idea of PhotoSharing is to show how content shared in the virtual world can be shared in the physical world, and its social situations, when we actually meet our friends and family: The devices then proactively initiate and suggest a photo sharing session for their owners

when new photo album has been shared in social media, and the friends have gathered together in a cafeteria, for instance. PhotoSharing was presented in Figure 3.6 and a demonstration video can be found online (`https://youtu.be/DigLU_UjYSI`). This experiment help studying how the content from social media should be handled in the programmable interactions. Moreover, the experiment showed that AcOP capabilities implemented with Qt can directly work with in PC as well (C.29).

Another issue that came out in Benítez implementations was becoming aware of other users and devices (C.1). Benítez used classic Bluetooth classic technology for detecting other devices (and their owners) in the proximity and found that this was consuming a lot of battery (C.4). This observation is align with issues presented in Section 3.2 as well as in Publication [III]. Benítez's solution for the problem was to use GPS for detecting when the users are in the vicinity, and then start using classic Bluetooth discovery for detecting the actual distance, as illustrated in Figure 5.9. However, also Benítez believes that in near future Bluetooth Low Energy technology has potential to void these issues.

Social Games

In addition to the experiment reported in Publication [V], where a group of students implemented a zombie game named *Apocalympics* (see Figure 5.11), also others have utilized the Action-Oriented Programming for developing social games. *CarGame*, represented already earlier in the thesis (see Figure 3.9), was implemented by a MSc student. In this turn-based game the idea is to get more speed for your car by cheering as loud as you can. Again, in this programmable interaction the Social Devices are proactively initiating and suggesting the game for friends who are meeting in a bar, for instance (P.3). See demonstration video from (`https://youtu.be/T3sL3JYjCEM`).

A more profound analysis on the Action-Oriented Programming for developing social games has been conducted by Aguilar in his MSc thesis project [3]. Aguilar studied Orchestrator.js middleware and Action-Oriented Programming in details to find out how the provided concepts and components can be used for developing games where multiple devices are interacting (C.20). The findings of Aguilar were that that many such games require implementing the same features over and over again, and that support for these features were yet missing. Aguilar also helped solving these issues by implementing *Game Composer Framework*, which out-of-the-box offers features like profile management (username and avatar), spectator mode, player disconnection handling (C.24), and rematch management. The framework was implemented as AcOP capability, and it can easily be placed to the Android project that is used for developing the actual AcOP capability for the game. Hence, the framework can be used in future while implementing new games. Moreover, during the experiment a need for new feature came up, and it was implemented by the author of this thesis. This feature now allows new devices to join into a running AcOP action (C.23) and (C.19).

To demonstrate his framework in practice, Aguilar implemented two traditional games *Texas Hold'Em* and *Parcheesi*, which have been presented in Figure 5.10. Also demonstration videos are available from online (`https://youtu.be/vOlawVvZT1k`) and (`https://youtu.be/LIyB6zfel7U`). Additionally, the thesis offers documentation about the features of Game Composer Framework, and a guide for installing and using it in practice. Game Composer Framework and documentation are available from GitHub (`https://github.com/dpares/Game-Composer-Framework`).

Figure 5.10: Texas Hold'Em and Parcheesi implemented with Game Composer Framework, Action-Oriented Programming, and Orchestrator.js.

Characteristics of Social Devices

One of the fundamental principles of programmable interactions is to be social (P.1). Social Devices is the most visible instance of these social and human-like interactions, and for this reason the ability to communicate with voice has from the beginning been studied. The first Social Devices demonstration implemented was *Talking Devices* where mobile phones translated text to speech while they were communicating in the background via cloud, giving a human-like impression for the co-located people. Later on the *Talking*-capability (C.21) developed for Talking Devices was used by developers for different purposes. For instance, in *MedicineReminder* voice modality is used for reminding to take the medicine when the user happens to be in the close proximity of the medicine jar and it is about the right time to take the medicine. Moreover, if taking the medicine is forgotten, the device keeps track of this, and may even tell family members about this if this type of behavior is allowed by the user. *BusReminder*, on the other hand, is targeted to office or home environment for observing busses in real-time, and then notifying the user when it is time to go to bus stop. Other similar examples are *CalendarReminder* and *SMSReminder*, which can both utilize user's other devices and even other users' devices to notify about urgent events and emails. These Social Devices demonstrations implemented by various people proved that the Action-Oriented Programming can be used for programming meaningful interactions between different types of devices (C.21).

Currently, however, this type of proactive (P.3) and social (P.1) behavior and audio-based communication with the machines may feel unnatural for the humans. Jarusriboonchai *et al.* have studied this by conducting a series of *Wizard-of-Oz user studies* with Social Devices Platform in [67]. In their study, they were simulating the social behavior of devices in a semi-public coffee room area with multiple participants. Later on the participants were interviewed and the results pointed that many people were suspicious about the speech-based interactions with the devices in a social context. Since this was the first time most of the participants experienced this type of social behavior coming from the devices, it is only natural that people were suspicious and did not know how to react. Audio-based interactions, however, are quickly becoming common, as the emerging digital assistants from many companies are indicating. Moreover, with social robots, which aim to behave and appear like humans, the voice is likely also the most natural modality

for a human to communicate with them, since this is the way people are accustomed to interact with other humans. Hence the actual question for audio-based Social Devices is: what the device should be like that communicating with it with voice feels natural? Nevertheless, the experiment from Jarusriboonchai *et al.* [67] proved that the platform and Action-Oriented Programming can also be used for conducting Wizard-of-Oz user studies.

Programmable Interactions for Constrained and Custom Social Devices

Many types interactions can easily be programmed with AcOP using either mobile devices or traditional desktop computers as has been discussed above. Orchestrator.js, however, enables that all types of devices and their resources to be utilized in the programmable interactions (C.20). Some physical objects can even be turned into Social Devices by simply tagging them with Bluetooth beacons which allows them to broadcast their presence. The actual functionality then locates in AcOP's app and action components, as well as in the capabilities of other co-located devices that can be utilized for user input and output. A good example of this is the aforementioned MedicineReminder, where most of the functionality is implemented in the app component, and a mobile device is used for giving user the audio and dialog based input and output, and where the actual medicine jar is simply a Bluetooth beacon.

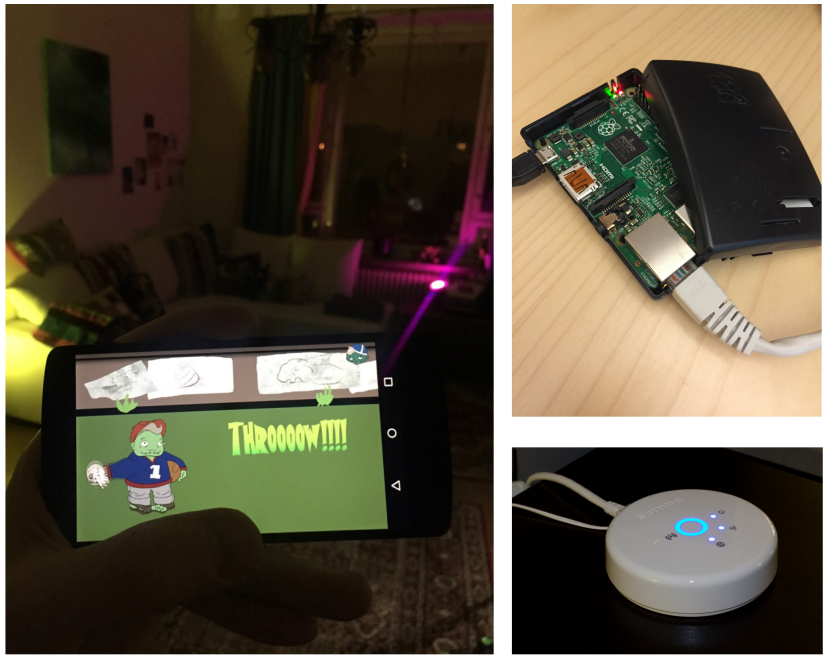


Figure 5.11: Phillips HUE lights used for changing the atmosphere of the physical world in an environment where Apocalympics zombie has been triggered and Lights-capability is available.

Another example of programmable interactions with home electronics is presented in Figure 5.11. In this setting a relatively simple Phillips HUE lightning system is enabled for programmable interaction via its communication bridge that operates in the same home network with a Raspberry Pi running Orchestrator.js middleware (C.16) and (C.26). The

AcOP *Lights*-capability then runs on the Raspberry Pi, which simply acts as proxy relaying these commands (C.27). The capability makes it simple to program the interactions between different types of devices, since the light bulbs can then then simply be seen as programmable JavaScript objects by the developer (C.20). Figure 5.11 illustrates how the lightning in the home has become part of the Apocalympics game, making the atmosphere in the physical surroundings more zombie-like when the game has proactively been triggered (C.22).

The goal from the beginning has been that all types of devices could be enabled for programmable interactions (C.26), and that the developers should be able to easily built their own custom devices and new interaction capabilities (C.21). To study the possibility of using constrained hardware platforms with Action-Oriented Programming, we hired a MSc student. In his Master's thesis project, Kelloniemi implemented a capability framework for Arduino hardware platform which, compared to mobile platforms, for instance, gives much better options to affect and change state of the physical world. The implemented capability framework allows building new types of Social Devices, which capabilities are then programmed using C++ (C.21).

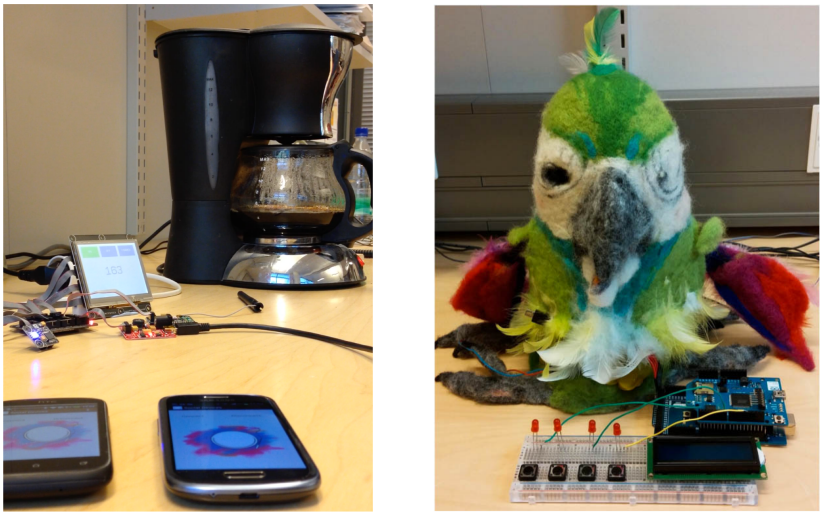


Figure 5.12: .NET Gadgeteer and Arduino hardware platforms enabled for programmable interactions.

To test his framework with Action-Oriented Programming and its runtime, Kelloniemi implemented few example Social Devices, like *Parrot* that has been illustrated on the right hand side in Figure 5.12. The initial idea of the Parrot was to be a simple social robot, telling rude jokes about nearby users. The torso and wings of the Parrot are controlled with couple of servo motors attached to Arduino. The Parrot also has a camera eye, and speakers for audio output.

The first version of Kellonimi's framework utilized HTTP/Comet communication with the older SDP. From this experiment, we learned that Arduino had difficulties in maintaining a stable connection with Comet. For this reason, Kelloniemi implemented also another communication protocol with Socket.IO which was working without any issues. Hence, the results were in align with the results of Publication [III]. Despite of the improvement, the lag in communication may yet prevent implementing some highly communication

critical applications, but these issues are mostly due to the limited computing resources of Arduino. However, new communication protocols can yet be studied in future, and one potential option would be the widely used and lightweight MQTT. However, since MQTT and Socket.IO remind each other from their principles, it should be relatively simple replace the communication with MQTT in future (C.14).

Nevertheless, Arduino framework now allows utilizing one of the world's most popular prototyping system in the programmable interactions, and for implementing new types of Social Devices with only the imagination being the limit of programming interactions with the physical world (C.22). Moreover, as reported in Publication [II] .NET Gadgeteer capability framework also enables building similar custom devices even easier, as shown by this tutorial video (`https://vimeo.com/89557849`). The framework has been used for building *Social Coffee Machine* presented in the left hand side in Figure 5.12.

5.5 Related Approaches

Recently, many Internet of Things frameworks have been introduced, and some major Internet companies are also launching their own solutions. Typically, however, these platforms are not especially targeted for implementing social and proactive interactions between multiple co-located humans and computers. Hence these approaches do not well respond to challenges from (C.1) to (C.11). The following introduces some of the most well-known approaches, and compares them to the presented Action-Oriented Programming model and its runtime environment implementation.

End-User Approaches

The company *IFTTT* (`http://ifttt.com/`) offers an end-user platform for defining rule-based control flows. The name comes from words *if this then that*. A control flow is defined with the concept of *receipt*, which is an easy and understandable way to define what happens when certain rules are met. The receipts consist of two parts, the *if* part and the *then* part, which makes them clear though quite limited. Some examples of receipts could be: if I receive an email with a specific tag, then print it, or, if I mark a vacation on my calendar, then schedule my smart thermostat accordingly. See Figure 5.13 for examples of receipts.

IFTTT was initially released in September 2011, but later on, in 2015, the app itself was renamed IF. At the same time, the company also introduced the Do Button app, which allows users to manually trigger the then parts of the receipts. An iOS version of the app was released in July 2013, and an Android version in April 2014. Although IFTTT mainly offers a means for automating daily routines by improving the interoperability of end-user cloud services, the approach has some similarities to the Action-Oriented Programming.

The runtime model of IFTTT has similarities to the runtime model in AcOP. They both allow running components inside the cloud environment and trigger tasks according to changes in context (C.31). However, AcOP allows defining more complicated flows (C.20) and running the actual entity-controlling logic as well as other components, both inside the cloud as well as on device side (C.28). This can be done as the "flows" in AcOP have been split in to two concepts, app and action.

The then part of the receipt closely relates to the concept of action in AcOP, but is much simpler (C.22). Where the then parts of receipts are mainly used for turning things on or off, or triggering rather trivial tasks such as sending an email, actions can be

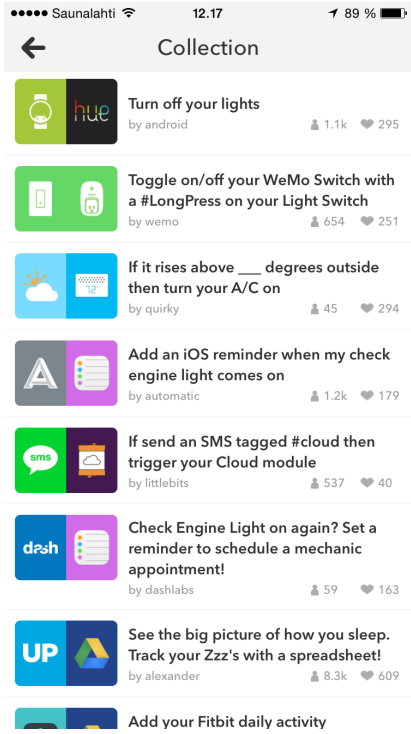


Figure 5.13: Screenshot from IF app's collection of receipts.

more long-lasting, containing several such trivial tasks that together solve a complicated problem. An example of a long-lasting action could be a game logic which coordinates several entities during the game. Actions can also be as simple as the then parts of IFTTT and used similarly. In AcOP, the idea is also that the actions can be deployed in an entity that is selected as a coordinator, dynamically, before an action begins (C.29).

The if part of a receipt resembles a single observer component of an AcOP app; these both are used for defining the rules that need to be met in order to trigger the next step (C.22). However, the if part is rather simple, and only allows defining very simple context that does not take other people into account (C.23). Thus, the collaboration is essentially limited to a specific user only (C.1). In IFTTT approach, the user may not be aware of the ongoing interactions (C.8), and also has no ways of affecting to triggered then parts, unlike in AcOP (C.23). Consequently, the AcOP app may contain several observers for detecting more complicated context as the app is a JavaScript application. Moreover, at present the IFTTT does not seem to offer any deeper way of defining the behavior of the entities, nor for considering their differences (C.21). As an example, Phillips Hue (`http://www.meethue.com/`) lights can be turned on or off, but adjusting the colors more precisely currently seems not possible. This lack of considering the differences of the entities seems odd as the aim of the concept is to automate things. To summarize, in spite of the limited nature of IFTTT it offers a large set of receipts that can be used to automate simple tasks, and hence make daily life easier.

Mainstream Approaches for Developers

Node-RED (`http://nodered.org`) is a lightweight tool for building service compositions. It is an open-source project developed by IBM Emerging Technology unit, which the authors describe as "a visual tool for wiring the Internet of Things". The main idea is to make multiple lightweight processing nodes to communicate with each other and with online services (C.20). For this purpose, Node-RED offers a graphical tool which is used for defining how the nodes are wired together. Figure 5.14 is an example flow that fetches the United Kingdom's current power consumption from the Web, then runs a parsing function given as a parameter inside a message, and finally prints the results to output debug nodes.

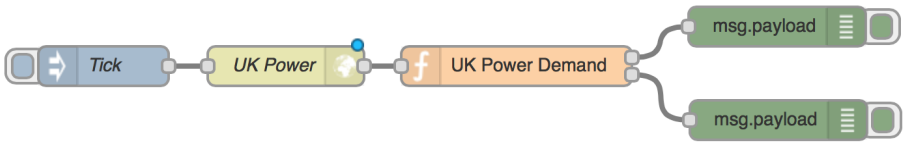


Figure 5.14: Node-RED example flow that gets current power demand of United Kingdom.

The graphical *control flows* are translated into JSON messages that are sent between the nodes (C.15). Hence, the idea resembles BPEL (Business Process Execution Language) [140], although BPEL is typically implemented with SOAP-based communication. In contrast to BPEL, however, Node-RED's messages are defined in JSON which makes the communication more lightweight. Moreover, unlike in SOAP, the messages in Node-RED may also contain a property named *func* which contains the functionality that the node is expected to perform.

Node-RED flows to some extent resemble actions in AcOP, and the messages defined inside the flows recall the messages that travel via the communication framework. However, the coordination and runtime models essentially differ from AcOP. As an example, the AcOP action can be run on entities capable of executing JavaScript, but the coordination logic still remains centralized. Thus a single entity coordinates others during the whole action execution. In contrast, when a Node-RED node receives a message, it executes a method defined inside the message, and then sends a message to the next node. As the control travels with the flow and the nodes can send messages to other nodes, the model cannot be described as a start model, but rather as a mesh model.

The openness of the Node-RED platform allows defining much more complicated control flows than in the IFTTT platform. Furthermore, IFTTT runs the flows inside their own cloud, whereas Node-RED distributes the flows and deploys them to the nodes, hosted by IoT hardware as well as by more traditional PC hardware (C.25). Currently, officially supported hardware seems to be limited to Arduino, Raspberry Pi, BeagleBone Black, and PC/Unix platforms.

Node-RED supports multiple IP-based networks and communication protocols such as WebSockets, HTTP, and MQTT (C.14). Despite the wide support for different communication protocols, Node-RED does not perform well in utilizing heterogeneous networks (C.13). Nor does Node-RED seem to offer support for dynamically changing environments as communication is hardcoded into the flows (C.19). However, in the future the support for other connectivity types will most likely improve as Node-RED

has been implemented by a large company with a lot of resources that can be committed to IoT [121].

Despite the promise that Node-RED holds in, wiring the IoT, its approach still has some deficiencies. It remains uncertain whether a tool for building control flows can be utilized for building complex interactions (C.20). Also for AcOP, a graphical tool for defining actions was originally considered, as mentioned in the initial release of Social Devices in Publications [II]. However, later the idea was abandoned as it can be hard to implement more complicated coordination logics with this type of graphical tool. For instance, consider how a developer would be able to define a game logic with such a tool. It follows that its limitations are similar to those of BPEL's [32].

Another deficiency of Node-RED flows is that they do not seem to offer a clear way to react to context changes during the execution of a flow. As the flows typically seem to be short-lived this may not be an issue. On the contrary, managing contingencies may be problematic if the system does not allow reacting to the error cases, for example by giving users information (C.24). After all, managing contingencies and being context-aware are two of the main requirements for pervasive service compositions [17].

Finally, Node-RED does not seem to consider the social dimension properly challenges (C.1) – (C.11). Currently, the interaction with people and with the social dimension in general, is supported only by sending messages with basic tools like email, twitter, or IRC. The users are not really empowered for interacting with the things, and hence may remain outsiders. Due to the deficiencies mentioned here, at present Node-RED may be well-suited for automating simple things, but not for programming social interactions between people and the things.

Octoblu (`http://octoblu.com`) is a company that offers its customers a system for creating IoT services. Their whole system consists of several platforms like Meshblu, Gateblu, and Microblu that enable different tasks. The core of the system seems to be the Node-RED-based Meshblu platform (`https://developer.octoblu.com`), which together with their graphical tool, enables creating flows similar to Node-RED. With their other platforms, however, the support for hardware platforms is much more extensive than in Node-RED (C.26). Octoblu seems an interesting platform that is growing fast, but so far many of the same deficiencies of the original Node-RED still apply – it can be challenging to define the programmable interactions for a group of people, especially complex application logic (C.20).

EVRYTHNG (`https://evrythng.com/`) is a promising IoT platform initially launched several years ago. It offers an Internet cloud-based platform that enables reading, writing, and analyzing data generated by the different types of objects, that in the system are called Thngs. For connecting the various hardware platforms to become Thngs, libraries are offered for software developers (C.26). Every connected object is then given an identity (C.12), and can be utilized in applications via RESTful API. The programming model of EVRYTHNG differs from the AcOP model, although there are some similarities. For instance, EVRYTHNG contains a concept named application, which corresponds to a native or a Web app, allowing them to communicate with each other – The idea is to control the IoT from native apps, unlike in programmable interactions. An AcOP app is fundamentally different from a native or Web app, and hence also different from EVRYTHNG application – it can run on different entities (C.25), and its purpose is to observe the environment (C.22) and schedule actions in a proactive way (P.3). Moreover, EVRYTHNG also has a concept named action, but its meaning is different from AcOP actions, since a single action in EVRYTHNG is essentially considered an operation

performed with a Thng from native app (C.20). Hence, EVRYTHNG actions are closer to capabilities than actions in AcOP (C.21). In AcOP runtime, however, the physical objects as well as their resources are regarded as JavaScript objects. AcOP actions, on the other hand, contain several operations with the entities, and the action also defines the precondition for these operations to take place.

Bluemix is IBM's generic cloud platform with access to many of their services, runtimes, and frameworks. Bluemix offers similar cloud services as Amazon Web Services (AWS), and Microsoft's Azure. These services cover support for virtualization and different platform-as-a-service solutions that can be managed and bridged together with Bluemix (`http://bluemix.net/`). This will support scalability (C.28), although due to business reasons, the integration of the services works the best within their own ecosystem (C.16). One specialty is Watson, IBM's artificial intelligence service originally developed to compete against humans in the TV show Jeopardy! (`http://www.ibm.com/smarterplanet/us/en/ibmwatson/what-is-watson.html`). Now Watson can read and understand natural language, learn dynamically by tracking its users, and generate hypotheses and evaluate them. These advanced features help to make the platform more social than the other related approaches (P.1). Bluemix offers developers APIs for accessing Watson (C.16). IBM claims that Watson can already diagnose cancer more accurately than doctors. It will be interesting to see how Watson will develop when it gets access to the data coming from our health and wellness sensors.

The actual IoT service is called the IBM Internet of Things Foundation (`http://www.internetofthings.ibmcloud.com`) and was released in 2014. This service is the most heavyweight IoT approach from these related approaches. As a typical industrial approach, the thinking is very business-oriented. IBM sees the Internet-connected devices as network-connected pumps that offer a "huge river of data" as reported by IEEE Spectrum [121] (C.26). The money comes from helping companies analyze data coming from the millions of sensors and devices, such as smart watches and other wellness devices (co-operation with Apple), and from industrial equipment (M2M) (C.16). If IBM succeeds in earning a profit around this, they most likely will continue investing in successful communication channels as they have already done by participating in multiple open-source projects, developing MQTT libraries for instance. However, as IBM's business currently seems to focus on analyzing the data, their interest is mainly on getting the real-time data out from the IoT rather than on the coordination and cooperation of the devices as such (C.20) – (C.25).

Recent Comers

Parse for Internet of Things platform is one of the most recent IoT middlewares, released by Facebook (`https://www.parse.com`) in 2015. Parse offers SDK's for multiple platforms, which developers can use for building multi-device applications. The coordination approach seems to have some similarities to the initial implementation of Web of Things Platform [45]. The communication is based on the REST API which allows devices like Arduino and Raspberry Pi to report information to Facebook's servers, enabling these entities to communicate with each other (C.15). The runtime environment of Parse allows running parts of the software in the cloud, as does the runtime of AcOP (C.25).

The most interesting point with the Parse platform is that it comes from Facebook, which is not a hardware manufacturer, and hence does not have a hardware ecosystem as such. This supports implementing more vendor-neutral applications (C.26). As Facebook is a big company, and has a huge user base, the platform has plenty of potential to succeed,

and also to make the interactions social (P.1) and personalized (P.2). However, the downside is that many users are already not too happy with Facebook's privacy rules and permissions for users' content. When IoT devices can collect highly intimate information automatically, users will have even less control over what information Facebook has access to, compared to the current model where users make conscious choices while making status updates (C.9). At present, using the service seems to be free. This raises a question over what kind of revenue generation model Facebook will take with the platform if it becomes popular. In the worst case, if it turns out that Facebook uses private information against users' preferences, their suspicions about privacy in IoT would be reinforced. This would not support the principle of being predictable (P.4).

Brillo and *Weave* are the most recent IoT approaches by Google, initially released in 2015, but at present is not yet publicly available. Weave (`https://developers.google.com/weave/`) is an open protocol that promises to support IoT devices, as well as mobile platforms, like Android and iOS. Brillo (`https://developers.google.com/brillo/`), on the other hand, is an embedded operating system that can communicate with other devices using the Weave protocol. Together they seem to be some kind of equivalent to Apple's *HomeKit* approach, although they target the whole IoT sector, not just home automation. At the moment there is only a little information available about Brillo and Weave, but as they come from Google, it can be assumed that they will quickly become popular among developers.

Physical Web (`https://google.github.io/physical-web/`) is another interesting research project form Google. It uses Bluetooth LE beacons for wirelessly broadcasting the URIs of the physical entities nearby, and the URIs can then be opened in a web browser for interacting with objects or services linked to these beacons. The idea itself, however, is essentially the same as with QR codes: detecting the entities yet requiring picking up a mobile device, and then manually selecting the object or service [16]. Thus the approach simply automates some steps for the user to initiate the interaction, but is not truly proactive (P.3). Moreover, the approach is not social as it enables the user to interact with an entity remotely via a web browser user interface, but not with multiple objects or people at the same time (P.1).

Summary

To summarize, many IoT platforms exist that can be used for building *cyber – physical* services, referring back to Chapter 2 and Figure 2.7. These solutions mainly are Internet cloud-based, meaning that at least some of their components run on a remote server. All the presented solutions have their own concepts for defining the services, communication, and interactions. These concepts, however, are rather technical, and not targeted for defining co-located and social interactions that take place in the physical world – Humans and especially social aspects typically seem to be forgotten almost completely in all of the discussed approaches. Although, in the end it is the developer who sets the tone for interactions, and can make the interactions social (P.1), personalized (P.2), proactive (P.3), and predictable (P.4), still, the tools used for programming and executing the interactions is also important and should help in this attempt.

Chapter 6

Overview of the Included Publications

This chapter gives an overview of the publications included in the thesis. None of the publications have previously been used as a part of any other doctoral thesis. Also, more details are given from the candidate's contribution for implementing some of the systems. The publications were written between 2011 and 2015.

Publication [I] introduces a manifesto named Internet of People. This IoP manifesto is one of the most important contributions of the thesis, as it defines the fundamental principles of the programmable interactions. Additionally, the publication defines a blueprint architecture where two concepts, Social Devices and People as a Service, work seamlessly together, improving the users' experiences about the Internet of Things. Finally, this collaboration proves a valid point that it becomes increasingly important to consider interoperability with other systems. Thus, in its way, this sets the future path for Social Devices research.

Publication [II] introduces how the programmable interactions can be implemented with the Action-Oriented Programming (AcOP) model. The publication updates the original way of implementing interactions in the Social Devices concept to its current state. For the updated AcOP model, a new open-source middleware and runtime environment is introduced in the publication. This new Orchestrator.js is demonstrated with a running example that shows how all the concepts of the AcOP model are implemented with Android and .NET Gadgeteer platforms, leading to a complete programmable interaction.

Publication [III] covers the lessons learned from coordinating the interactions of Social Devices. The publication introduces four implementations and experiments that describe how the coordination aspects have been improved during the years of Social Devices research. The technical contribution of the publication is a capability framework for iOS platform that allows selecting one entity for the role of the coordinator, and then deploying JavaScript-based actions to the device. For this purpose, the communication layer was complemented with Bluetooth Low Energy connectivity, enabling device-to-device communication and coordination. Moreover, at the time Bluetooth beacons were a novel technology, and for this purpose their suitability for improving the proactive detection of Social Devices was studied. The experiments proved that JavaScript is well-suited as a coordination language, and that the devices can coordinate with each other as connectivity support between platforms improves.

Publication [IV] describes the original idea of collaborative and co-located interactions of the Social Devices concept. The publication introduces a couple of example use cases, and an initial proof of concept system named Social Devices Platform (SDP). SDP architecture relies on many cloud-based components, and the scheduling of interactions was complex compared to the new implementation introduced in Publication [II]. The programming language used for programming the interactions in SDP was Python. The device-side capability framework was also based Qt/Python, and was able to operate on Maemo/MeeGo mobile devices, as well as on Windows, Mac OS X, and Linux desktop platforms. The author has implemented SDP's Orchestrator component described in the publication, as well as the device-side capability framework.

Publication [V] reports outside developers' experiences regarding programming interactions with the Action-Oriented Programming model. Previously, AcOP middleware had been used by project members and students working for the department. In order to get feedback from outside developers, we hired a four-person student team from Demola (`http://demola.fi`), an innovation unit allowing students and companies to meet. The team had a project manager, graphical designer, and two undergraduate students as developers. None of the team members had any previous experience on game development, or even Android programming in general. The team was given a short presentation regarding Social Devices and AcOP, and then free hands to implement whatever idea they wanted. The team came up with an idea about a Zombie Olympics game named Apocalympics, which is meant to be played on a bus or other public place with familiar strangers (persons that you only know by their faces). On the following day after the presentation, the team came to show us that they had gotten the environment working and already had some initial graphics implemented. In total, it took about a couple of weeks for them to implement the game. The results of this experiment were encouraging, and in general the students were happy with the programming model and its concepts. Based on the interviews, we prepared claims about AcOP and the Social Devices concept, and later on these claims were discussed in a workshop with other professionals. The final results of the experiment have been reported in this included book chapter.

Publication [VI] describes Mobile Content as a Service approach for vendor-neutral content sharing. As digital content is an essential part of people's daily lives, it becomes essential to consider accessing this content within programmable interactions as well. The publication introduces a vendor-neutral blueprint architecture that enables accessing the content regardless of which device the content actually resides. In short, the idea of this approach is that no content needs to be stored in hardware vendors' clouds or cloud services. Instead, the content should be directly accessible between the devices. For this reason, the content is split into two concepts: metadata and essences. The metadata is shared with other entities via cloud-based service, and the essence, when required, can then be accessed in various ways based on the metadata.

Publication [VII] describes the technical implementation of the system behind Mobile Content as a Service approach. These details help in understanding how the content can actually be shared and utilized within the programmable interactions. The introduced technical implementation, named VisualREST, has been published as open source software, and it can be deployed flexibly: the system can be hosted by home computers, cloud service providers, or it could even be pre-installed on some hardware sold commercially. The publication also describes the most crucial features for the distributed content management system that are based on the literature survey conducted in the candidate's Master of Science thesis [86].

Chapter 7

Conclusions

Interactions in computing are becoming increasingly important. Already a lot of people's time and attention is commanded by the current technology. People are constantly connected and online, communicating remotely with their friends and family. With more and more computing-enabled devices, the way we interact with computing must fundamentally change. This means that devices need to be able to interact with each other more autonomously, and with humans in more user-friendly ways.

This doctoral thesis studied the concept of programmable interactions, and the hypothesis was that human-centric interactions should play the key role in computing. For this purpose, technical solutions were presented to enable such *physical – social interactions*. This chapter summarizes the findings and results of the work. The research questions are revisited, and some final remarks and considerations for future work are presented.

7.1 Summary

As the main theoretical and technical contribution, this thesis presented the Action-Oriented Programming model and its runtime environment implementation. Together they offer abstractions and tools for designing and implementing interactions between co-located entities and people. AcOP is especially targeted to take advantage of the diverse set of devices and heterogeneous networking technologies, but in a human-centric way. The model is flexible and lightweight, and allows delegating tasks among the participants of the interactions. Runtime implementation of AcOP utilizes JavaScript as a coordination language for defining the joint behavior of multiple entities and people with a concept called *actions*. AcoP *apps* support sensing various kinds of events coming from the world, and implementing different scheduling strategies for acting upon these events. The devices can be described in a unified way based on their resources with *capabilities*, which makes it intuitive for developers to utilize the resources within the interactions.

The social aspects and users' personal preferences are considered based on *personal profiles*. These profiles are maintained by *companion devices* that act as virtual representatives and extensions of their users. They are used for composing a *social proximity graph*, which describes the users' relationships to nearby entities and other users. Users' digital content can also be utilized within the interactions. An approach called Mobile Content as a Service was introduced for offering access to the users' content directly from their devices, as well as from different cloud services. Additionally, AcOP allows for observing

and attaching to external services in cyberspace. This helps bridge the gap between the physical and virtual worlds.

AcOP can be approached by people with different levels of technical skill: AcOP apps can simply be turned on/off by end-users. Simple actions can be developed with capabilities defined by other developers, with no need to touch device-end code. Developers with some knowledge about device-end development can implement their own capabilities with readymade capability frameworks. Apps for scheduling interactions can be implemented either based on capability or framework events, but new event types can also be implemented by the developers. Finally, more advanced developers can implement capability and communication frameworks for new hardware platforms.

7.2 Research Questions Revisited

The first chapter introduced the research questions of the thesis, which included one main question and three other questions. As was stated in the first chapter, enabling the programmable interactions is a highly demanding task, and should be considered as a continuous process with many problems to solve and challenges to face. For this reason, the main research question of the thesis was:

What are the challenges of enabling programmable interactions?

The thesis answered the main research question by identifying the challenges from different perspectives. In Chapters 3, 4, and 5 a total of 31 challenges were identified and listed. The identified challenges were then referenced with their identifiers, while describing how the implemented technical solutions respond to these challenges. The presented Action-Oriented Programming model and its runtime environment implementation did respond to most of these identified challenges from a software development perspective. However, there are also other aspects related to programmable interactions, and thus the following three other research questions were presented:

Q1: ***What should the programmable interactions be like, and why?***

The fundamental idea behind programmable interactions is to make interactions with computing more human-centric. For this reason, the programmable interactions are based on four principles: be social (P.1), be personalized (P.2), be proactive (P.3), and be predictable (P.4). The principles (P.2) and (P.4) are relatively easy to adapt and agree with. Being social and proactive, however, often causes contradictory feelings, especially when mentioned together. Being proactive, however, is essentially important while considering the number of computing devices that are expected to emerge. Simply managing them all and initiating each task manually is not an option, so proactive approaches are needed. The principle of being social, on the other hand, has two sides. On one hand, it defines which entities interact with each other, and what resources they share; on the other, the social also refers to the interaction modalities, how the user and the device interact. Considering that the digital assistants on mobile devices and social robotics are already emerging, it could be assumed that in the future interacting with the devices with more social modalities will become the standard way. For this reason, it is important that, in addition to humans, the entities are also able to initiate interactions. Consequently, these four principles are generic guidelines that are important to have in mind while implementing any interactions that employ multiple devices.

Q2: ***How to harness the various resources of each device?***

In order to interact with each other, entities must be able to share and utilize each other's physical and digital resources. Sharing both types of resource requires fast and seamless communication between the entities, taking care of access rights, and describing the resources for intuitive access. Harnessing, however, requires different solutions, as the resource types are fundamentally different. Digital content consists of metadata (describing the content) and essence (which is typically a large binary object). Hardware resources, on the other hand, can be utilized by sending lightweight messages, but accessing them typically happens in high frequency. Conversely, the hardware resources form the platform for utilizing the digital resources. Both types of resources can be harnessed by taking advantage of the pervasive Internet and heterogeneous networks. Cloud services are then used for offering information on how the different resources can be accessed. Based on this information, the actual communication can then be implemented either directly between the entities, or indirectly via cloud-based service. At present, however, sharing both types of resources, digital and physical, requires frameworks that run on the devices since current platforms do not yet offer their resources as easily programmable objects for other entities. In this thesis, *capability frameworks* were proposed as a solution for sharing the hardware resources of the devices, and *container programs* for sharing the digital resources. Although device-to-device communication can improve interactions and sharing resources substantially, the common communication protocols are still missing and support between mobile platforms from different vendors is poor. This, as a result, often encourages utilizing Internet-based communication.

Q3: ***How to enable proactivity in social interactions?***

Current technologies are not targeted for building such proactive and social interactions between co-located entities. Initiating a joint behavior between multiple devices and people requires being aware of these entities. This thesis presented the concept of mobile phones as companion devices. The idea is that people nowadays carry their mobiles nearly everywhere, and thus they can be considered as virtual representatives of their owners and utilized for detecting which other entities (and people) are nearby. For the actual detection, different technologies can be used, but Bluetooth technologies are in general the most feasible ones as they work in any location, indoor and outdoor, and in an ad hoc manner. However, Bluetooth was not originally meant to be utilized in such a way, and thus many issues still exist for detecting the entities in a proactive manner. This means that when the device is running the application in background, or the device is in sleep mode, it becomes much more challenging to detect. Fortunately, Bluetooth technology is developing quickly, and major companies now also have ideas of using this technology for detection of entities. These ideas, however, are different from ideas of programmable interactions as they are not aimed at triggering interactions proactively. In the future, it will be interesting to see if Bluetooth gets to a stage where it can truly foster the triggering of proactive interactions.

7.3 Future Development Ideas

Enabling programmable interactions is a continuous process and improves over time, as was noted already in the first chapter of the thesis. Moreover, presently interactions are a timely topic in computing, and hence new solutions for implementing interactions between

entities are introduced. The following discusses two topics that can be considered highly relevant in the near future.

7.3.1 Ecosystem Interoperability

Over the years, mobile platform vendors have proposed ecosystems for improving interoperability among their devices. However, vendors have typically kept their ecosystems closed with support limited to their own devices, or in some cases, devices branded specifically to have such support. Business decisions often override promotion of interoperability. Recently, however, some changes in the highly limited ecosystem thinking have been seen. For instance, Google has opened its Android platform and offers interoperability support at least for Apple's products. Also, some of Google's research projects have had support for non-Android platforms. As an example, the recently launched Brillo platform can operate with iOS devices with Google's new open Weave protocol. For Microsoft's mobile devices the support, however, is more limited since the marketshare of these devices is small. Microsoft itself, on the other hand, offers many of its application softwares and services for other platforms than their own, like its recently introduced digital assistant.

These examples indicate that the closed ecosystem thinking is changing, and that vendors are now realizing the business opportunities of supporting interoperability with competitors' devices – after all, the world consists of numerous hardware platforms from different vendors, rather than just few big ecosystems. And mobile devices are just a handful of the emerging new smart objects. Most likely in the future the successful vendors will be the ones offering the best interoperability between all the popular devices, and not just interoperability support within their own ecosystem. In the end, people are the ones making the decisions of buying the devices and using the services. Nevertheless, it will still be a long time before vendors can offer common APIs or communication protocols that will enable ecosystem interoperability.

From the programmable interaction perspective, however, improved interoperability and new communication protocols would open up improved possibilities for enabling the interactions, since communication and operations between platforms would then become more seamless. This would not only help in taking more advantage of different resources and capabilities of individual devices, but would eventually also help take advantage of entire ecosystems – the ecosystems already offer services and support for many tasks that can help implement the interactions. Hopefully, in the future all vendors open their platforms or ecosystems, and enable common communications possibilities. The Action-Oriented Programming model could then be applied for programming the interactions in vendor-neutral way. Naturally, there would then be much research to do regarding the runtime implementation of the AcOP model.

7.3.2 Security and Privacy

One key principle of programmable interactions is to be predictable. This includes the fundamental needs for privacy and sense of security – users must be able to trust the system. Programmable interactions involve a lot of personal information, and hence many possible threats are present. Although security and privacy are now more important than ever, only initial ideas for improving these aspects have been discussed in this thesis. Privacy and security are complex issues, and many perspectives need to be considered while developing software. At least laws and regulations, business and profits, user

experiences, and technical solutions all have an influence on privacy and security, and often these aspects also somehow affect each other.

Indeed, security can be improved with traditional methods for encrypting data and securing connections. For instance, vendors of mobile platforms now assure that the data stored in their devices is safe by encrypting it. Moreover, communication protocols (e.g. Google's Weave and Apple's HomeKit) are especially designed to offer highly secure connectivity. These methods, however, become irrelevant if users are willingly giving up their privacy in exchange for services. Vendors now offer services for searching information, storing and editing our personal digital content, making payments, instant messaging, and emailing, just to name few. Additionally, most platforms now trace and report information about user activity in the background, without many of the users even being aware of this. With the physical world now melting into the digital world, a lot of new personal, even intimate, information becomes available to the vendors. As an example, information is already gathered on the health and physical activity of the users. Thus it could be said that the privacy of our digital life is in the hands of these vendors, and that the users have little, if any, control over how the information collected on them is utilized. Inevitably, this raises the question of whether any single entity should have access to such a comprehensive set of information that involves digital and physical, and thus nearly all the aspects of an individual's life?

Naturally, privacy is not only threatened by the ecosystem and platform vendors. Other Internet companies are interested in this information as well. Their access, however, is typically limited to only certain sectors. Considering programmable interactions and any similar concepts, vendor-neutral approaches also have access to an increasing amount of highly personal information. For this reason the providers of the services have a lot of responsibility on how they themselves utilize this information, as well as to keep this information away from criminals. Fortunately, these issues have now started to be taken seriously by authorities. For instance, the European Commission has defined laws and regulations for protecting citizens (e.g. Right to be Forgotten). At the moment, there are several lawsuits against major Internet companies regarding possible privacy violations. The benefit of new laws and these lawsuits is that they make visible what information the companies are gathering about users, as well as making users more aware of their rights to privacy. In the end, this hopefully will lead to clear rules regarding how sensitive data should be handled by all service providers and companies.

The proposed solution in this thesis was to keep the most sensitive data stored inside the user's own device, where it was first produced. The Action-Oriented Programming model supports this approach, since it allows observer components to run in the devices. In such a solution, sensitive data does not have be transferred over the network and stored on a remote system. Instead, the observer components maintain the state locally on the device-end, and when the environment meets the scheduling strategy defined by the developer and approved by the end-user, the action is triggered. Still, however, the support of the AcOP runtime environment for these observers does not handle the data in a secure way, and more research is required for improving these aspects. Also, end-user awareness of how their data is actually utilized could be improved, although already different mechanisms have been implemented.

Privacy must be considered also from a user experience perspective. The programmable interactions are proactive, social, and personalized, which causes many novel privacy issues. When devices are able to initiate such interactions by themselves, the context becomes highly important. As the full context-awareness is somewhat impossible to

achieve, the proposed solution is to be at least socially-aware of the entities nearby. Whereas the social-awareness can be supported to some extent with the social proximity graph and other technical solutions, obviously, however, even this type of awareness is hard to achieve, since the devices are not capable of detecting all the entities, and indeed not all the humans. Hence, in the end, it is the responsibility of the programmable interaction developer to implement the scheduling in predictable way, and consider what kind of interactions should be implemented in the first place. The programming model and runtime can only support the developer in implementing the scheduling strategies.

Indeed, there is much more research to be done to offer support for implementing better strategies, and to protect users from unwanted behavior. One possible solution could be that the programming model would offer new types of programming primitives that could help define the nature of interactions, and the runtime environment would then automatically prevent the interactions from taking place if the user's environment is wrong or preferences deny triggering certain interactions. The current AcOP action precondition method already does this, but it may be too limited for the task. Moreover, the current solution leaves too many possibilities for developers to implement unwanted behavior. Another way of improving the accuracy of the triggering is to improve the personal profiles. Although there are obvious privacy concerns regarding the personal profile stored in the user's companion device, eventually these profiles can actually help improve the privacy since they help to define what kind of interactions and in which environment the user prefers. However, more research is needed for making these profiles more accurate, and to allow the profiles to learn from the user's behavior, as well as from mistakenly triggered interactions.

Since the programmable interactions leave so many possible security and privacy threats open, any popular system offering such interactions would require supervising what kind of interactions are developed. To some extent, this could be done in an automated way by analyzing code, but any suspicious implementations should be inspected by professionals.

Bibliography

[1] Aaltonen, T., Myllärniemi, V., Raatikainen, M., Mäkitalo, N., Pääkkö, J.: An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud. In: 20th Asia-Pacific Software Engineering Conference, APSEC'13 (2013)

[2] Aggarwal, A., Naguib, A., Sridhara, V., Das, S.: Wireless position determination using adjusted round trip time measurements (2012). URL `http://www.google.com/patents/EP2368131B1`. EP Patent 2,368,131

[3] Aguilar, D.P.: Framework de juegos para móviles basados en Social Devices (Framework for Mobile Games based on Social Devices). Master of Science thesis, University of Málaga (2014)

[4] Ahuja, S., Carriero, N., Gelernter, D.: Linda and friends. Computer **19**(8), 26–34 (1986)

[5] Akyildiz, I.F., Stuntebeck, E.P.: Wireless underground sensor networks: Research challenges. Ad Hoc Networks **4**(6), 669 – 686 (2006)

[6] Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: A survey. Comput. Netw. **38**(4), 393–422 (2002)

[7] Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Computer Networks **54**(15), 2787 – 2805 (2010)

[8] Avancha, S., Joshi, A., Finin, T.: Enhanced service discovery in bluetooth. Computer **35**(6), 96–99 (2002)

[9] Back, R.J., Kurki-Suonio, R.: Distributed cooperation with action systems. TOPLAS, ACM **10**(4), 513–554 (1988)

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

[11] Benincasa, G., Morelli, A., Stefanelli, C., Suri, N., Tortonesi, M.: Agile communication middleware for next-generation mobile heterogeneous networks. Software, IEEE **31**(2), 54–61 (2014)

[12] Benítez, J.A.C.: Emerging models for the development of social mobile applications: People as a Service, and Social Devices. A Proof of Concept. Master of Science thesis, University of Málaga (2014)

[13] Berger, S., McFaddin, S., Narayanaswami, C., Raghunath, M.: Web services on mobile devices-implementation and experience. In: Mobile Computing Systems and Applications, 2003. Proceedings. Fifth IEEE Workshop on, pp. 100–109 (2003)

[14] Blair, K., Murphy, R., Almjeld, J.: Cross Currents: Cultures, Communities, Technologies. Cengage Learning (2013)

[15] Bosch, J., Bosch-Sijtsema, P.: From integration to composition: On the impact of software product lines, global development and ecosystems. J. Syst. Softw. **83**(1), 67–76 (2010)

[16] Broll, G., Rukzio, E., Paolucci, M., Wagner, M., Schmidt, A., Hussmann, H.: Perci: Pervasive service interaction with the internet of things. Internet Computing, IEEE **13**(6), 74–81 (2009)

[17] Brønsted, J., Hansen, K., Ingstrup, M.: Service composition issues in pervasive computing. Pervasive Computing, IEEE **9**(1), 62–70 (2010)

[18] Bureau, I.T.D.: ICT Facts and Figures. `https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf` (2013). [Online; accessed 20-Jan-2015]

[19] Business Process Model and Notation (BPMN). `http://www.omg.org/spec/BPMN/2.0/PDF/` (2011). [Online; accessed 30-March-2015]

[20] Chard, K., Bubendorfer, K., Caton, S., Rana, O.: Social cloud computing: A vision for socially motivated resource sharing. Services Computing, IEEE Transactions on **5**(4), 551–563 (2012)

[21] Constrained Application Protocol (CoAP): `http://tools.ietf.org/html/draft-ietf-core-coap`. [Online; accessed 5-March-2015]

[22] Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design (4th Edition) (International Computer Science). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2005)

[23] Crane, D., McCarthy, P.: Comet and Reverse Ajax: The Next-Generation Ajax 2.0. Apress, Berkely, CA, USA (2008)

[24] Crockford, D.: JavaScript: The Good Parts. O'Reilly Media, Inc. (2008)

[25] Darling, K.: Why Google's Robot Personality Patent Is Not Good for Robotics. `http://spectrum.ieee.org/automaton/robotics/robotics-software/why-googles-robot-personality-patent-is-not-good-for-robotics` (2015). [Online; accessed 23-Jul-2015]

[26] Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)

[27] Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering, pp. 285–311. Springer (2008)

[28] Ebeling, B., Hoyer, S., Bührig, J.: What are your Favorite Methods? - an Examination on the Frequency of Research Methods for is Conferences from 2006 to 2010. In: The 20th European Conference on Information Systems (ECIS'12) (2012)

[29] ECMA International: ECMA-262: ECMAScript Language Specification 2015, 6th Edition (2015)

[30] Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The Many Faces of Publish/Subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)

[31] Fakoor, R., Raj, M., Nazi, A., Di Francesco, M., Das, S.K.: An integrated cloud-based framework for mobile phone sensing. In: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, pp. 47–52. ACM, New York, NY, USA (2012)

[32] Ferreira, P., Martinho, R., Domingos, D.: IoT-aware business processes for logistics: limitations of current approaches. In: Inforum, vol. 3, pp. 612–613 (2010)

[33] Fielding, R.T.: REST: architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, Irvine (2000)

[34] Fowler, D., Pitta, D.: Advances in mining social media: Implications for marketers. Journal of Information Technology Management **24**(4), 26 (2013)

[35] Francis, A., Lewis, T.: Methods and systems for robot personality development (2015). URL `https://www.google.com.ar/patents/US8996429`. US Patent 8,996,429

[36] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)

[37] Ganz, F., Puschmann, D., Barnaghi, P., Carrez, F.: A practical evaluation of information processing and abstraction techniques for the internet of things. Internet of Things Journal, IEEE **PP**(99), 1–1 (2015)

[38] Gelernter, D., Carriero, N.: Coordination Languages and Their Significance. Commun. ACM **35**(2), 97–107 (1992)

[39] Giulia, B.: A semantic model for socially aware objects. Advances in Internet of Things **2012** (2012)

[40] Gohil, A., Modi, H., Patel, S.: 5G technology of mobile communication: A survey. In: Intelligent Systems and Signal Processing (ISSP), 2013 International Conference on, pp. 288–292 (2013)

[41] Grau, A.: How to Build a Safer Internet of Things. `http://spectrum.ieee.org/telecom/security/how-to-build-a-safer-internet-of-things` (2015). [Online; accessed 25-Feb-2015]

[42] Group, G.: Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. `http://www.gartner.com/newsroom/id/2636073` (2013). [Online; accessed 20-Nov-2015]

[43] Grudin, J.: Computer-Supported Cooperative Work: History and Focus. Computer **27**(5), 19–26 (1994)

[44] Guillen, J., Miranda, J., Berrocal, J., Garcia-Alonso, J., Murillo, J., Canal, C.: People as a Service: A Mobile-centric Model for Providing Collective Sociological Profiles. Software, IEEE **31**(2), 48–53 (2014)

[45] Guinard, D.: A Web of Things Application Architecture - Integrating the Real-World into the Web. Ph.D. thesis, ETH Zurich (2011)

[46] Guo, B., Zhang, D., Yu, Z., Zhou, X., Zhou, Z.: Enhancing spontaneous interaction in opportunistic mobile social networks. Communications in Mobile Computing **1**(1), 6 (2012)

[47] Hall, E.T.: The Hidden Dimension. Anchor (1990)

[48] Hall, J.K., Kiyoki, Y.: Creating a personal-context oriented real-time dynamic and collaborative space. In: P. Vojtás, Y. Kiyoki, H. Jaakkola, T. Tokuda, N. Yoshida (eds.) EJC, *Frontiers in Artificial Intelligence and Applications*, vol. 251, pp. 82–101. IOS Press (2012)

[49] Hartman, J., Manber, U., Peterson, L., Proebsting, T.: Liquid software: A new paradigm for networked systems. Tech. rep., Technical Report 96 (1996)

[50] Hasan, S., Curry, E.: Thingsonomy: Tackling variety in internet of things events. Internet Computing, IEEE **19**(2), 10–18 (2015)

[51] Hassmen, P., Koivula, N., Uutela, A.: Physical exercise and psychological well-being: a population study in finland. Preventive medicine **30**(1), 17–25 (2000)

[52] Henson, C., Sheth, A., Thirunarayan, K.: Semantic Perception: Converting Sensory Observations to Abstractions. Internet Computing, IEEE **16**(2), 26–34 (2012)

[53] Henson, C., Thirunarayan, K., Sheth, A.: An ontological approach to focusing attention and enhancing machine perception on the web. Appl. Ontol. **6**(4), 345–376 (2011)

[54] Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Q. **28**(1), 75–105 (2004)

[55] Hodges, S., Villar, N., Scott, J., Schmidt, A.: A New Era for Ubicomp Development. Pervasive Computing, IEEE **11**(1), 5–9 (2012)

[56] Honkanen, M., Lappetelainen, A., Kivekas, K.: Low end extension for bluetooth. In: Radio and Wireless Conference, 2004 IEEE, pp. 199–202 (2004)

[57] Hu, X., Li, X., Ngai, E.H., Leung, V., Kruchten, P.: Multidimensional context-aware social network architecture for mobile crowdsensing. Communications Magazine, IEEE **52**(6), 78–87 (2014)

[58] Huang, A.S., Rudolph, L.: Bluetooth Essentials for Programmers, 1 edn. Cambridge University Press (2007)

[59] Huang, D., Xing, T., Wu, H.: Mobile cloud computing service models: a user-centric approach. Network, IEEE **27**(5), 6–11 (2013)

[60] (IETF), I.E.T.F.: IETF the websocket protocol. `http://tools.ietf.org/html/rfc6455`. [Online; accessed 5-Nov-2015]

[61] Internet Engineering Task Force (IETF): RFC 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2) (2015)

[62] Jabber Software Foundation: RFC 3920 - Extensible Messaging and Presence Protocol (XMPP) (2004)

[63] Jansen, M.: About an architecture that allows to become a mobile web service provider. In: ICIW 2012, The Seventh International Conference on Internet and Web Applications and Services, pp. 90–96 (2012)

[64] Jansen, M.: About using mobile devices as cloud service providers. In: CLOSER'12, pp. 147–152 (2012)

[65] Jarusriboonchai, P., Olsson, T., Malapaschas, A., Väänänen, K.: Increase Collocated People's Awareness of Mobile User's Activities: Field Trial of Social Display. In: The 19th ACM conference on Computer-Supported Cooperative Work and Social Computing, CSCW'16, pp. 1691–1702 (2016)

[66] Jarusriboonchai, P., Olsson, T., Prabhu, V., Väänänen-Vainio-Mattila, K.: Cuesense: A wearable proximity-aware display enhancing encounters. In: Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems, Seoul, CHI 2015 Extended Abstracts, Republic of Korea, April 18 - 23, 2015, pp. 2127–2132 (2015)

[67] Jarusriboonchai, P., Olsson, T., Väänänen-Vainio-Mattila, K.: User Experience of Proactive Audio-based Social Devices: A Wizard-of-oz Study. In: Proceedings of the 13th International Conference on Mobile and Ubiquitous Multimedia, MUM '14, pp. 98–106. ACM, New York, NY, USA (2014)

[68] Järvinen, H.J., Kurki-Suonio, R., Sakkinen, M., Systä, K.: Object-oriented specification of reactive systems. In: Proceedings of ICSE'90 (1990)

[69] Järvinen, P.: Action research as an approach in design science. European Academy of Management. Citeseer, Munich (2005)

[70] Jazayeri, M.: Some Trends in Web Application Development. In: Future of Software Engineering, 2007. FOSE '07, pp. 199–213 (2007)

[71] Jiang, D., Delgrossi, L.: Ieee 802.11p: Towards an international standard for wireless access in vehicular environments. In: Vehicular Technology Conference, 2008. VTC Spring 2008. IEEE, pp. 2036–2040 (2008)

[72] Jiang, J.R., Lin, B.R., Tseng, Y.C.: Analysis of Bluetooth Device Discovery and Some Speedup Mechanisms. Journal of Electrical Engineering **11**(4), 301–310 (2004)

[73] Kaur, N.: An introduction to wireless mobile social networking in opportunistic communication. International Journal of Distributed & Parallel Systems **4**(3) (2013)

[74] Kelloniemi, A.: Social Devices Client for Arduino. Master of Science thesis, Tampere University of Technology (2014)

[75] Kindberg, T., Fox, A.: System Software for Ubiquitous Computing. IEEE Pervasive Computing **1**(1), 70–81 (2002)

[76] Kurkovsky, S., Kurkovsky, S.: Multimodality in Mobile Computing and Mobile Devices: Methods for Adaptable Usability, 1st edn. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA (2009)

[77] Lagerspetz, E., Tarkoma, S.: Mobile search and the cloud: The benefits of offloading. In: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on, pp. 117–122 (2011)

[78] Latronico, E., Lee, E., Lohstroh, M., Shaver, C., Wasicek, A., Weber, M.: A vision of swarmlets. Internet Computing, IEEE **19**(2), 20–28 (2015)

[79] Laukkarinen, T.: Abstracting Application Development for Resource Constrained Wireless Sensor Networks. Ph.D. thesis, Tampere University of Technology (2015)

[80] Lee, E.: Cyber physical systems: Design challenges. In: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on, pp. 363–369 (2008)

[81] Licklider, J.C.R.: Man-computer symbiosis. IRE Transactions on Human Factors in Electronics **HFE-1**, 4–11 (1960)

[82] Lin, K.J., Chen, C.W., Chou, C.F.: Preference-aware content dissemination in opportunistic mobile social networks. In: INFOCOM, 2012 Proceedings IEEE, pp. 1960–1968 (2012)

[83] Loui, R.: In praise of scripting: Real programming pragmatism. Computer **41**(7), 22–26 (2008)

[84] Lucero, A., Keränen, J., Korhonen, H.: Collaborative use of mobile phones for brainstorming. In: Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services, MobileHCI '10, pp. 337–340. ACM, New York, NY, USA (2010)

[85] Lyon, J., Lau, J., Sarin, R., Park, J., Veron, M.: Controlling audio of a device (2012). URL `http://www.google.com/patents/US20120231838`. US Patent App. 13/046,680

[86] Mäkitalo, N.: RESTful content management system for distributed environment. Master of Science thesis, Tampere University of Technology (2011)

[87] Mäkitalo, N., Aaltonen, T., Mikkonen, T.: Coordinating Proactive Social Devices in a Mobile Cloud: Lessons Learned and a Way Forward, (Accepted). In: Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft 2016). ACM (2016)

[88] Mäkitalo, N., Peltola, H., Turto, T., Mikkonen, T., Savolainen, J.: Complementing web service architecture. In: P. Vojtás, Y. Kiyoki, H. Jaakkola, T. Tokuda, N. Yoshida (eds.) Information Modelling and Knowledge Bases XXIV, *Frontiers in Artificial Intelligence and Applications*, vol. 251, pp. 23–30. IOS Press (2013)

[89] Marquardt, N., Greenberg, S.: Informing the Design of Proxemic Interactions. Pervasive Computing, IEEE **11**(2), 14–23 (2012)

[90] Mauthe, A., Thomas, P.: Professional content management systems: handling digital media assets. John Wiley & Sons, West Sussex, England (2004)

[91] Mikkonen, T., Salminen, A.: Implementing mobile mashware architecture: downloadable components as on-demand services. Procedia Computer Science **10**, 553–560 (2012)

[92] Mikkonen, T., Taivalsaari, A.: Reports of the web's death are greatly exaggerated. Computer **44**(5), 30–36 (2011)

[93] Millard, P., Saint-Andre, P., Meijer, R.: XMPP Standards Foundation. XEP-0060: Publish-Subscribe. `http://xmpp.org/extensions/xep-0060.html`. [Online; accessed 15-March-2015]

[94] Neisser, U.: Cognition and Reality: Principles and Implications of Cognitive Psychology. Books in psychology. W. H. Freeman (1976)

[95] Network Working Group: A Universally Unique IDentifier (UUID) URN Namespace (2005)

[96] Newmarch, J.: A RESTful approach: clean UPnP without SOAP. In: Consumer Communications and Networking Conference, 2005. CCNC. 2005 Second IEEE, pp. 134–138 (2005)

[97] Nielsen, J., Rock, L., Rogers, B., Dalia, A., Adams, J., Chen, Y.Q.: Automated social coordination of cyber-physical systems with mobile actuator and sensor networks. In: Mechatronics and Embedded Systems and Applications (MESA), 2010 IEEE/ASME International Conference on, pp. 554–559 (2010)

[98] Pääkkö, J., Raatikainen, M., Myllärniemi, V., Männistö, T.: Applying Recommendation Systems for Composing Dynamic Services for Mobile Devices. In: Proceedings of APSEC'2012 (2012)

[99] Parameswaran, M., Whinston, A.B.: Social computing: An overview. Communications of the Association for Information Systems **19**(1), 37 (2007)

[100] Peltola, H.: Utilizing External Services and Sharing Content in a Content Management System. Master of Science thesis, Tampere University of Technology (2012)

[101] Peska, L., Lasek, I., Eckhardt, A., Dedek, J., Vojtás, P., Fiser, D.: Towards web semantization and user understanding. In: P. Vojtás, Y. Kiyoki, H. Jaakkola, T. Tokuda, N. Yoshida (eds.) EJC, *Frontiers in Artificial Intelligence and Applications*, vol. 251, pp. 63–81. IOS Press (2012)

[102] Pramudianto, F., Kamienski, C.A., Souto, E., Borelli, F., Gomes, L.L., Sadok, D., Jarke, M.: IoTLink: An Internet of Things Prototyping Toolkit. IEEE (2014)

[103] Pramudianto, F., Simon, J., Eisenhauer, M., Khaleel, H., Pastrone, C., Spirito, M.: Prototyping the Internet of Things for the future factory using a SOA-based middleware and reliable WSNs. In: Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on, pp. 1–4 (2013)

[104] Raatikainen, M., Myllärniemi, V., Ghosh, S., Pääkkö, J., Männistö, T., Ylikangas, M., Korjus, O., Uusitalo, E., Mäkitalo, N., Peltola, H., et al.: Towards Mobile Device Cloud. Communications of the Cloud Software **1**(1) (2011)

[105] Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly, Sebastopol, CA, US (2007)

[106] Rivest, R.: RFC 1321: The MD5 message-digest algorithm, April 1992 (1992)

[107] Roberts, L.: The arpanet and computer networks. In: Proceedings of the ACM Conference on The history of personal workstations, pp. 51–58. ACM (1986)

[108] Rogers, Y.: Moving on from Weiser's Vision of Calm Computing: Engaging Ubicomp Experiences. In: Proceedings of the 8th International Conference on Ubiquitous Computing, UbiComp'06, pp. 404–421. Springer-Verlag, Berlin, Heidelberg (2006)

[109] Russell, M., Russell, M.: Mining the Social Web: Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social Media Sites. Head First Series. O'Reilly Media, Incorporated (2011)

[110] Saint-Andre, P., Smith, K., TronCon, R.: XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies, 1 edn. O'Reilly Media, Sebastopol, CA, US (2009)

[111] Satyanarayanan, M., Lewis, G., Morris, E., Simanta, S., Boleng, J., Ha, K.: The role of cloudlets in hostile environments. Pervasive Computing, IEEE **12**(4), 40–49 (2013)

[112] Schmidt, A., Doring, T., Sylvester, A.: Changing How We Make and Deliver Smart Devices: When Can I Print Out My New Phone? Pervasive Computing, IEEE **10**(4), 6–9 (2011)

[113] Schneider, J., Lumpe, M., Nierstrasz, O.: Agent coordination via scripting languages. In: Coordination of Internet Agents: Models, Technologies, and Applications, pp. 153–175. Springer (2001)

[114] Schuler, D.: Social computing. Communications of the ACM **37**(1), 28–29 (1994)

[115] Sha, L., Gopalakrishnan, S., Liu, X., Wang, Q.: Cyber-Physical Systems: A New Frontier. In: Sensor Networks, Ubiquitous and Trustworthy Computing, 2008. SUTC '08. IEEE International Conference on, pp. 1–9 (2008)

[116] Sheth, A., Anantharam, P., Henson, C.: Physical-cyber-social computing: An early 21st century approach. IEEE Intelligent Systems **28**(1), 78–82 (2013)

[117] Sheth, A.P.: Computing for human experience: Semantics-empowered sensors, services, and social computing on the ubiquitous web. IEEE Internet Computing **14**(1), 88–91 (2010)

[118] SMPTE/EBU: Task Force for Harmonized Standards for the Exchange of Programme Material as Bitstreams–Final Report: Analyses and Results (1998)

[119] Socket.IO protocol specification. `https://github.com/automattic/socket.io-protocol`. [Online; accessed 29-May-2015]

[120] Sosinsky, B.: Networking Bible. Bible. Wiley (2009)

[121] Spectrum, I.: IBM Bets $3 Billion on the Internet of Things. `http://spectrum.ieee.org/tech-talk/telecom/internet/ibm-bets-3-billion-on-the-internet-of-things` (2015). [Online; accessed 20-March-2015]

[122] Statista: Number of apps available in leading app stores as of July 2015. `http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/` (2015). [Online; accessed 20-Aug-2015]

[123] Stirbu, V., Aaltonen, T.: Enabling Real-Time Resource Oriented Architectures with REST Observers. In: C. Pautasso, E. Wilde, R. Alarcon (eds.) REST: Advanced Research Topics and Practical Applications, pp. 51–67. Springer New York (2014)

[124] Susman, G.I., Evered, R.D.: An assessment of the scientific merits of action research. Administrative science quarterly pp. 582–603 (1978)

[125] Szoniecky, S., Hachour, H., Bouhai, N.: The role of semantic topology in sense-making processes: using metalanguages for knowledge representation. In: 22nd European-Japanese Conference on Information Modeling and knowledge bases, pp. pp. 244–257. Prague, Czek Republic (2012)

[126] Taivalsaari, A., Mikkonen, T.: The web as an application platform: The saga continues. In: Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on, pp. 170–174 (2011)

[127] Taivalsaari, A., Mikkonen, T., Anttonen, M., Salminen, A.: The Death of Binary Software: End User Software Moves to the Web. In: Creating, Connecting and Collaborating through Computing (C5), 2011 Ninth International Conference on, pp. 17–23 (2011)

[128] Taivalsaari, A., Mikkonen, T., Systä, K.: Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In: Proceedings of COMPSAC'2014 (2014)

[129] Talcott, C.: Cyber-Physical Systems and Events. In: M. Wirsing, J.P. Banâtre, M. Hölzl, A. Rauschmayer (eds.) Software-Intensive Systems and New Computing Paradigms, *Lecture Notes in Computer Science*, vol. 5380, pp. 101–115. Springer Berlin Heidelberg (2008)

[130] Tennenhouse, D.: Proactive Computing. Commun. ACM **43**(5), 43–50 (2000)

[131] Triglianos, V., Pautasso, C.: asqium: A javascript plugin framework for extensible client and server-side components. In: 15th International Conference on Web Engineering (ICWE 2015). Springer, Springer, Rotterdam, NL (2015)

[132] Väänänen-Vainio-Mattila, K., Olsson, T., Laaksonen, J.: An exploratory study of user-generated spatial gestures with social mobile devices. In: Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, MUM '12, pp. 11:1–11:4. ACM, New York, NY, USA (2012)

[133] Venkitaraman, N.: Wide-Area Media Sharing with UPnP/DLNA. In: Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE, pp. 294–298 (2008)

[134] Viestintävirasto: Viestintäpalveluiden kuluttajatutkimus. `https://www.viestintavirasto.fi/attachments/toimialatieto/Viestintapalvelujen_kuluttajatutkimus_2015.pdf` (2015). [Online; accessed 1-Sep-2015]

[135] Villar, N., Scott, J., Hodges, S., Hammil, K., Miller, C.: .NET gadgeteer: a platform for custom devices. In: Proceedings of the 10th international conference on Pervasive Computing, Pervasive'12, pp. 216–233. Springer-Verlag, Berlin, Heidelberg (2012)

[136] (W3C), W.W.W.C.: W3C The WebSocket API. `http://www.w3.org/TR/websockets/`. [Online; accessed 5-Nov-2015]

[137] (W3C), W.W.W.C.: Simple Object Access Protocol (SOAP) 1.2. http://www.w3.org/ (2007). [Online; accessed 5-Nov-2015]

[138] Wang, F.Y., Carley, K.M., Zeng, D., Mao, W.: Social Computing: From Social Informatics to Social Intelligence. IEEE Intelligent Systems **22**(2), 79–83 (2007)

[139] Wang, T., Niu, C., Cheng, L.: A Two-Phase Context-Sensitive Service Composition Method with the Workflow Model in Cyber-Physical Systems. In: Proceedings of IEEE 17th International Conference on Computational Science and Engineering (CSE 2014), pp. 1475–1482 (2014)

[140] Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf (2007). [Online; accessed 30-March-2015]

[141] Wei, X., Fan, J., Lu, Z., Ding, K.: Application scheduling in mobile cloud computing with load balancing. Journal of Applied Mathematics **2013** (2013)

[142] Weiser, M.: The Computer for the 21st Century. Scientific American **265**(3), 66–75 (1991)

[143] Wieringa, R.: Design science methodology: Principles and practice. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pp. 493–494. ACM, New York, NY, USA (2010)

[144] Xiao, M., Wu, J., Huang, L.: Community-Aware Opportunistic Routing in Mobile Social Networks. Computers, IEEE Transactions on **63**(7), 1682–1695 (2014)

[145] Yin, R.: Case Study Research: Design and Methods. Applied Social Research Methods. SAGE Publications (2003)

Publications

I

Publication I

J. Miranda, N. Mäkitalo, J. Garcia-Alonso, J. Berrocal, T. Mikkonen, C. Canal, and J. M. Murillo. From the Internet of Things to the Internet of People. In *IEEE Internet Computing*, Volume 19, Issue 12, pages 40 – 47, March, 2015, IEEE Computer Society.

From the Internet of Things to the Internet of People

Javier Miranda, Gloin, Niko Mäkitalo, Tampere University of Technology, Jose Garcia-Alonso, University of Extremadura, Javier Berrocal, University of Extremadura, Tommi Mikkonen, Tampere University of Technology, Carlos Canal, University of Malaga, and Juan M. Murillo, University of Extremadura

Abstract

There is growing interest in developing applications on the Internet of Things. These applications' main objective is to integrate technology into people's everyday life in order to be of service to them en masse. The form in which this integration is implemented, however, still leaves much room for improvement. Usually, the user needs to set parameters within the application. When the person's context changes, they have to manually re-configure the parameters. What was meant to be a commodity, in an unforeseen situation then becomes extra noise. This paper describes a reference architecture that improves how people are integrated with the Internet of Things, with smartphones doing the connecting. The resulting integration opens the way to new IoT scenarios supporting evolution towards the Internet of People.

Keywords: Internet of Things, Internet of People, mobile devices, sociological profiles, pervasive computing, sensors, adaptation to user state, affective computing

1 Introduction

Today's availability of the Internet almost everywhere has favoured the emergence of all kinds of devices equipped with a connecting interface. Linking those and other everyday physical objects to the Net is getting ever easier, thus enhancing the popularity of the Internet of Things (IoT). One of the main goals behind such smart integration of devices is to simplify people's lives by having technology work for them seamlessly [1]. For instance, you can remotely switch on your house's air-conditioning system using your smartphone to get a comfortable temperature when arriving home. Or you might schedule this task in accordance with your daily routine.

However, how IoT technologies are currently integrated with humans still leaves much room for improvement [2]. The technology has yet to develop suitable mechanisms to properly adapt to people's context or mood. Instead, far from making the technology working for people, people are forced to either change their context to fit technological requirements, or to be slavishly aware of the system so as to send it commands or modify their schedule if their routine changes.

In a more desirable IoT scenario, technology would take people's context into account, learn from it, and take proactive steps according to their situation and expectations, avoiding user intervention as much as possible. Thus, if someone plans to arrive home late, they would like the air-conditioning kept switched off until they are actually on their way back home.

Enabling such scenarios requires moving from the Internet of Things to the *Internet of People* (IoP). This article proposes an infrastructure supporting this evolution, and making it possible to construct software for it. In our proposal, smartphones play a central role, reflecting their current use as the main interface allowing people to be connected to the Internet. Thus, we advocate endowing smartphones with a set of capabilities that improve the connection between people and the IoT. First, the smartphone needs to be capable of learning about its owner and their context by constructing their digital profile (understood as the whole of the context information managed by the device). Second, it needs to transparently negotiate and propose interactions with other devices on the Internet, reacting to stimuli and handling relationships. Finally, it needs to be able to manage digital profiles and act accordingly, providing its owner's context as a service for others, and scanning for services that might be of interest to its owner or to update his or her digital profile.

The infrastructure to support the IoP is managed through a combination of the Social Devices [3] and People as a Service (PeaaS) [4] platforms. Social Devices enhances the proactive capabilities of smartphones to orchestrate their interactions with other devices connected to the IoT. PeaaS provides

smartphones with serving capabilities that allow people to offer services from their devices, including providing their context and sociological profile. The combination of the two contributes to constructing an IoP by allowing people's context information to be included in the coordination and interaction management of IoT devices. The main benefit is the integration of people as first-class citizens in the IoT universe. This opens the way to the development of new kinds of human-friendly services and applications in the field.

2 People are not Things

The IoT is increasingly pervading our daily lives. Currently 12.6 billion connected devices are estimated [5], including people, processes, data, and things. This last "connected things" group includes not only mobile devices such as smartphones or tablets, but also desktop and laptop PCs, printers, smart TVs, cars, refrigerators, smart light bulbs, etc. IoT numbers are increasing at a dizzying pace. Experts [6] forecast 25 billion connected devices in 2020. Even devices that do not come equipped with a connecting interface can easily be integrated into an IoT system using such hardware platforms as Arduino, .NET Gadgeteer, or even Lego Mindstorms for kids.

The complexity and heterogeneity of such a large variety of smart devices that can be connected to the Internet requires specific tools to manage them. Current approaches, at least within the industry, have merely been based on offering remote interfaces and endpoints to configure and manage devices, leading to the "Basket of Remotes" problem [7].

The need to combine interactions between several devices led to the development of Machine-to-Machine (M2M) communications. There have been attempts at standardization, such as DPWS [8] and CoAP [9]. While user-friendlier approaches, such as Apple's HomeKit, help integrating home automation via smartphones, they are still in the development stage.

For practical applications, individual gadgets alone are insufficient. Instead, many real-life applications connect devices into cloud computing and storage facilities, enabling the creation of arbitrarily complex systems in which nodes, ranging from sensors to data centres, perform computations. This architecture is a natural starting point for creating mashups that combine data from various sources into new, compelling ways to consume those data. It has, for instance, rapidly become a practical way to keep track of personal sports activities and interests.

However, although the problem of accessing, programming, and combining IoT objects can be settled with a cloud-based architecture, making them accessible to a broad spectrum of users, the way that IoT devices are related to people, still requires more attention. Current tools and technologies supporting the interaction between humans and IoT force the user to adapt to technology, instead of making technology adapt and assist the user. As the user's circumstances change, a conductor is required to manage the orchestra of IoT-linked devices, and reduce the complexity of user interaction with them. In our opinion, such a role can be successfully performed by smartphones. They are the natural interfaces to their owners, connecting them to the outside world. Due to their intrinsically personal nature, smartphones are almost always on their owner's person, and their capabilities, geolocating in particular, can be used to learn about the person's context, i.e., to detect how the person behaves when they are at certain locations, surrounded by certain other people, and so on. Combined with smartphones' growing computing capabilities, it makes them ideal for learning about people and orchestrating their surrounding IoT. Still, every interaction involving people should be casual and hassle-free, giving the user full control of private data so that the interactions are perceived as taking place between the people, and not between the things, connected to the Internet.

3 Background (sidebar)

3.1 Social Devices

Social Devices was first introduced in 2011 as a joint work by Nokia Research Center, Aalto University, and Tampere University of Technology [3]. The motivation behind the model was that smartphones have not only a lot of information about their owners, but also modalities that enable them to resemble humans. They can translate text into speech, for example. At present, the Social Devices concept is supported by a middleware platform named Orchestrator.js (`http://www.orchestratorjs.org`). This allows

proactive triggering of interactions between devices of co-located people. Additionally, it offers a complete set of Web-based tools to define the interactions and their triggering contexts. Currently, the middleware allows custom IoT smart objects to be constructed using Arduino, Raspberry Pi, or .NET Gadgeteer. Work is on-going to extend the support to new platforms since the goal of Social Devices is to allow heterogeneous multi-user and multi-device applications to be created for any type of device.

Various IoT-related scenarios have been implemented and introduced using Social Devices. One example is The Social Coffee Machine. This, based on user context, asks if the user wants to have coffee, and then, when the coffee is ready, invites other people to join the coffee break.

3.2 People as a Service

People as a Service (PeaaS) [4] is a mobile-centric computing model that allows a user's sociological profile to be generated, kept, and securely provided as a service to third parties directly from a smartphone.

PeaaS emphasizes smartphones' capabilities, and relies on them for inferring and sharing sociological profiles. These are kept on the device, making it easier for the owner to keep their virtual identity under their own control while still allowing external systems to consult those identities.

Serving individuals' virtual sociological profiles through smartphones goes one step beyond other mobile-centric models that only serve data such as GPS coordinates and temperature. PeaaS allows a variety of information to be collected, such as the moods, trends, social statuses, and health habits of a group of people, in order to define their digital projection. However, filtering and analysing that information to infer user's characteristics or to extract useful data is not a trivial task. Different techniques, including activity recognition approaches [10] and affective computing [11], are considered in PeaaS for building the richest sociological profile as possible.

Current mechanisms used to specify the orchestration of smart-things could take advantage of the final user's sociological profile to suggest (or even automatically provide) a customized IoT workflow in accordance with the user's likes and preferences. Similarly, in events in a smart-city context, the set of sociological profiles extracted from a crowd could be applied to automatically redirect people so as to avoid unnecessary agglomerations.

The PeaaS model is currently implemented as component of a mobile notification platform named nimBees (`http://www.nimbees.com/`), which provides smart push notifications with advanced segmentation capabilities based on user's sociological profile.

4 Internet of People Manifesto

As noted above, current IoT technology is in need of people-centric enhancements. Our proposal is intended to be a step in this direction. It conforms to a set of features that we believe to be essential foundations for any approach to the IoP. In this section, we shall put forward four guidelines for an IoP Manifesto in terms of the desirable service composition goals for any pervasive computing context (context awareness, contingencies management, heterogeneity and empowering of the users), as set out in [12].

4.1 Be Social

Interactions in which things, devices, and people participate must be social. In particular, the IoP should allow for heterogeneity by supporting the different types of devices that people use, and let them interact with each other and with people more socially than does the IoT. Devices will have to be context-aware and capable of automatically adapting their own and other devices' social behaviour. Users need to be empowered to adjust their preferences and policies about when and with whom their devices are socializing, and with which modalities.

4.2 Be Personalized

Interactions between devices must be personalized to the sociological profiles and contexts of their users, allowing for contingencies, and providing a transparent mechanism for this customization. The sociological profiles of all participating people must be taken into account. Again, users must be empowered to adjust their preferences in order to control how other IoP stakeholders use their profile.

4.3 Be Proactive

The triggering of interactions must be proactive, not manually commanded by the user. Most IoT scenarios today only consider remote interfaces for managing connected things. But ever more things online mean more distractions and work in managing them. The IoP should allow for device heterogeneity to allow them all to interact more proactively. Users must be empowered to adjust their preferences to control how proactive their surrounding things are. It should be noted than letting devices to act proactively could entail security risks that must be analysed and delimited, keeping users informed of any established proactivity policy.

4.4 Be Predictable

Interactions must be predictable, i.e., they should be triggered according to a predictable context that the user had previously identified, and for which a specific behaviour had been defined. Users must be empowered to identify and tag that context, specify the expected behaviour of the things involved, and set the privacy policies for sharing their information by being advised of what information they are sharing and with whom. As complete predictability of interactions is very hard to achieve, the user needs to always understand how the interaction can be stopped immediately, like whacking your phone as proposed in [13], and also how this misbehaviour can be prevented in future.

5 Internet of People Blueprint

We have designed a reference architecture supporting our vision of the IoP, based on the PeaaS and Social Devices platforms. The result is a more human-like and less user-dependent IoT ecosystem. The architecture is a natural consequence of the objectives set out in the IoP Manifesto. As illustration, we shall consider a particular case study scenario.

5.1 Scenario: Smart Transportation

David lives in a residential suburb of Paris. Like every day, he is driving his daughter to school before going on to work. His smartphone has learnt where he lives, where he works, and the route he usually takes and at what time. It has reported this information to the city's transportation control systems. David is happy to anonymously contribute such information since it is used for simulations and previsions of potential traffic problems, and to plan improvements in the city.

David has no need to interact with his smartphone because, when he left home this morning, it already knew where he was going. This was confirmed when it detected that the route and speed were as usual. Unfortunately, a traffic accident has just occurred, and David is stuck in a jam. His smartphone has detected an abnormally low speed at a certain point of his route. It asks the smartphones of people around whether they are stuck too. On confirmation, it alerts the transportation control systems of a possible incident. Since every smartphone is reporting the same issue, the control system raises a traffic alert, and notifies the smartphones of people usually take the same route, suggesting an alternative. David's smartphone is now receiving information about the new route to follow, and the estimated time of arrival. It then passes the new route to the car navigator that immediately informs David about the best way out of the jam. The smartphone knows that David is now late and reports to his office, indicating the expected time of arrival.

Most of the people behind David manage to avoid the jam, and David arrives only ten minutes late. He is not too unhappy because he recalls the days when smartphones were only used to talk, read emails, and surf the Web during traffic jams. He is also pleased to have contributed to reducing the impact of the traffic jam for his neighbours.

5.2 Internet of People Middleware

Figure 1 shows a high-level architectural description of the IoP middleware, conceived as a service-oriented system comprising various components: a Device Registry, an Application Manager, an Application Repository, and an Action Repository.

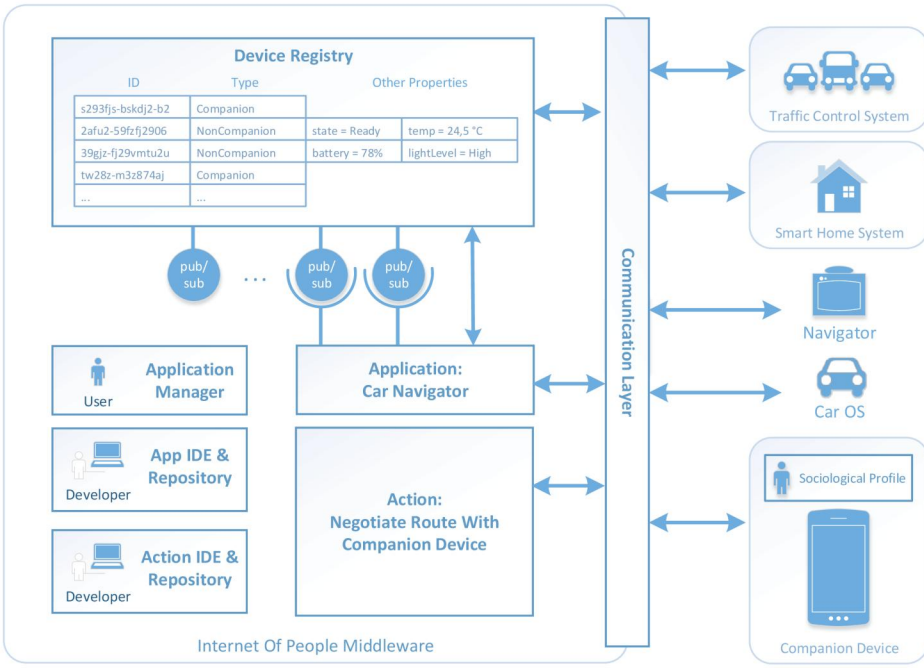


Figure 1: Internet of People Middleware

The central component is the Device Registry. This maintains the information about the different devices managed by the system. These devices are classified into Companion and NonCompanion.

Companion devices are those that maintain contextual and sociological information about their owners, and serve that information through the implementation of the Sociological Profile component inside them. In the illustrative scenario, David's smartphone is registered as a Companion device in the IoP context to allow interactions with other things in that context, such as surrounding people's smartphones or his car navigator. Identifying smartphones as Companion devices in the IoP registry allows them to be considered as representations of people in the IoP context, and, if they are socially capable devices, as conforming to the Manifesto's "be social" principle.

NonCompanion devices are those without a Sociological Profile component. They are "things" which can register their capabilities and any additional information about their state, including any sensing properties and values. The registry incorporates a publish-subscribe API mechanism to allow monitoring the state and property changes of the registered devices. In our scenario, things like car navigators and the traffic control are registered as NonCompanion devices They publish their properties' values in the registry to allow the IoP system to take the appropriate measures according to monitoring rules, such as alerting the smartphones of people close to a traffic jam. This behaviour conforms to the Manifesto's "be proactive" principle.

The Action Repository component manages the middleware's set of actions. Each action defines how the devices should interact with each other and with people. The Application Repository component stores the different applications defined in the IoT context and managed by the IoP middleware. An application consists of a set of instructions defining under what conditions to trigger a set of actions and which devices will be involved. An application's triggering conditions can be defined by monitoring the changes in the state of one or more devices using the registry's publish-subscribe API protocol. In the smart transportation scenario, that mechanism allows an application to be written to monitor the state of the traffic reported by the traffic control system and to trigger the appropriate actions to redirect people in the traffic jam.

The Application Manager component allows people to enable or disable the different applications according to whether they would like to be present and contribute with their sociological profiles.

The application and action specifications may include requests to the different Sociological Profile components of the Companion devices involved, so that the specified behaviour can be adapted to the user's context, personality, or mood.

5.3 Sociological Profile

The Sociological Profile component is a fundamental part of every Companion device. This component maintains the information that has been gathered and inferred about people and their contexts. This information covers people's behaviour, preferences, and also raw contextual data such as current or historical locations, sensing information (e.g. accelerometer status), and processed contextual information such as a proximity graph of surrounding people/devices/things. In the smart transportation scenario, David and other users' Companion devices detected a delay on their route on the basis of their location, movement information, and expected arrival time according to the behavioural pattern data of their sociological profiles.

This component serves information in response to requests via a query-based service. When it receives a request for information about some property stored in the Sociological Profile, it is answered by the Companion device, providing that information as a service response. Companion devices can also notify the registry about significant changes in their Sociological Profile, so that subscribed applications will be notified when an on-change event occurs, allowing them to re-run the queries to get the freshly updated information. In the smart transportation scenario, David's smartphone uses this mechanism to ask nearby people's smartphones for verification of the traffic state.

The component also allows users to define and adjust the sharing schema for the profile. Users should define what information will be available for which applications, devices, or users, and in which context the information should be provided. In the smart transportation scenario, David's smartphone shares his daily route with his workplace, but makes it inaccessible for other systems. Such privacy rules align this approach with the Manifesto's "be predictable" principle.

Finally, the Sociological Profile provides an interface for device owners to personalize the way in which the device behaves under specific circumstances. In the smart transportation scenario, when the traffic control system notifies people's smartphones about the traffic jam on their route, their profiles can mould the resulting alarm action by translating the message into their native language, and shaping the voice to the proper accent, timbre, or mood according to their profile information and context. This conforms with the Manifesto's "be personalized" principle.

Most of the profile's customizable policies and preferences will be supported by a user-friendly wizard. The component will progressively learn the customization during the device's use from the interactions and contexts in which it is involved, and from the owner's decisions about how the device should behave in those interactions. In the smart transportation scenario, David's device recognized a previously learnt pattern about his daily driving routine, which made it possible to detect deviations, and react accordingly. This behaviour also conforms to the Manifesto's "be predictable" principle.

5.4 Building Proactive Interactions

Many IoT and pervasive service systems have their own communication protocols and ways of interconnecting objects. But they typically overlook how the "things" interact with people. To construct more human-like, predictable, proactive, and social interactions between people and "things", the IoP middleware offers a Web-based IDE and tools. These tools allow applications to be constructed to monitor user context, and then, based on the user's context, proactively trigger interactions between the people and "things" involved. Two examples in the smart transportation scenario might be that when the Companion device receives an alert, it can read it aloud for the owner while he is driving, and that the smartphone and the navigator can inform the user while rerouting the way to work so that David can be aware of what is going on. These tools, and the development effort for building new devices and applications has been described in [14] and `http://orchestratorjs.org`.

6 Discussion and Concluding Remarks

The success and growth of IoT is unquestionable. However, people's interaction with this kind of systems is still far from friendly. We have here presented a reference architecture to smooth out how people relate with

IoT systems. By basing people's interactions with the system through their smartphones, our proposal reduces the complexity inherent in the IoT, contributing to evolution towards a true IoP.

The term Internet of People has been used before, but usually to refer to traditional Web systems designed only for humans to use. Here, we have used it in the sense of bringing the Internet of Things closer to people, for them to easily integrate into it and take full advantage of its benefits. Technologies such as Siri for iOS or Sherpa for Android have endowed smartphones with a kind of persona. The next step could be to adjust this persona to fit the user's personality and mood.

This potential use of smartphones as personal information gatherers and managers, and conductors of orchestras of other devices, has also motivated other research initiatives. As well as the aforementioned PeaaS and Social Devices, in [15] smartphones are seen as service providers, with an implementation based on Web Service standards interfacing the user with other smartphone services. This could provide a mechanism for serving user's contextual information, but the management of IoT devices, as provided by the IoP approach, is out of its scope. Paraimpu (`http://www.paraimpu.com`) provides a social tool for people to connect, compose, and share things, services, and devices to create personalized IoT applications. Node-RED (`http://www.nodered.org`) provides a Web-based visual interface for connecting and building compositions of hardware devices, APIs, and online technical and social services. Both works deal great with IoT management but they lack on the predictability and sociological profile features provided in the IoP approach. Apple HomeKit promises integration with a large set of compatible devices, and managing them remotely, but it is also a platform-specific solution.

We have presented here a manifesto to encourage other IoT system developers and researchers to pay attention to the relation between people and IoT systems. As noted above, the architecture proposed here is based on combining the existing Social Devices and PeaaS proposals. Currently, Orchestrator.js and nimBees are consolidated implementations that complement each other in Internet of People middleware. Later on, other systems, like recommendation systems for instance, can be applied to our approach.

The maturity of our previous work allow supporting interactions between people and household devices based on pattern recognition applied to smartphone usage. For example, we can now turn on a coffee machine when a user starts her journey to work. The implementation of scenarios, such as the smart transportation one, enables addressing possible scalability issues.

The major privacy and socially related questions that proposals such as ours raise will set a new horizon for research in the coming years. Our own future work plans focus on taking advantage of the ever-increasing capabilities of smartphones so as to improve how they make inferences about their owners, and to use that information to enhance their owners' roles in IoT systems.

Acknowledgments

This research has been partially funded by the Spanish government (projects TIN2012-34945, and TIN2012-35669), the Academy of Finland (264422) and by Nokia Foundation. It has also been funded by the Government of Extremadura and the European Regional Development Fund (FEDER).

References

[1] L. Atzori, A. Iera, and G. Morabito, “From "smart objects" to "social objects": The next evolutionary step of the internet of things,” *Communications Magazine, IEEE*, vol. 52, no. 1, pp. 97–105, January 2014.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of things (iot): A vision, architectural elements, and future directions,” *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645 – 1660, 2013.

[3] N. Mäkitalo, J. Pääkkö, M. Raatikainen, V. Myllärniemi, T. Aaltonen, T. Leppänen, T. Männistö, and T. Mikkonen, “Social Devices: Collaborative Co-located Interactions in a Mobile Cloud,” in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, 2012.

[4] J. Guillen, J. Miranda, J. Berrocal, J. Garcia-Alonso, J. Murillo, and C. Canal, “People as a service: A mobile-centric model for providing collective sociological profiles,” *Software, IEEE*, vol. 31, no. 2, pp. 48–53, Mar 2014.

[5] CISCO, “Cisco iot connections counter,” 2014, http://newsroom.cisco.com/feature-content?articleId=1208342.

[6] Gartner, “Internet of Things Installed Base Will Grow to 26 Billion Units By 2020,” 2013, http://www.gartner.com/newsroom/id/2636073.

[7] Jean-Louis Gassée, “Internet of Things: The ‘Basket of Remotes’ Problem,” 2014, http://www.mondaynote.com/2014/01/12/internet-of-things-the-basket-of-remotes-problem/.

[8] OASIS, “Devices Profile for Web Services Version 1.1,” 2009, http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html.

[9] M. Kovatsch, “CoAP for the web of things: From tiny resource-constrained devices to the web browser,” in *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*. plus 0.5em minus 0.4emNew York, USA: ACM, 2013, pp. 1495–1504.

[10] P. Gupta and T. Dallas, “Feature selection and activity recognition system using a single triaxial accelerometer,” *IEEE Transactions on Biomedical Engineerin* , vol. 61, no. 6, pp. 1780–1786, Jun. 2014.

[11] Y. Ma, B. Xu, Y. Bai, G. Sun, and R. Zhu, “Daily mood assessment based on mobile phone sensing,” in *Wearable and Implantable Body Sensor Networks (BSN), 2012 Ninth International Conference on*, May 2012, pp. 142–147.

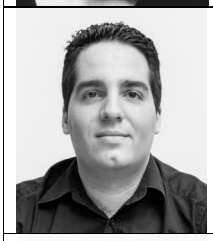
[12] J. Bronsted, K. Hansen, and M. Ingstrup, “Service composition issues in pervasive computing,” *Pervasive Computing, IEEE*, vol. 9, no. 1, pp. 62–70, Jan 2010.

[13] J. Lyon, J. Lau, R. Sarin, J. Park, and M. Veron, “Controlling audio of a device,” Sep. 13 2012, uS Patent App. 13/046,680. [Online]. Available: http://www.google.com/patents/US20120231838

[14] N. Mäkitalo, “Building and Programming Ubiquitous Social Devices,” in *Proceedings of the 12th ACM International Symposium on Mobility Management and Wireless Access*. plus 0.5em minus 0.4emNew York, NY, USA: ACM, 2014, pp. 99–108.

[15] M. Jansen, “About using mobile devices as cloud service providers,” in *CLOSER’12*, 2012, pp. 147–152.

Short bios

	**Javier Miranda** is a software engineer, researcher and CEO of Gloin. His research interests include mobile computing and cloud computing. Mr. Miranda received an MSc degree in computer science from the University of Extremadura, where he is currently a PhD candidate. Contact him at jmiranda@gloin.es.
	**Niko Mäkitalo** is a PhD candidate and researcher at Tampere University of Technology. His primary focus of research has been on mobile devices, cloud software, and pervasive systems. Particularly the focus has been on coordinating IoT devices and enabling humans to interact with these devices in more human way. Contact him at niko.makitalo@tut.fi.
	**Jose Garcia-Alonso** is a co-founder of Gloin. His main research interests include product lines, software architectures and model-driven development. He received his PhD degree in computer science from the University of Extremadura, where he is currently an Assistant Professor. Contact him at jgaralo@unex.es.
	**Javier Berrocal** is a co-founder of Gloin. His main research interests are software architectures, model-driven development, and business process. Mr. Berrocal received his PhD degree in computer science from the University of Extremadura, where he is currently an Assistant Professor. Contact him at jberolm@unex.es.
	**Tommi Mikkonen** is a professor of software systems at the Tampere University of Technology. His research interests include software architectures, distributed systems, and mobile and Web software. Mikkonen received a PhD in computer science from the Tampere University of Technology. Contact him at tommi.mikkonen@tut.fi.
	**Carlos Canal** received his PhD degree in Computer Science from the University of Málaga, Spain where he is currently an Associate Professor. His research interests include component, service and cloud development, in particular formal methods to the specification, composition, and adaptation of interacting components. Contact him at canal@lcc.uma.es.

Juan M. Murillo is a co-founder of Gloin. His research interests include software architectures, mobile computing and cloud computing. Mr. Murillo received his PhD in computer science from the University of Extremadura where he is currently an Associate Professor of Software Engineering. Contact him at juanmamu@unex.es.

II

Publication II

N. Mäkitalo. Building and Programming Ubiquitous Social Devices. In *Proceedings of the 12th ACM International Symposium on Mobility Management and Wireless Access (MobiWac'14)*, pages 99–108. Montreal, QC, Canada, September 21 – 26, 2014, Association for Computing Machinery (ACM).

Building and Programming Ubiquitous Social Devices

Niko Mäkitalo
Tampere University of Technology
Department of Pervasive Computing
Tampere, Finland
niko.makitalo@tut.fi

ABSTRACT

Today's mobile and embedded devices are Internet-connected, and have decent computing power, which creates a possibility for complex, cooperative multi-device platforms. Yet, from user's perspective, it seems that we are not freed to use different technologies seamlessly and without thinking. From developer's perspective, on the other hand, building seamlessly communicating devices requires implementing coordination process separately in an application specific fashion, and yet the devices can only communicate through these specific apps that are not aware of each other.

In this paper we introduce Orchestrator.js middleware, which is a tool for readily building multi-user, and multi-device applications in heterogeneous environment. This new lightweight and agile middleware is designed to support Social Devices concept, which aims to bring people together, and increase social interactions when people are co-located. With a concrete example, we also describe the process of building and implementing Social Devices applications with embedded and mobile devices.

Categories and Subject Descriptors

H.5.3 [**Information Interfaces And Presentation**]: Group and Organization Interfaces – Collaborative computing, Synchronous interaction, Computer-supported cooperative work

Keywords

Social Devices; ubiquitous development tools; proximity-based cooperation; mobile cloud; ubiquitous multimedia

1. INTRODUCTION

On 1991 Mark Weiser envisioned in his article: *The Computer for the 21st Century* [26] how computers will work on the next millennium, and called this paradigm as *Ubiquitous Computing*. The article introduced *pads*, *tabs* and *boards*, which are computers of different sizes and basic building blocks for Ubiquitous Computing. The goal in this paradigm is to make the technology disappear in the environment by hiding several, probably hundreds, of these and other types of devices to our surroundings, and where they then would seamlessly communicate with each other as well as with humans. According to Weiser, only when things disappear to our surroundings we are freed to use them without thinking.

Currently, over two decades later, people have a growing number of digital devices of different types, such as smartphones, tablets, laptops, TVs, cameras, smart watches, etc. These devices have varying resources, like computing power, screen size, or ability to produce and store content. Moreover, the current Internet of Things (IoT) trend is to connect all types of objects from physical world to digital world. Connecting these objects to Internet opens infinitely possibilities for creating services where the resources of these *smart objects* could be utilized. However, we are living in a world of many manufactures, and their competing standards, and hence the objects are not communicating with each other like Weiser envisioned. This has led us to a situation where managing the resources of these devices has become time-consuming task, which requires manual efforts from the user, and takes a lot of user's attention. On the other hand, many of the resources are inefficiently used since people typically are capable of using only one device at a time. Although connecting objects to Internet has now become a trivial task, the real problem lies in the lack of seamless user experience, which has even lead to manifestos like *Manifesto for Experience of Things* [20], and *Liquid Software Manifesto* [23].

Cloud and social media services have emerged to support the collaboration and communication of the people. However, the support for mobile devices and especially for smart objects in these services is typically very limited offering only remote interfaces for utilizing the services rather than exploiting the different resources each device or object has to offer. Furthermore, the cloud services are not offering many ways for automating things on the device-side, and hence several manual efforts are yet required from their users.

In many cases the technology can actually separate people from other people and from their surroundings; It is common to see people gazing their mobiles, possibly communicating remotely with someone rather than face-to-face with people nearby. To turn around the current setting, we have introduced Social Devices concept [14]. This concept aims to turn the attention away from the technology itself, and offer more seamless experiences by facilitating, enriching and increasing interactions in various kind of situations.

Compared to our earlier work [14, 1], this paper contributes the following. $Firstly$, we introduce a completely new *Orchestrator.js (OJS)* middleware that supports Social Devices concept. Unlike the initial approach, Orchestrator.js is implemented as a single cloud service that offers a complete set of tools to implement and prototype Social Devices applications. $Secondly$, although it has been our purpose from the beginning that every device could be a Social Device, the initial approach was not applicable for embedded hardware due to its complexity. In this paper we describe how a So-

MobiWac'14, September 21–26, 2014, Montreal, QC, Canada.

http://dx.doi.org/10.1145/2642668.2642678.

cial Device can be build and programmed with Microsoft's .NET Gadgeteer platform. $Thirdly$, the overall process of building and implementing Social Devices applications have not yet been covered in our previous publications. In this paper we explicitly describe the implementation process for Social Devices applications as a whole. $Finally$, with this paper we also hope to encourage people to try out developing Social Devices applications as with the web-based tools developers don't have to install or set up IDEs. On the other hand, our abstractions may allow even non-developers to try out implementing simple interactions.

The rest of the paper is structured as follows. In Section 2 we prime a running example of coffee breaks with Social Devices. In Section 3 we continue motivating by reflecting related work to our work. In Section 4 we describe Social Devices research area and introduce the new Orchestrator.js middleware. In Section 5 describe how OJS can be used for implementing the coffee break example. In section 6, we evaluate the OJS middleware, and in Section 7 we discuss about other aspects and future work. Finally, in Section 8 we draw conclusions.

2. THE RUNNING EXAMPLE: LETS'S HAVE A NICE CUP OF COFFEE

Imaging a following scenario: *Bob goes to work every day by bus. When Bob arrives to office at 9:00 am, he typically has a nice cup of coffee with his colleagues Jane, Joe, and Lisa before they actually start working. They also like to have coffee after lunch, but sometimes get it for free when they go for a lunch to a restaurant nearby. On the other hand, the colleagues often have meetings with their clients at different times, so it is hard to estimate when people want to have coffee, or how many wants to have coffee after lunch.*

Social Devices offers a better way of having coffee breaks with colleagues: the system can for example ask if people want to have coffee when they are on their way to the office, or invite and inform colleagues when there is a coffee break going on. The system can also trace how much coffee there is left, switch off if no one wants to have more coffee, and remind when it is time to clean and refill the machine. For the sake of brevity, we focus on the following part of the scenario: *Bob is on his way to work, and his phone asks him if he wants to have coffee. When Bob answers yes, the Social Devices coffee machine at the office starts making coffee. When the coffee is ready, the machine invites Bob and his colleagues to have coffee.* Photos A, B and C in Figure 1 represent this scenario. From now on, we use name Coffee Break for this application. After Bob and his friends have gathered together, other interactions (e.g. game, photo sharing or refill coffee machine) would be triggered between them.

3. RELATED WORK AND MOTIVATION

For many years ubiquitous research groups had to build hardware for their specific needs. However, over the past decade several platforms and open source communities have emerged for prototyping and building embedded and low-level ubiquitous systems from pre made building blocks [10]. Platforms like Arduino, littleBits, Lego®Mindstorms, d.tools, and .NET Gadgeteer are only some examples of these platforms. All these platforms seem to have a common goal that is to support amateurs and professionals to quickly build custom hardware for their varying needs. However, the tools and IDEs used for developing for each platform vary from simple text editors and advanced IDEs such as Visual Studio to web–based editors and IDEs. From our own experience, we feel that the native tools of each platform typically work best when implementing the actual functionalities for a device, and thus we want to keep it that way: the OJS middleware introduced in this paper offers tools and supports designing interactions between devices and people, and for defining the contexts when the interactions take place, whereas the actual functionality on device–side yet is done with the native IDEs of each platform.

For this paper we chose the .NET Gadgeteer platform as it offers a very easy way for developers to prototype new concepts. Gadgeteer is a modular platform containing mainboard where other modules of different types can be connected simply by plugging cables into sockets. Programming for the platform is made very simple with Visual Studio IDE. Currently the .NET Gadgeteer project has tens of different types of modules and new ones are constantly being developed by commercial retailers such as GHI Electronics, as well as by open source communities around the project. These modules can be used in example for connecting the device to Internet, storing data to memory card, playing sounds, or controlling relays and servo motors.

Although many of these platforms offer network connectivity, they lack the common communication protocols and programming models so that they could genuinely communicate with each other. The lack of these features makes it hard to create seamlessly communicating devices as the communication protocols and backend services need to be separately implemented for each purpose. Similarly, it requires work to make mobile devices and regular computers to communicate with these devices as well as with each other.

There have been many approaches to form service compositions for pervasive computing before, such as Context Toolkit [19], ICrafter [2], Aura [22], Gaia [18], Obje/Speakeasy [8], and SociableSense [17], just to name a few. (More e.g. in [5]). All these have the same goal, that is to offer more seamlessly cooperating entities, but all of them have their own goals too. The main goal of OJS is to support Social Devices concept, which means that *the middleware is targeted to facilitate and increase interactions between people and devices when they meet face-to-face*. Hence, we are not aiming to automate services in pre-defined locations, such as smart spaces (e.g. iDevices), but rather to coordinate devices wherever they are in the proximity of each other. The Coffee Break application used in this paper is an example of how Social Devices can be used to *bring people together*, using coffee as a motivator [25].

Another feature that separates Social Devices from many other service compositions is that the independently socializing devices start to resemble humans, and get similar characteristics. To support proactive interactions where also devices get treated as human-like actors, we are relying on Action–Oriented Programming Model (AcOP) for pervasive systems [1]. In AcOP the operations and the device coordination are based on proactive actions. The notion of an action is rooted in the DisCo method [12], which is a formal specification method for reactive and distributed systems based on the *joint action* theory [3]. Unlike DisCo actions that can be mapped to terms of logic, the actions in the AcOP do not have a formal meaning in the mathematical sense. In DisCo actions, the participants are typically processes, whereas in the AcOP the action participants are usually, but not always, devices.

Similarly than Context toolkit, AcOP has several abstractions that are delegated to take responsibility of different level tasks. However, the concepts are different from one to another. E.g. AcOP concept of app mostly relates to the concept of $Interpreter$ in Context Toolkit as they both receive changes in contextual data, and translate it so that it can be utilized by other components. Consequently, Context toolkit allows adding context awareness to native apps that also contain the communication logic between entities. In AcOP, on the other hand, the coordination happens with device

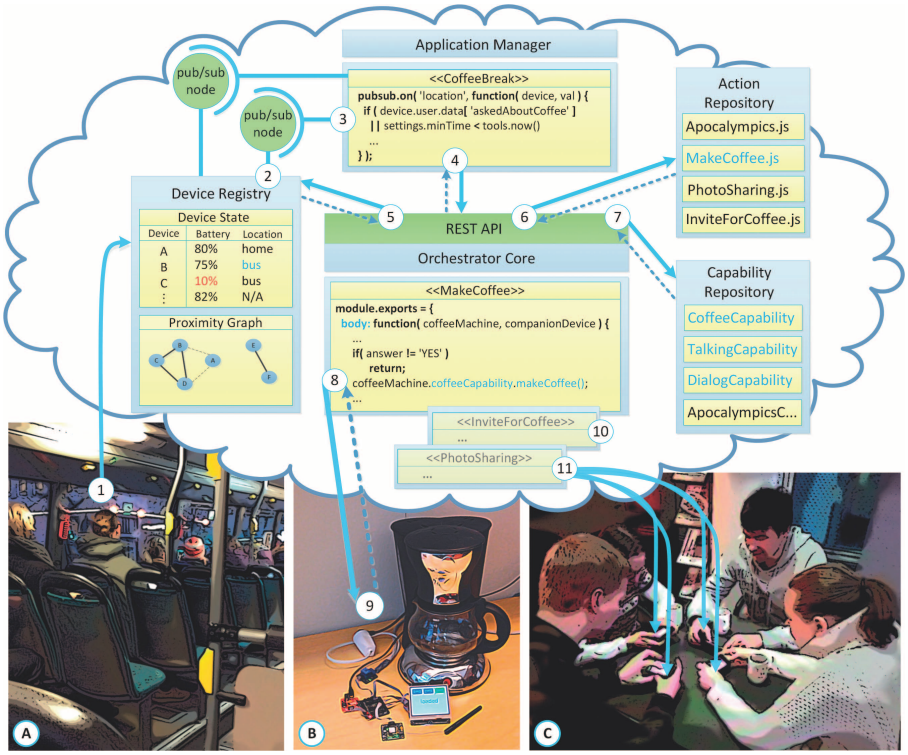


Figure 1: Orchestrator.js middleware and Coffee Break application configuration.

capabilities, and has been separated in to its own unit as this makes the process of defining interactions more concrete.

The execution model of actions behaves like a single guarded loop in Dijkstra's guarded command language (GCL) [7]. In the GCL, the basic building blocks are guarded commands which is a statement list prefixed with boolean expressions. The statement list is eligible for execution only if the boolean expression evaluates to $true$. Similarly, an action can be executed if the precondition is $true$, and the statement list corresponds to the body of an action. In addition to action, AcOP also offers other concepts, for example describing human-like features of the device with capabilities, and interaction contexts with apps.

Finally, as Social Devices is a new type of social–digital environment where companion mobile devices represent their owners and have human-like characteristics, we also find several common touch points to *People as a Service model (PeaaS)* [9], and thus we have already started cooperating with the PeaaS research group.

4. SOCIAL DEVICES

The goal of Social Devices is to increase, facilitate, and enrich social interactions between people in various kinds of co-located and face-to-face situations. The interactions can take place in any location, and can be very different from each other by their nature. The interactions can be, for example, multiplayer games, sharing multimedia in face-to-face situations, or synchronization services, like changing contact information when a group of businessmen meets. Moreover, the interactions can be used for bringing people together (like in CoffeeBreak app), and then for finding some common interests. Our concept website has some demo videos [21].

4.1 Overview of Social Devices

The concept of *Social Devices* and our initial approach of coordinating the devices *(SDP)* was first introduced in [14]. The structure of SDP was based on several components: *State Server*, *Proximity Server*, *Configurator as a Service*, *Controller*, and *Orchestrator*. The components were *Python/Django*–based cloud services, communicating with each other through REST APIs. This architecture offered for us and for our collaborators the possibility to study several areas related to the concept, such as: *how to find a suitable configuration set of devices in multi–device applications* [16], *how these kind of multi–device applications should be modelled* [1], and *how people embrace the concept* [24].

As with many other pervasive service compositions [5] the architecture and the automatic recommendations, however, became very

complex, and thus did not well fit to our prototyping and research purposes. Based on this earlier research (between 2011–2013) we are now able to streamline the architecture, and hence this paper contributes by introducing a completely new *Orchestrator.js* middleware, which covers the main responsibilities of the initial approach. Moreover, this new middleware offers a complete set of tools for building and running Social Devices applications. An instance of *Orchestrator.js* currently runs in Amazon cloud[1], and is now available from GitHub under MIT license.

4.2 Social Devices Applications

Developing applications in context–aware, heterogeneous multi–device environment can be confusing task, and often requires reinventing the wheel for tasks like scheduling, or abstractions of device coordination. For these purposes *Action-Oriented Programming Model (AcOP)* offers abstractions for developers that help them to structure pervasive applications [1]. The main concepts of *AcOP* are: *actions*, *capabilities*, and *triggers*. *Apps* and *observers*, which were not originally key concepts of AcOP, have now an important role in the new Orchestrator.js middleware. (We later revisit these concepts by describing how they should be used in OJS, and how the new concepts improve the AcOP). Social Devices applications are based on AcOP, and hence differ from the native and web applications in several ways:

Firstly, the Social Devices apps are extremely modular, and thus the execution model of Social Devices apps differs from the traditional applications. The Social Devices apps mainly run inside cloud where observer components monitor contextual data, and based on developer defined logic schedule interactions between the users and the devices. Hence, the apps are the top-level components that select when and what interactions will be scheduled and to whom. Social Devices functionality, however, is not entirely tied to apps and the interactions can be triggered by other components, like other interactions for instance. Typically the apps are personal, meaning that the user can start an instance for her and grant permissions to monitor her personal data. However, the developers can also implement apps that monitor data from multiple users, or data from external sources like [15]. Naturally, the users have to enable and grant permissions for using their personal data also with these non-user-specific apps (similarly than Facebook apps).

Secondly, the core part of apps are *actions*, which are small modular units that define the joint behavior of multiple devices, and interactions between people. An app may consist of multiple actions as the running example of this paper shows: For example, when the system asks Bob: "Do you want to have coffee?" is one action, which defines the interaction between Bob (through his mobile phone) and the coffee machine at the office. Another action can then invite people to have coffee when it is ready, and yet another one can remind someone to clean up the machine. On the other hand, an action can be more long lasting, like a game where multiple users take part using their phones.

Thirdly, capabilities are used for describing what a device can do. For example, if device has a *dialog*–capability, it can interact with a user through simple dialog user interface. Or, if the device has *talking*–capability, it can translate text to speech and communicate with the user with voice. Hence, the capabilities define in which roles the device can be in an action and which actions the device can take part to.

Finally, an action can be part of several applications, and an application can consist of any number of actions and devices with different capabilities. Hence, the boundaries between applications are very loose. As an example, Figure 2 presents the execution model of Social Devices applications from one device perspective. Figure 2 contains three different applications, and illustrates how four different actions have been scheduled by these three apps based on several observers.

[1] `http://www.orchestratorjs.org`

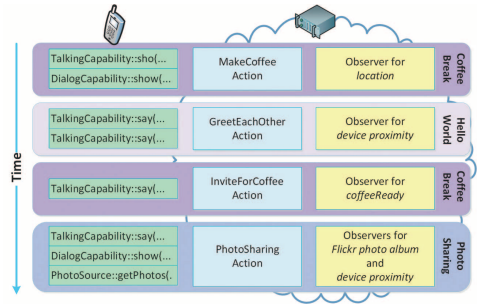


Figure 2: Example of executing three different Social Devices applications with four interaction definitions from one device perspective.

4.3 Orchestrator.js: Middleware for Social Devices

4.3.1 Overview

Orchestrator.js coordination middleware is especially designed to support Social Devices concept as it offers tools for implementing interactions between devices and people, and for observing situations when these interactions should proactively get triggered. Although Social Devices applications differ from traditional mobile apps, the Orchestrator.js can still be used for quickly developing regular heterogeneous multi–device applications as the lightweight server can easily be hosted individually for each purpose. Thus, the developer don't have to pay too much attention to the execution model of Social Devices apps, but can instead focus on a single application, and host it on any regular computer like a living room media center for example. The main task of Orchestrator.js is to maintain persistent connection to devices, store applications, store contextual information of devices, and offer means for triggering the applications. Moreover, Orchestrator.js offers tools for developers to implement, test, and deploy their applications.

4.3.2 Architecture

The Orchestrator.js is based on Node.js, which is a server–side JavaScript platform built on Google's V8 JavaScript Engine. The communication is based on Socket.IO protocol, which is a new, officially non–standardized protocol for relying events between device and server. It utilizes WebSockets and other common communication protocols and abstracts their differences. This technology stack offers highly efficient input/output operations between Orchestrator.js server and its devices.

Figure 1 illustrates how the most relevant components of Orchestrator.js system work in the Coffee Break use case. These components are Orchestrator Core, Application Manager, Device Registry, Action Repository, and Capability Repository. The Orchestrator Core creates new action instances, and makes sure that the actions work properly. It maintains the connections to de-

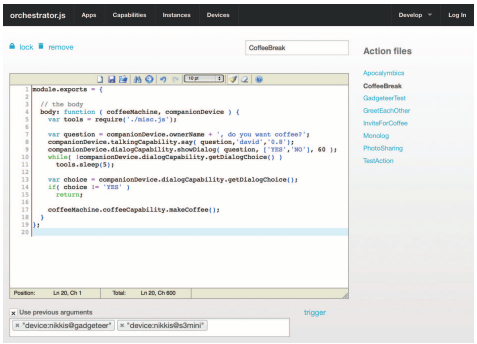


Figure 3: Orchestrator.js web-based Console and IDE

vices and coordinates the devices by handling incoming and outgoing events based on the logic defined in actions.

The Application Manager stores and executes Social Devices applications (apps), allows users to start/stop them, and stores user-related settings that may be required by the app. The apps are simple daemons or child processes that run constantly and monitor changes in different sources with help of observers. The sources can be any type and the observing components can be implemented with various technologies: the observer can, for example, poll a web site with HTTP, use RSS feed technology, or use XMPP's publish–subscribe protocol for receiving notifications and detecting changes. The app may contain one or several of these observers as shown in Figure 4. After the app detects changes, it uses some scheduling algorithm defined by the app developer to determine whether some interaction should take place.

The Device Registry maintains contextual state information about the devices, which is stored to MongoDB database. The non-SQL database makes it convenient to store any kind of sparse metadata of the devices. Moreover, the registry maintains *proximity graph* of the devices, which is formed based on Bluetooth signal strength values (RSSI). This allows triggering applications for a set of devices that are close to each other. Additionally, Device Registry offers notifications through Publish–Subscribe API for any changes in device context data, like the location and the coffeeReady states in the running example. The architecture of Orchestrator.js middleware, however, is not limited for using only Device Registry, but instead also supports using third-party services, like Aware Framework [15] for storing the data of the devices.

The Action Repository contains the action definitions stored by the developers. Basically these definitions are JavaScript files which each contains precondition, error handling, and the description of joint behavior of devices and interactions between users by utilizing device capabilities.

The Capability Repository contains descriptions of each capability. The actual functionality is not stored here; the capability description only defines what methods the capability must implement, and what parameters each method takes. Based on these descriptions, the developers can implement the capabilities like they wish as long as they follow the capability description. In a sense, these descriptions remind Java's interfaces. The actual implementation for the capability, however, can be done with any language.

4.3.3 Tools: Web Console, IDE and APIs

For users, developers, and administrators Orchestrator.js offers a web–based user interface named Web Console. Users can manage their profile and their devices, and to start and stop the applications from Web Console. As the Web Console is completely responsive, it can be used directly from mobile phone's web browser, and also embedded inside mobile apps with WebView UI components. This tool offers means for detecting which devices are online, what is their state, and which applications and actions are running etc.

The Web Console also offers a Web–based IDE for implementing the applications. Developers can define capabilities, and create new observers and actions directly from their browsers, and then test these apps by triggering them manually for a predefined set of devices. The IDE allows the developers to read logs directly from their web browser's log: The developer can write messages to these logs both, on server side (in action code) and on device side (capability code). Writing these messages can simply be done with single method call: ojsLog("this will show up in web browser");.

Unlike implementing observers and actions, the capabilities for the devices are implemented with the native platform specific tools, like with Eclipse (Android) or Visual Studio (.NET Gadgeteer) as these tools typically work the best. However, Orchestrator.js IDE offers a tool defining the capability descriptions, and also helps the developers by generating stubs for their capability implementations which can then be copied to the native IDEs.

The Orchestrator Core allows managing resources remotely through its REST API. The API supports posting apps, and action/capability definitions, triggering actions, changing device states, starting/stopping apps etc. For example, if the developer is not happy with the web–based IDE and prefers to use her favorite code editor, she can easily push the code to the server through the REST API. Additionally, as was noted the Device Registry offers Publish-Subscribe interface, which automatically publishes notifications always when device context data changes. Subscribing these updates is very simple as can be seen from Figure 4. Similarly than writing into Web Console's log, the developers can update the context data of a device with a single method call inside the capability implementation: ojsContextData("coffeeReady", true);.

4.3.4 Orchestrator.js Device Frameworks

Currently, Orchestrator.js offers frameworks for Android, iOS, .NET Gadgeteer and Python supported systems, such as Linux, Mac and Windows computers. Additionally, web-based client can be embedded to web sites, acting as *virtual devices*. The requirements for OJS are easy to meet, and hence expanding the support is easy. Presently, we are implementing support for Arduino.

5. IMPLEMENTING COFFEE BREAK

The following revisits the concepts of the programming model and describes the process of implementing Social Devices applications as a whole. The section covers how to implement and meet the requirements of Coffee Break example with an app and action, and how to implement capabilities for both, Android and Gadgeteer.

5.1 Apps: When an interaction should take place?

Lets start from the beginning, when Bob wakes up and rushes to the morning bus to his office. After Bob has made it to the bus and is now sitting a bit more relaxed on his seat, his phone starts vibrating and asks if Bob wants to have coffee when he arrives to the office. This functionality can be archived as his phone sends sensor data to Device Registry (see phase #1 in Figure 1). In this paper

```
var httprequest = require( 'request' );
var tools       = require( 'tools.js' );
var pubsub      = tools.pubsub();

module.exports = {
  // set by the user while the app starts
  settings: [ coffeeMachineId, location,
              distance, minTime, maxTime ],

  logic: function() {
    pubsub.on( 'location',
      function( cxtDataVal, device ) {
        if ( device.user.data[ 'askedAboutCoffee' ]
          || settings.minTime < tools.now()
          || settings.maxTime > tools.now()
          || settings.distance > tools.distance(
          cxtDataVal, settings.location )
          ) return;

        httprequest( {
          uri: '/api/1/action/MakeCoffee',
          method: 'POST',
          form: { [ settings.coffeeMachineId,
                    deviceIdentity ] }
        } );
    } );

    pubsub.on( 'coffeeReady',
      // observer code omitted for brevity (20 loc)
    } );
  }
};
```

Figure 4: The Coffee Break application code contains observers for detecting location and coffeeReady context data.

the focus is not on activity tracking as there is plenty of research around this topic (e.g. [6, 11, 13, 4]). This kind of tracking can be done, for example, based on GPS coordinates, average speed and direction. In some public transport systems (like here in Tampere, Finland) there is also free WiFi, which can be used to track that the user is in a bus. Moreover, even the context can be misinterpreted, it won't hurt much as Bob can always decline having coffee, or just ignore the notification as it automatically vanishes after a short period. As the Device Registry allows storing contextual data in any format and publishing it through pub/sub API, the location can be in any format. If further filtering is needed, the app developer can use third party services for translating GPS coordinates to address for instance. In the example Bob's phone reports its GPS location to the system. After this the Device Registry publishes this update through pub/sub API (Figure 1, phase #2).

Implementing a new app with Web IDE is easy: just with a couple of clicks the developer has a new template for an app module open in an editor, and there is no need to install anything. The app module contains *settings* dictionary and *logic* method. With *settings* the developer can define what information the app asks from the user when she starts the app. These *settings* are then saved for later use, and available within the app. The *logic* method is used for defining *what interactions should be triggered and when* by implementing simple observers.

Coffee Break app has two observers that get notified when user's *location* changes, and when *coffee is ready* (see Figure 4). As the app is a regular server–side JavaScript program, it is possible to define and call other functions and use libraries. The app can also access Bob's and his devices' contextual information, and get list of devices nearby for example. The logic inside the location-observer (Figure 4, lines 13–18) checks that the user hasn't yet been asked about having coffee, that the current time is in between the values user saved in settings, and that the user is not too far from the saved address. If all the rules set by the user are met, it uses http-library to trigger MakeCoffee action. (Figure 1, phase #4). On the other hand, when later on the Coffee Break app's coffeeReady observer gets notified, the app will trigger InviteForCoffee action for Bob and his colleagues at the office (Figure 1, phase #10). Furthermore, when the colleagues have gathered together in the coffee room, some other app could try to enhance interactions between the people. An example of this kind of app is PhotoSharing app which gets triggered when a person has shared new photos in Flickr, the phone has been standing on table for a while, and there are friends nearby. (Figure 1, phase #11).

5.2 Actions: How the devices behave and interact?

Orchestrator.js allows the developers to use existing interaction definitions in Action Repository, and create new definitions with the integrated Web IDE similarly they can create new apps. With the action definition the developer defines *how the devices should interact with each other and with users*.

When Orchestrator Core receives a trigger for MakeCoffee action it first ensures that Bob's device and other participating devices are free (not allocated for other actions), and then allocates them (Figure 1, phase#5). After this it fetches the newest version of the MakeCoffee action from Action Repository (Figure 1, phase#6), and creates device stubs by fetching the capability descriptions from Capability Repository (Figure 1, phase#7). As the action has precondition defined, the Orchestrator.js Core executes it to ensure that the action works as the developer has designed. If the devices had the required capabilities, the coffee machine was LOADED, and the user was not yet asked about having coffee, OJS starts executing the body. The coordination is based on invoking device capabilities with JavaScript. For example, on lines 6-7 Bob is asked about having coffee with talkingCapability and dialogCapability. If the answer was not 'YES', the action gets closed on line 8. If, on the other hand, Bob didn't decline the coffeeMachine starts making coffee on line 10 with coffeeCapability.

5.3 Capabilities: How a device appears to the user?

5.3.1 *Implementing capabilities for Android*

The capabilities are used for defining *what a device can do*, and users can enable/disable these capabilities. In a way, the capability also defines *how a device does something*; Different devices can implement the capabilities in different ways. For example, an embedded device could implement *dialog*-capability with physical buttons and leds instead of a graphical UI.

With OJS framework for Android the capabilities can be implemented as native code. As Figure 6 shows, developers define the capabilities as Java classes that contain the methods. The methods can take json and basic datatypes as parameters, and also return the same types from methods back to actions. Moreover, the developers can define and start Activities (Android concept for a graphical view) from the capabilities. The main requirement for a capability is that it needs to have initCapability function, which is called when the user enables the capability on the phone. The initCapability function sets the applicationContext field for the capability, which allows the capability to access device resources and start android Activities. The developers are allowed to add their own code in initCapability function for reserving the resources that are required by the methods of the capability. For example, in Figure 6 the lines 8 - 13 initializes the text-to-speech component, which is then used by the say method on line 19.

```
module.exports = {

  body: function ( coffeeMachine, companionDevice ) {
    companionDevice.user.data[ 'askedAboutCoffee' ] = true;
    var m = 'Hi, ' + companionDevice.ownerName + '! ' + coffeeMachine.name + ' asks if you want to have coffee?';
    companionDevice.talkingCapability.say( m );
    var response = companionDevice.dialogCapability.show( m, [ 'YES', 'NO' ], 60 );
    if( response != 'YES' )
      return;
    coffeeMachine.coffeeCapability.makeCoffee();
  },

  precondition: function ( coffeeMachine, companionDevice ) {
    return ( coffeeMachine.coffeeCapability && companionDevice.talkingCapability && companionDevice.dialogCapability
      && coffeeMachine.data[ 'state' ] == 'LOADED' && !companionDevice.user.data [ 'askedAboutCoffee' ] );
  },

  error: function ( error ) {
    var companionDevice = error.getAction().companionDevice;
    companionDevice.talkingCapability.say( 'Oops, ' + error.getDevice().getName() + ' is in trouble!' );
    companionDevice.screenCapability.show( error.getReason() );
  }
}
```

Figure 5: MakeCoffee action defines interaction between the coffee machine and the user (through her companion device).

```
public class TalkingCapability {
  private TextToSpeech tts_;
  private boolean readyForUse_;
  private static Context applicationContext_;

  public void initCapability( Context appCxt ) {
    TalkingCapability.appCxt_ = appCxt;
    tts_ = new TextToSpeech( appCxt,
      new OnInitListener() {
        @Override
        public void onInit(int status) {
          readyForUse_ = true;
    } });
  }

  public void say( String str ) throws Exception {
    if( !readyForUse_ )
      return;
    tts_.speak( str, TextToSpeech.QUEUE_FLUSH );
  }
}
```

Figure 6: TalkingCapability implementation for Android.

Now, when the the Orchestrator.js invokes talkingCapability's say method (e.g. on line 6 in Figure 5), the say method's code on Figure 6 (lines 16 - 20) on Bob's phone gets executed. The dialogCapability definition is not presented in this paper, but is available on GitHub.

5.3.2 *Building Social Devices with Gadgeteer*

Where as Android offers a widely spread platform among mobile devices it is not yet very common in embedded world and don't allow building custom devices. The Microsoft's .NET Gadgeteer platform, on the other hand, allows developers and designers to quickly prototype their own devices, and programming is made very easy with this platform with very expressive C# language. The Orchestrator.js Framework for .NET Gadgeteer was implemented so that the developers could easily build multi-device applications also with embedded devices.

The CoffeeMachine Social Device was assembled from Gadgeteer's WiFi RS21, Relay X1 and Display TE35 modules, and from a regular coffee machine that can be found from any super market. The Relay X1 module controls the power of the coffee machine. The Display TE35 touch screen allows users to control the coffee machine with three touch buttons: ON, OFF, and LOADED. The ON and OFF buttons work just like the normal on/off switch of the coffee machine, and hence the user can use the machine just like she would normally use it without caring about other features. With the LOADED button user sets coffeeMachine state in Device Registry to indicate that it has been filled with water and coffee grounds, but is not yet powered up. This way the system knows when the machine can be switched on remotely (Figure 5, line 15). Ideally, the user would fill up and set the state to LOADED every time after cleaning the old filter and coffee grounds.

Figure 7 shows how the CoffeeMachine example device and its CoffeeCapability has been implemented with Orchestrator.js Framework for .NET Gadgeteer. The total length of the code was 240 lines, but some boilerplate code was omitted for brevity reasons. The example is also kept very simple and only focuses on the main points: other sensors could easily be attached to the device to support more features. For example, with Moisture Module it would be very easy to check if there is water in the coffee machine's tank to automatically set the state to LOADED.

The line 6 shows that the machine state can be ON, OFF, or LOADED, which maps directly to the buttons shown for the user on display module. Lines 12–15, on the other hand, show how simple it is to initialize the connection to Orchestrator.js and start observing for capability method calls. The method call handler (lines 15 – 44) gets called when the device receives capability method call, and the developer can easily check which capability and method was invoked, and then implement the features for her gadget.

Now, when on the Orchestrator.js calls CoffeeMachine's makeCoffee method (line 16), the machine ensures that it was filled up with coffee grounds and water (line 18), and then switches on the relay which powers up the coffee machine (line 22) and changes the state to ON (line 25). Updating the user interface is omitted from the code for brevity reasons. Switching on the machine also starts up a timer which counts down how long it takes until the coffee is ready and updates the user interface accordingly (lines 31–33). On line 27 the coffee machine sets its coffeeReady context data to false to indicate that the coffee is not yet ready. Similarly, when the coffee is ready the device reports this to the server.

```
// using definitions omitted for brevity. ( 20 loc )
namespace CoffeeMachine
{
  public partial class Program
  {
    public enum CoffeeMachineState { ON, OFF, LOADED };
    public CoffeeMachineState coffeeMachineState = CoffeeMachineState.OFF;
    // other class variable definitions omitted for brevity. (30 loc)
    void ProgramStarted() {
      // wireless lan setup omitted for brevity. ( 8 loc )
      // create new instance of Orchestrator.js connection
      ojsClient = new OJS.OrchestratorJSClient(deviceIdentity);

      // register handler for listening method calls from OJS
      ojsClient.MethodCallReceived += new OJS.OrchestratorJSClient.OnMethodCallRecievedHandler((e) => {
        if ( e.capabilityName == "CoffeeCapability" && e.methodCall == "makeCoffee" ) {

          if( coffeeMachineState != CoffeeMachineState.LOADED )
            return;

          // turns the coffee machine on
          relay_X1.TurnOn();

          // set state of the device, and report OJS that coffee is not ready
          coffeeMachineState = CoffeeMachineState.ON;
          ojsClient.contextData( (new Hashtable()).Add('coffeeReady', false) );
          ojsClient.contextData( (new Hashtable()).Add('state', 'ON') );

          // UI buttons modification omitted for brevity ( 3 loc )
          // Timer initialization until coffee ready. TickHandler omitted for brevity. ( 10 loc )
          coffeeReadyTimer = new GT.Timer( 1000 );
          coffeeReadyTimer.Tick += new GT.Timer.TickEventHandler( coffeeReadyTimer_Tick );
          coffeeReadyTimer.Start();

          ojsClient.ojsLog( "I am now making coffee." );
          ojsClient.sendResponse( true );
          return;
        } else if ( e.capabilityName == "CoffeeCapability" && e.methodCall == "turnOff" ) {
          // Stopping timer, updating UI and state omitted for brevity ( 10 loc ).
          return;
        } else {
          ojsClient.ojsError( "No such method" );
        }
      });
    }
    // SetupWindow method for initializing UI in Display_TE35 omitted for brevity ( 100 loc )
  }
}
```

Figure 7: Example code of attaching .NET Gadgeteer enabled coffee machine to Orchestrator.js.

6. EVALUATION

Brønsted et al. have evaluated several service compositions for pervasive computing, and set four main goals the systems should meet [5]. In the following we evaluate how Orchestrator.js meets these goals.

6.1 Context Awareness

According to *the first goal*, the system should be able to respond context changes by providing information or executing command. OJS offers several ways to respond context changes. $Firstly$, OJS is especially designed to compose the services in an ad hoc manner as the interactions (actions) are very momentary by their nature. This supports the first goal as short-lived compositions can be re-specified at runtime based on the new context. $Secondly$, OJS offers means for reporting and storing heterogeneous and sparse context data, and also means for receiving notifications whenever the context data changes. Hence, the developer can define her own logic to react these changes, and decide how to inform the user. Compared to our earlier approach in SDP, the mechanism to select the devices is much more lightweight, but also gives free hands for the developers to form the compositions. Moreover, the developers can also use other services for recommending the devices to the compositions.

6.2 Managing Contingencies

The second goal is to manage contingencies. OJS actions are JavaScript modules that contain the interaction definition ($body$), but also may also contain three optional methods: $precondition$, $warning$, and $error$. If the error method has been defined it gets called when an exception is thrown in action body, or in device-side code as OJS automatically relays exceptions from device to the server. Inside this method the developer can trace what happened, and then react accordingly, for example by informing the user that a device was disconnected and for finding a new device, after which the execution of body continues from where it stopped. If error method has not been defined, the default method kills the action process. The precondition is used for making sure that the action can take place, which helps to prevent the contingencies already beforehand. The warning method works similarly than error method, but it is used by the device-side developer to manually inform the action about some minor error cases, and also the default handler does not kill the action. Furthermore, although the actions are typically short-lived, we have been planning to implement *an*

interruption handler method that would allow new devices to ask if they can join into a running action instances (e.g. to games).

6.3 Leveraging Device Heterogeneity

According to *the third goal*, the system should distribute responsibilities in the composition on the basis of capabilities of different types of devices. As has been discussed above, one of the main concepts of AcOP are capabilities which are used for describing what a device can do. This makes it straightforward for the app developer to pick appropriate devices to the interactions and set their roles. As the contextual information stored in registry as well as the information in social media and other third party services can be accessed within apps and interactions, it is possible to define algorithms to always select the device that best fits for each purpose. Moreover, the constantly extending support for mobile and embedded devices allows harnessing many of the everyday physical objects to OJS, which also supports the goal as it helps to built custom devices to some specific purposes.

6.4 Empowering Users

The fourth goal demands that a pervasive app should change purpose and functionality over their life time, and also that the automatic solutions for this do not always work, so the middleware should offer means for the user to recompose the service. The architecture of OJS and the concepts of AcOP model allow users to control interactions in several ways. The user can now easily control what *applications* she wants to run, what *interactions* her phone can take part to, in addition to enabling/disabling device *capabilities*. If the user never wants her phone to talk, she can easily disable this capability. If the user doesn't like some specific action she can easily disable it instead of disabling some capability. And if the user wants to prevent interactions to take place in some specific situations, it is easy to stop the application. The user can also *change the settings of some specific app* by restarting the app. Finally, we are also implementing an emergency break for interactions: users can slightly snap their phones in order stop an action if it took place in wrong context. This incident gets then reported to server, and for the app developer to improve the triggering process in the future.

7. DISCUSSION AND FUTURE WORK

Currently, our ongoing CoSMo[2] research project (2013-2015) conducts studies in real context to gain empirical understanding of how users experience Social Devices applications [24], and also develops the Orchestrator.js further. Moreover, to improve the overall privacy in OJS and Social Devices, we have started cooperation with a research group working on with *People as a Service (PeaaS)* model [9]. PeaaS offers user profiles as anonymous services running on users devices, and hence no information leaves from user's device without user's decision. What is more, we believe that richer user profiles of PeaaS can help Social Devices to offer better interactions between users. In the following we discuss about some other aspects of OJS and Social Devices and outline future work.

Originally, the main concepts of *AcOP* were: *actions*, *capabilities*, and *triggers* [1]. Whereas the actions and the capabilities are yet important for device coordination in the new Orchestrator.js system, applying AcOP for the rest of the system has changed a lot. In SDP the actions were started with a complex distributed scheduling mechanism that was based on *triggers*, and evaluating actions' *preconditions*. However, this generic mechanism soon became very complex, and yet required implementing observer components that were very much applications specific. Now, the new concept of *application* in AcOP allows developers to have much more influence on the way the actions get triggered. It also offers an easy-to-understand tool for wrapping observers and scheduling algorithms in one place. The end results is that the execution model is now much more lightweight, flexible and streamlined. As the apps are separate processes that users can start/stop, it allows users to have more control over their devices. Finally, the apps that know device context and can trigger actions based on the context, are also aligned with Weiser's claim that *"If a computer knows merely what room it is in, it can adapt its behavior without requiring even a hint of artificial intelligence"* [26].

[2] http://www.cs.tut.fi/ihte/projects/CoSMo/

	SDP			Orchestrator.js		
Δt_i	wlan1	wlan2	3G	wlan1	wlan2	3G
$\overline{\Delta t}$	409.3	450.6	973.9	75.2	84.9	457.7
σ	22.88	26.85	38.29	6.46	18.39	53.70

Table 1: Communication latency in device coordination (ms).

The tools of Orchestrator.js allow implementing and testing all the interaction specific components directly within web browser. We believe that these tools make it very easy to start developing for Social Devices as developers don't have to install any additional tools. However, for the device-side development the developers can still use native and their favorite tools that they would use anyway as these tools typically work the best. On the other hand, if developer wants to install OJS, she can download it from GitHub, and simply run `npm install` command in the extract directory (assuming that Node.js and MongoDB are already installed).

To get input of Social Devices concept from outside developers we hired a student team, and gave them free hands to implement what they wanted. The team chose to implement Apocalympics zombie game [21], which was targeted to *"familiar strangers"* (e.g. people traveling everyday in the same bus, but only know each other by their faces). It was easy for us to explain the system for the team, and after a ten minute presentation they were already able to start implementing the game, and it only took two weeks to implement the game. The students were happy with the AcOP concepts and device coordination. However, the triggering during that time was based on the old SDP system, and hence was not implemented. Now, we have built couple of triggering strategies with the concept of app. As this experiment only gives initial feedback from outsiders, we are arranging code camps with students in near future to learn more about the programmability. By publishing the Orchestrator.js as open source, we aim to grow a community around Social Devices, and hope to get new applications as well as other input.

In order to get some perspective of the communication latency and connection stability of both, Orchestrator.js and SDP, we performed a set of measurements. As the table 1 shows the lag with Orchestrator.js is significantly less than with the SDP when using WLAN connection, and half smaller when using 3G. Also the standard deviation is a bit smaller when using WLAN. However, with 3G the standard deviation is relatively big with both approaches as the 3G can be less stable than WLAN, and may become slow at times. This asymptotic difference in the latency becomes even more substantial if we consider that the reaction time of a human is typically around 150 to 200 ms for auditory and visual stimulus, and hence, it could be assumed that soon after this time has passed people start wondering why the system is working slowly. Furthermore, we have already started experimenting with Bluetooth Low Energy based coordination, where one of the participating devices

commands other devices with Bluetooth, or through OJS if some device only supports Internet connectivity.

8. CONCLUSIONS

The amount of mobile and embedded devices increases at accelerating speed; The world is changing and no single device dominates user's digital life anymore. However, it is not seamless to use and manage multiple devices, and managing requires a lot of time and people's attention. In this paper we presented a new Orchestrator.js middleware and described how it can be used for building proactive applications in heterogeneous, multi-user and multi-device environment. The lightweight and agile middleware offers a complete set of tools for building more seamless interactions and experiences between users and devices. Moreover, with the new tools the programming and testing can be done directly within a web browser. The middleware is freely available and it is easy to start using it. However, we will continue developing it further, and plan to test it with students. Especially, we hope to get feedback from outside developers and that it would eventually lead to a developer community around Social Devices.

Furthermore, with our Social Devices concept we are studying what kind of interactions is appropriate to be proactively triggered for people, and how people take this type of new socio-digital systems. We are continuously implementing studies with real users and in real-life contexts. We also hope that the new middleware encourages people to start implementing Social Devices applications and build new innovative ideas and social applications.

9. ACKNOWLEDGEMENTS

The research was funded by Academy of Finland (264422) and by Nokia Foundation.

10. REFERENCES

[1] T. Aaltonen, V. Myllärniemi, M. Raatikainen, N. Mäkitalo, and J. Pääkkö. An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud. In *Proceedings of APSEC'2013*, 2013.

[2] G. Abowd, B. Brumitt, and S. Shafer. *Ubicomp 2001: Ubiquitous Computing: International Conference Atlanta, Georgia, USA, September 30 - October 2, 2001 Proceedings*. Lecture Notes in Computer Science. Springer, 2001.

[3] R.-J. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *TOPLAS, ACM*, 10(4):513–554, 1988.

[4] J. Bojja, M. Kirkko-Jaakkola, J. Collin, and J. Takala. Indoor localization methods using dead reckoning and 3d map matching. *Journal of Signal Processing Systems*, 2013.

[5] J. Bronsted, K. Hansen, and M. Ingstrup. Service composition issues in pervasive computing. *Pervasive Computing, IEEE*, 9(1):62–70, Jan 2010.

[6] D. Choujaa and N. Dulay. Tracme: Temporal activity recognition using mobile phone data. In *Proceedings of EUC'08*, 2008.

[7] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[8] W. Edwards, M. Newman, J. Seciivy, and T. Smith. Bringing network effects to pervasive spaces. *Pervasive Computing, IEEE*, 4(3):15–17, July 2005.

[9] J. Guillen, J. Miranda, J. Berrocal, J. Garcia-Alonso, J. Murillo, and C. Canal. People as a service: A mobile-centric model for providing collective sociological profiles. *Software, IEEE*, 31(2):48–53, Mar 2014.

[10] S. Hodges, N. Villar, J. Scott, and A. Schmidt. A New Era for Ubicomp Development. *Pervasive Computing, IEEE*, 11(1):5–9, 2012.

[11] L. Hong, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The jigsaw continuous sensing engine for mobile phone applications. In *Proceedings of SenSys '10'*, pages 71–84, New York, NY, USA, 2010. ACM.

[12] H.-J. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of ICSE'90*, 1990.

[13] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song. Comon: Cooperative ambience monitoring platform with continuity and benefit awareness. In *Proceedings of MobiSys '12'*, New York, NY, USA, 2012.

[14] N. Mäkitalo, J. Pääkkö, M. Raatikainen, V. Myllärniemi, T. Aaltonen, T. Leppänen, T. Männistö, and T. Mikkonen. Social Devices: Collaborative Co-located Interactions in a Mobile Cloud. In *Proceedings of MUM '12'*, 2012.

[15] U. of Oulu. Aware Framework. `http://www.awareframework.com`. [Online; accessed 5-March-2014].

[16] J. Pääkkö, M. Raatikainen, V. Myllärniemi, and T. Männistö. Applying Recommendation Systems for Composing Dynamic Services for Mobile Devices. In *Proceedings of APSEC'2012*, 2012.

[17] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. Sociablesense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of MobiCom '11*, New York, NY, USA, 2011.

[18] M. Roman, B. Ziebart, and R. Campbell. Dynamic application composition: customizing the behavior of an active space. In *Proceedings of PerCom '03*, 2003.

[19] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI*, CHI '99, pages 434–441, New York, NY, USA, 1999.

[20] Seamless.li. Manifesto for the Experience of Things. `http://seamlessli.github.io/mfteot/`. [Online; accessed 5-March-2014].

[21] Social Devices. `http://social.cs.tut.fi`. [Online; accessed 25-May-2014].

[22] J. Sousa and D. Garlan. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Software Architecture*, volume 97 of *The International Federation for Information Processing*, pages 29–43. Springer US, 2002.

[23] A. Taivalsaari, T. Mikkonen, and K. Systä. Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In *Proceedings of COMPSAC'2014*, 2014.

[24] K. Väänänen-Vainio-Mattila, T. Olsson, J. Palviainen, and P. Jarusriboonchai. Social devices as a new type of social system: Enjoyable or embarrassing experiences? In *Workshop on Experiencing Interactivity in Public Spaces in conjunction with CHI '13*, 2013.

[25] B. Waber, O. D. Olguin, T. Kim, and A. Pentland. Productivity Through Coffee Breaks: Changing Social Networks by Changing Break Structure. In *Proceedings of SSNC'10*, 2010.

[26] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, January 1991.

III

Publication III

N. Mäkitalo, and T. Mikkonen. At the Edge of the Cloud: Improving the Coordination of Proactive Social Devices. In *Proceedings of the 23rd International Conference on Information Systems Development (ISD'14)*, Varaždin, Croatia, September 2 – 4, 2014, AIS Electronic Library (AISeL).

At the Edge of the Cloud: Improving the Coordination of Proactive Social Devices

Niko Mäkitalo *niko.makitalo@tut.fi*
Tampere University of Technology / Department of Pervasive Computing
Tampere, Finland

Tommi Mikkonen *tommi.mikkonen@tut.fi*
Tampere University of Technology / Department of Pervasive Computing
Tampere, Finland

Abstract

Today's Internet-connected devices, such as tablets and mobile phones, have excellent computing power, which creates a possibility for complex, cooperative multi-device platforms. However, coordinating these devices typically requires implementing the coordination process separately in an application specific fashion, which takes focus away from the actual application development. For this purpose we have introduced Social Devices middleware, which allows developers to easily coordinate proactive interactions on a heterogeneous set of devices. Since the proactivity sets its own elements to the coordination, in this paper we introduce our research for coordinating Social Devices. Moreover, as cloud-based solutions typically assume established and fast Internet-connectivity, we also describe how we have complemented the coordination paradigm with Personal Area Network (PAN) based coordination. Social Devices applications can now adapt and choose between cloud and Bluetooth Low Energy based coordination as the JavaScript-based coordination logic can be executed on both, device and server side.
Keywords: Social Devices, Mobile Cloud, Proactive Interactions, JavaScript as a Coordination Language, iBeacon-based coordination

1. Introduction

In recent years, smart devices have become increasingly capable and connected. They are used for everyday purposes: for entertainment, for socializing with friends, and for sharing life events. Continuous connectivity enables the devices to utilize cloud services and perform tasks at the background. Additionally, new sensors are emerging and these devices can be used for tracking user activities and context. However, the cloud services are yet typically utilized by the user using the device, not by the device itself. Thus, cloud or social media services do not support seamless cooperation and interoperability of the devices but rather collaboration of the people. Vendors like Apple or Samsung have created their own standards for sharing resources among devices, like streaming music and videos for instance. These solutions, however, are usually initiated by users and typically require manual efforts to coordinate the devices and their resources. Moreover, these solutions are vendor specific and may eventually lead to vendor locks.

To support cooperation and interoperability in a heterogeneous set of devices we have introduced concept of *Social Devices* and its initial implementation named *Social Devices Platform (SDP)* [11]. The system infrastructure is *mobile cloud* based, abstracting the physical differences of the devices. The concept of mobile cloud here refers to a system where different types of devices are connected with some technology, and hence communicate with each other either directly or through a communication service. Social Devices support the heterogeneity and different resources of each device by regarding them as capabilities; The capabilities describe

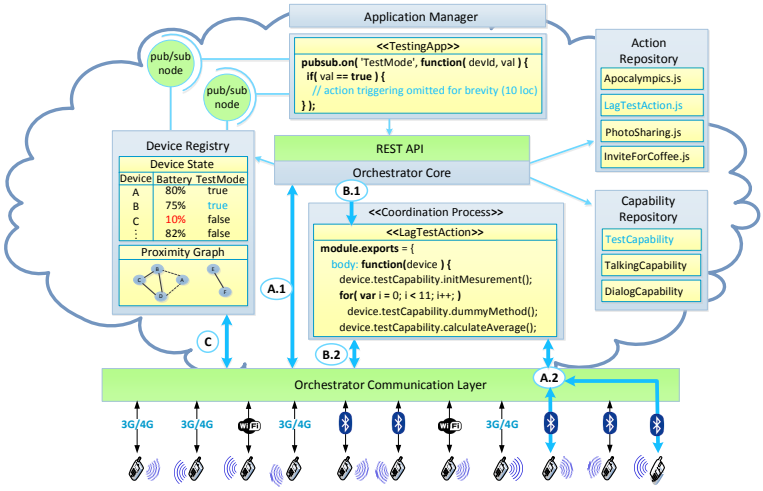


Fig. 1. Social Devices middleware running LagTest action.

what a device can do: the device may, for instance, have *TalkingCapability* installed enabling it to translate text to speech. Interactions between devices and people in Social Devices are described with a concept of *action*. An action contains the coordination logic, and hence defines how the devices interact with each other as well as with people. The actions are then proactively triggered by Social Devices applications, based on changes in devices' context. The current Social Devices middleware has been depicted in Fig. 1.

Initially, Social Devices concept was implemented as a cloud service where the communication between the service and the devices was based on Comet-technology (HTTP long-polling), and the coordination language was Python. While cloud-based orchestrating offered a good starting point for the coordination of the devices, our goal from the beginning was to move towards more flexible system architecture and coordination paradigm where also devices within each others proximity could directly coordinate each others by utilizing various communication technologies, such as Personal Area Networks (PAN). In this paper we report our research of coordinating Social Devices, and describe how we ended up using Socket.IO and Bluetooth Low Energy (BLE) as communication technology, and JavaScript as a coordination language.

The rest of the paper is structured as follows. We start with motivating and presenting some related work in Section 2. Then, in Section 3, we describe the device coordination inside Social Devices ad-hoc mobile clouds, and evaluate the different technologies we have used. In Section 4 we present some future work, and finally, in Section 5 we draw some conclusions.

2. Motivation and Related Work

Currently coordinating devices typically requires implementing the coordination process separately in an application specific fashion. Due to this, the applications running on separate devices are not aware of each other, which make it hard to implement seamlessly cooperating systems. The current situation is unsustainable and the lack of seamless user experience has lead to manifestos like *Manifesto for Experience of Things* [9] and *Liquid Software Manifesto* [14]. The approaches for coordinating multiple devices have mainly been focusing on information presentations (e.g. [6, 8, 10]), or for multimedia resource synchronization (e.g. [13]). How-

ever, our work is different, since with Social Devices we are not aiming to offer only automated services or new kinds of interfaces. For example, in [6], we find similarities in the approach for coordinating the devices, but the aim is different. As [3, 6, 10] focus on generating user interfaces and coordinating them on the devices, the system philosophy is more user-centric than ours. We, in contrast, aim to make devices interact and socialize independently, and make the operations visible for the users. When the majority of approaches focus on coordinating the devices in predefined locations, such as smart spaces or homes, our focus is in coordinating the devices wherever they are in the proximity of each other in any location. Several approaches have also been proposed for the modelling and specification of collective actions (e.g. [2]) and for coordinating computational resources (e.g. [1, 4, 5]). We are revitalizing the idea by applying it to mobile clouds, where actors correspond to individual devices forming the cloud, and whereas a centralized entity is responsible for coordinating the execution of mobile devices. In previous research, the closest relative to our PAN-based coordination approach is constituted by coordination languages for mobile agents (e.g. [12]). However, the PAN-based coordination works differently, since we are treating complete mobile devices as agents.

3. Elements of Coordinating Social Devices

The initial approach of coordinating Social Devices was implemented as a cloud service where the communication was based on Comet-technology. Basically, with Comet-based implementation the client maintained HTTP/TCP connection to the server until the server responded with a remote method call. After receiving the HTTP response the client executed the method call and connected again by sending method call response. The coordination language was Python as the coordination service was Django/Python based and the script language worked well in this centralized approach. In the following we describe why we chose to use JavaScript as a coordination language, and how Socket.IO and Bluetooth Low Energy measurably improved coordinating proactive interactions.

3.1. Minimizing Lag and Communication Latency with PAN-based Coordination

The Social Devices concept is meant to support all types of applications that can be proactively triggered in various situations. Consequently, there are also differences how well different interactions tolerate lag. Many of the Social Devices actions are not too critical about the latency or lag, as they are meant to happen in background mainly offering users support in their daily activities by automating things and informing what is currently going on. On the other hand, many actions are much more critical as they require real-time communication and fast interacting with other entities. A self-evident example of these requirements are games where multiple Social Devices take part and need to be coordinated according to the behaviour of other devices. To support faster coordination and situations where Internet connection cannot be utilized, we implemented the coordination process with Bluetooth 4 sockets (RFCOMM) for Android. In this paradigm one of the devices that participates to the interaction is selected to take the role of the coordinator (Fig. 1, phase #A.1), and hence it commands other devices as well as itself (Fig. 1, phase #A.2).

Social Devices are coordinated by invoking their capability methods by a coordinating entity. Basically this means that before a device can be commanded to start next process or update a running process, the coordinating entity needs to receive response from some other device. In a way this requires the device coordination to be synchronous, although the processes running on the devices can be asynchronous. As with any distributed systems, the communication latency between system entities becomes a relevant thing to consider while defining the interactions for Social Devices. The latency in communication affects heavily on the lag that a user typically experiences. In Social Devices the total lag consist of the following:

$$L_{total} = \Delta r_{self} + p + \left(\sum_{i=0}^{n} (g + \Delta c_i + P(x) + \Delta r_i + p) \right) + g + \Delta c_{self} \quad (1)$$

In the equation Δr_{self} and Δr_i reflect the latency of relaying the return value from device to the coordinator, p is the time spent in parsing the return value and passing it to the action body on the coordinator, g is the time spent in generating a method call, Δc_{self} and Δc_i reflect the latency in communicating the method calls to the receivers, and n is the number of capability calls invoked on the other devices before invoking the next method call on the measuring device. $P(x)$ reflects time spent on executing the capability method on a device, and thus, always depends on the function and the implementation of the method call.

To compare communication latency with Comet and Bluetooth (BT) 4 based coordination (and later on with Socket.IO and Bluetooth Low Energy) we minimized the method processing time, $P(x)$, and implemented a *TestCapability* containing a *dummyMethod* that only saved a time delta between two method calls when it was called by the coordinating entity. In *LagTestAction* (see Fig. 1) the dummyMethod was invoked eleven times in a row on one device, resulting to ten time delta values, which were then used to calculate the average time delta to reflect how long it takes to coordinate a device. No other method calls were invoked in between the dummyMethod calls. Consequently, the average lag in these measurements mainly consists of the communication latency, and also invoking, parsing and generating the method calls and their responses as described in equation (2).

$$\overline{L} = \frac{1}{n} \sum_{i=0}^{n} (\Delta r_i + r + g + \Delta c_i), n = 10 \quad (2)$$

The results in Table 1 show that the latency in communication clearly affects to the lag, and that the difference between Comet and BT 4 based coordination paradigms is prominent. On average, the latency is 10 to 20 times longer with Comet-based coordination, depending on the used internet connection. Moreover, this asymptotic difference in the lag becomes even more substantial if we consider that the reaction time of a human is typically around 150 to 200 ms for auditory and visual stimulus, and hence, it could be assumed that soon after this time has passed people start wondering why the system is not working properly.

The measurements results also reveal that the used Internet connection in Comet-based coordination has a strong influence on the latency in coordination. The average lag with 3G connection is almost one second, and twice as slow as with wlan connection. Basically this means that Comet-based coordination over 3G connection is too slow in cases where people participate to the interaction or otherwise intensively follow the interaction of the devices. On the other hand, with a decent wlan connection (or very fast 3G or 4G) the 0.45 second lag can still be tolerable in cases which don't require or offer intensive user input or output.

Compared to BT 4 based coordination there also seems to be more variation in the communication latency in Comet-based coordination, as even a fast Internet connection can become slow at times. Whereas the standard deviation in BT 4 based results is about 2 ms, with Comet-based coordination it is between 23 to 70 ms. This kind of fluctuation in coordination speed may confuse and frustrate users if they cannot be sure if the action execution has ended.

3.2. Improving Cloud-based Coordination with Socket.IO

Although the measurements clearly show that the device coordination with BT 4 sockets is much faster, not all the devices yet support Bluetooth, and thus we decided to try improving the cloud-based coordination with Node.js and Socket.IO technologies. Node.js is a server-side JavaScript platform built on Google's V8 JavaScript Engine. Socket.IO, on the other hand, is a new, officially non-standardized protocol for relying events between client and server, typically

Table 1. Communication latency in device coordination in milliseconds.

	Comet (HTTP)			Socket.IO			BT 4	BLE
Δt_i	wlan1	wlan2	3G	wlan1	wlan2	3G	Android	iOS
1	402	471	976	77	83	429	55	59
2	418	421	953	91	83	421	57	71
3	371	493	972	73	84	448	57	43
4	400	406	973	72	73	530	58	67
5	453	476	1078	74	77	579	60	63
6	434	469	966	75	79	420	57	62
7	395	433	958	70	74	422	61	46
8	393	446	1153	71	136	458	56	48
9	412	450	970	80	77	432	57	55
10	415	441	940	69	83	438	56	70
$\overline{\Delta t}$	409.3	450.6	993.9	75.2	84.9	457.7	57.4	58.4
σ	22.88	26.85	67.36	6.46	18.39	53.70	1.84	10.06

utilizing WebSockets as a communication protocol (Phases #B.1 and #B.2 in Fig. 1 represent the current cloud-based coordination). Both of these technologies are especially designed to support fast input/output operations, and hence the new coordination layer also offers an efficient way for the devices to update their contextual information to the Device Registry (Fig. 1, phase #C) and further notify applications through publish/subscribe interface.

The Table 1 shows that the coordination speed in cloud-based coordination was improved substantially, and is now 2-6 times faster, and almost as fast as with BT 4 sockets while using good quality wlan connection. A notable point is also that Socket.IO-based coordination with a wlan connection is faster than human reaction time, which makes it possible to utilize it in applications that require intensive interaction. Based on our experiences, and supported by the measurements standard deviation, the Socket.IO-connection seems to be more stable and only rarely drops compared to the Comet-technology. Also, as Fig. 2 shows, the interaction initialization now takes less time with the new implementation of the communication layer, mainly because of the faster communication. This supports the proactive nature of Social Devices applications, as some interactions are very critical about this. For instance, when people meet or pass by each others, the need for interaction between their devices may be over within seconds.

3.3. JavaScript as a Coordination Language

In the original Social Devices system the interactions were defined with Python programming language as this platform was Django-based cloud service. However, different mobile platforms, like Android and iOS for instance, natively use different programming languages, and hence cannot directly execute Python. As a solution, automated translation from Python to other programming languages could be applied, but most likely these solutions would be very error prone. Moreover, the actual deployment of the code would still be an issue. In the first experiment of using BT 4 sockets for PAN-based coordination, we used class loader to dynamically load Java byte code from Android application package files (.apk) as we deal with deploying new device capabilities to Android in our current implementation. However, this solution only worked with Android, and the Social Devices concept is designed to support all types of devices.

In Worldwide Developers Conference (WWDC) 2013 Apple introduced its officially sup-

ported JavaScriptCore.framework for executing JavaScript code on iOS 7 and OS X. This allows creating virtual machines or contexts where JavaScript code can be executed, and also allows invoking native Objective-C methods from the JavaScript code. Android has similar support for running Google's V8 engine on Android devices, although this is not currently part of the official SDK and needs to be separately compiled to the application. As the new communication layer was implemented with Node.js, and JavaScript support on the device-side also seems to be emerging, it was a natural choice for the new coordination language. The jump from Python to JavaScript was easy as they both are dynamically typed languages that can be used for scripting. The end result is that we can now run exactly the same interaction definitions on both, server and device sides. However, generating the device communication stub is done differently. Whereas with the cloud-based coordination the device stub utilizes Socket.IO sockets for sending and receiving events to the clients, with the PAN-based solution the device stub invokes a native Objective-C method on iOS that can communicate directly with the devices nearby. JavaScript seems to fit extremely well for this type of heterogeneous multi-device coordination purposes as the developers can implement the actions that directly run with both approaches. What is more, the support for JavaScript gets better all the time, which makes it possible to implement the PAN-based coordination on many other platforms as well.

3.4. iBeacons and Bluetooth Low Energy

The first implementation of PAN-based coordination that was based on Bluetooth 4 sockets had two major problems. *Firstly,* it required pairing of the participating devices. Fortunately, the paring only needs to be done once between each device, and thus would not be that big concern with user's own devices. However, the idea of Social Devices is also to support proactive interactions with friends' devices, as well as with non-personal and public devices, and thus paring with these devices would have to be conducted before the device can be utilized for the first time. The extra work for the user would have been against principles of Social Devices as one of the main ideas is to reduce the manual tasks that currently requires users' attention. Additionally, based on our measurements even though the devices were already paired, it took approximately 3.6 seconds to discover and establish connection between two devices. What is more, receiving the initialization command and retrieving participant device information from Device Registry it took about 6.3 seconds to start running the interaction with the BT 4 socket based approach. With cloud-based approach, on the contrary, the paring is never needed, and hence the action execution can be started more freely with previously unknown entities.

Secondly, although the BT 4 socket based communication between Android devices worked pretty well, there was no common way of making the communication work with other platforms, like iOS for instance. The problem with Bluetooth has always been that many devices support only some of the overspecialized subprotocols/services that merely allow communication with specific peripherals, but do not allow developers to specify their own communication protocols.

As the iOS has had Bluetooth Low Energy (BLE) support since version 5, we decided to try out this protocol for device coordination. BLE essentially works a bit different than its predecessors as it allows developers to define their own services. These services are then described with characteristics that can either be readable or writeable. The biggest advance is that BLE does not require pairing the devices, but instead allows them to communicate freely if they know each others protocols. The downside with BLE is that currently a device can act only in one role at a time, either as *central* or *peripheral*. However, this not an issue with Social Devices as the role of the coordinator is chosen by the server, or the Social Devices application logic to be exact, and hence the coordinating device is commanded to acts as BLE central, and the other participants are commanded to act as peripherals. As the measurement results in Table 1 show the coordination with BLE is as fast as with Bluetooth 4 sockets.

In WWDC 2013 Apple also introduced iBeacons. Whereas iBeacons (at least currently) is

nothing more than Apple's brand for BLE discovery this kind of branding may drive developers to start implementing proximity-based applications which, on the other hand, may improve the support for BLE as it is currently only supported by the iOS devices and the latest Android 4.3 devices. As from the beginning of developing Social Devices concept (since 2011) we have utilized various versions of Bluetooth discovery to detect other Social Devices nearby, and measured their distance with *Received Signal Strength Indication (RSSI)* values, we have encountered four major issues. *Firstly,* the biggest concern with Bluetooth discovery on Android devices has been that Bluetooth discovery at random times interferes with wlan, and breaks the phone's Internet connection. This happens when the two radios happens to work on the same frequency as Bluetooth changes it channel rapidly. With iOS and BLE discovery we have not experienced this kind of issues. *Secondly,* doing the discovery has been quite slow, although there has been some research of making the query faster (e.g. [7]). With BLE the discovery is very fast taking only few hundred milliseconds. *Thirdly,* many platforms, such as iOS and older Android versions only allow making the device discoverable for a short period of time. *Finally,* doing traditional discovery constantly drains the battery of the discovering device. However, although BLE offers some improvements, the discovery power consumption can still be an issue.

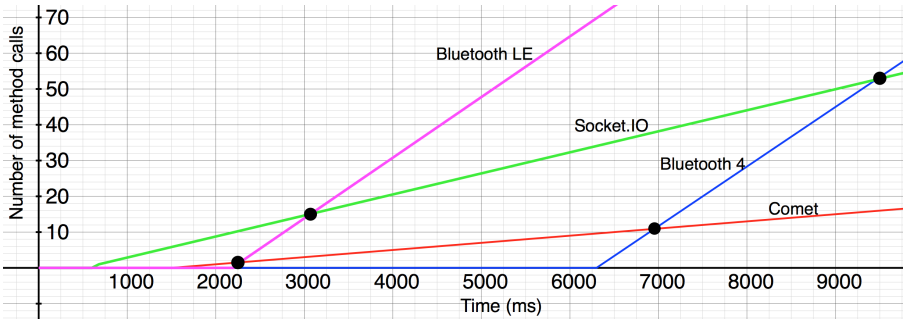


Fig. 2. Theoretically composed diagram of device coordination with different protocols.

4. Future Work

Although Bluetooth definitely offers faster coordination, the big downside is that only few devices yet support Bluetooth LE. While the support is slowly emerging to Android phones, many other devices like smart televisions and Internet of Things smart objects typically offer Internet connectivity only. In this sense cloud-based coordination can currently harness wider spectrum of Social Devices. On the other hand, some smart objects offer BLE connectivity only, and hence supporting also these may help to extend the edge of Social Devices mobile clouds. Moreover, we are currently implementing a hybrid model where during the execution of the action the devices with no BLE support could be coordinated through cloud. Furthermore, at some point we also plan to study peer-to-peer coordination, where the coordination would be distributed to each participant device, and where a token would then be used to allocate capability execution turns on each participant device.

5. Conclusions

In this paper we introduced our research of coordinating Social Devices, and how we have improved the original cloud-based coordination paradigm to better support proactive interactions between devices and people. The improvement in coordination speed has been depicted in

Fig. 2; The composed diagram shows that due to Socket.IO and Bluetooth LE the coordination is now substantially faster, and the interactions can take place in less time. Moreover, Bluetooth LE allows PAN-based communication without paring the devices, which supports Social Devices goal to make simultaneous usage of multiple devices more seamless. At the same time, Bluetooth LE makes discovering nearby devices fast, which again improves the proactiveness of the system. Finally, using JavaScript as a coordination language allows flexibly deploying the coordination logic from cloud to device to support situations where fast coordination is required.

Acknowledgements

The research was funded by Academy of Finland (264422 & 283276) and by Nokia Foundation.

References

1. Aaltonen, T., Helin, J., Katara, M., Kellomäki, P., Mikkonen, T.: Coordinating Aspects and Objects. Electr. Notes Theor. Comput. Sci., 68 (3), 248–267 (2003)
2. Back, R.-J., Kurki-Suonio, R.: Distributed Cooperation with Action Systems. ACM Transactions on Programming Languages and Systems, 10 (4), 513–554 (1988)
3. Broll, G., Rukzio, E., Paolucci, M., Wagner, M., Schmidt, A., Hussmann, H.: Perci: Pervasive Service Interaction with the Internet of Things. Internet Computing, IEEE, 13 (6), 74–81 (2009)
4. Chapman, B., Haines, M., Mehrota, P., Zima, H., van Rosendale, J.: Opus: A Coordination Language for Multidisciplinary Applications. Sci. Prog., 6 (4), 345–362 (1997)
5. Ciancarini, P.: Coordination Models and Languages as Software Integrators. ACM Computing Surveys, 28 (2), 300–302 (1996)
6. Elting, C.: Orchestrating Output Devices: Planning Multimedia Presentations for Home Entertainment with Ambient Intelligence. In: Proceedings of the 2005 Joint Conference on Smart Objects and Ambient Intelligence: Innovative Context-aware Services: Usages and Technologies, pp. 153–158. ACM, New York, NY, USA (2005)
7. Jiang, J.-R., Lin, B.-R., Tseng, Y.-C.: Analysis of Bluetooth Device Discovery and Some Speedup Mechanisms. J. Elec. Eng., 11 (4), 301–310 (2004)
8. Kray, C., Krüger, A., Endres, C.: Some Issues on Presentations in Intelligent Environments. In: Aarts, E., Collier, R.W., van Loenen, E., de Ruyter, B. Ambient Intelligence, pp. 15–26, Springer, Heidelberg (2003)
9. Manifesto for Experience of Things, http://seamlessli.github.io/mfteot/. Accessed June 5, 2014
10. Myers, B.A., Nichols, J., Wobbrock, J.O., Miller, R.C.: Taking Handheld Devices to the Next Level. Computer, 37 (12), 36–43 (2004)
11. Mäkitalo, N., Pääkkö, J., Raatikainen, M., Myllärniemi, V., Aaltonen, T., Leppänen, T., Männistö, T., Mikkonen, T.: Social Devices: Collaborative Co-located Interactions in a Mobile Cloud. In: Proceedings of 11th International Conference on Mobile and Ubiquitous Multimedia, pp. 10:1–10:10, ACM, New York, NY, USA (2012)
12. Peschanski, F., Darrasse, A., Guts, N., Bobbio, J.: Coordinating Mobile Agents in Interaction Spaces. Science of Computer Programming, 66 (3), 246–265 (2007)
13. Rekimoto, J.: Multiple-Computer User Interfaces: "Beyond the Desktop" Direct Manipulation Environments. In: Proceedings of CHI '00 Extended Abstracts on Human Factors in Computing Systems, pp. 6–7, ACM, New York, NY, USA (2000)
14. Taivalsaari, A., Mikkonen, T., Systä, K.: Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In: Proc. of 38th Annual International Computers, Software & Applications Conference (to appear), IEEE (2014)

IV

Publication IV

N. Mäkitalo, J. Pääkkö, M. Raatikainen, V. Myllärniemi, T. Aaltonen, T. Leppänen, T. Männistö, and T. Mikkonen. Social Devices: Collaborative Co-Located Interactions in a Mobile Cloud, In *Proceedings of the11th International Conference on Mobile and Ubiquitous Multimedia (MUM'12)*, pages 10:1 – 10:10. Ulm, Germany, December 4 – 6, 2012, Association for Computing Machinery (ACM).

Social Devices: Collaborative Co-Located Interactions in a Mobile Cloud

Niko Mäkitalo
Tampere University of Technology
Tampere, Finland
niko.makitalo@tut.fi

Jari Pääkkö
Aalto University
Espoo, Finland
jari.paakko@aalto.fi

Mikko Raatikainen
Aalto University
Espoo, Finland
mikko.raatikainen@aalto.fi

Varvana Myllärniemi
Aalto University
Espoo, Finland
varvana.myllarniemi@aalto.fi

Timo Aaltonen
Tampere University of Technology
Tampere, Finland
timo.aaltonen@tut.fi

Tapani Leppänen
Nokia Research Center
Tampere, Finland
tapani.leppanen@nokia.com

Tomi Männistö
Aalto University
Espoo, Finland
tomi.mannisto@aalto.fi

Tommi Mikkonen
Tampere University of Technology
Tampere, Finland
tommi.mikkonen@tut.fi

ABSTRACT

Online social media services, such as Facebook and Twitter, have set new standards on how people interact with each other online, share their everyday activities, and media services. While current mobile services supporting social interaction are typically primarily for remote communication, similar services can be introduced to co-located social interactions. In such a setting, people and proactive, context sensing mobile devices form a new kind of a socio-digital system where the mobile devices are active participants and can initiate interaction among the devices and people. Physical proximity of the devices becomes a key enabler to advance users' interaction with each other and the supporting mobile services. In this paper, we introduce the concept of Social Devices and its implementation. The Social Devices Platform facilitates autonomously composed cooperative services in co-located devices where the client part is simple and easily deployable to different kinds of devices.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; H.5.3 [**Information Interfaces And Presentation**]: Group and Organization Interfaces— *Collaborative computing, Synchronous interaction, Computer-supported cooperative work*

MUM '12 December 03 - 06 2012, Ulm, Germany

General Terms

Design, Theory, Human Factors

Keywords

social devices, proximity-based cooperation, device interaction, mobile cloud, ubiquitous multimedia

1. INTRODUCTION

Technological advances are redefining the ways how people behave and communicate, making digital communication increasingly important. Social media services, such as Facebook and Flickr, have created new means for people to find like-minded friends and to communicate regarding their everyday activities with others. While mobile devices support remote connectivity, communication still largely overlooks specific social situations. While there are obvious advantages in supporting remote social interactions, we argue that it is important to support the face-to-face, co-located interactions between people. In practice, support then relies on mobile phones or other personal devices which are in proximity of each other. Similarly, smart spaces are emerging to provide services between humans and computers in specific places, but similar kinds of smart spaces could be formed also in a mobile and ad-hoc manner between various devices.

Mobile cloud computing offers a platform for connecting mobile devices to web-based services and each other, and such services can also be introduced for co-located social interactions. In cloud computing, applications and computing resources are made available as services. When the concept is extended to the mobile domain, a so-called mobile cloud emerges, which in its simplest form refers to accessing cloud computing resources from a mobile device [14], but often also the ability to share services among mobile devices in the same cloud is assumed [23]. Mobile clouds are becoming a means for a technology platform for implementing social interactions and service sharing between mobile devices.

In this paper, we introduce the concept of *Social Devices* along with a prototype implementation called the *Social Devices Platform* (SDP). The concept of Social Devices focuses on enriching local interaction by a means of technology. The interaction can be between devices but also between humans and technology whereas the enrichment can rely almost solely on autonomous interaction within technology such as between mobile phones. Therefore, humans and proactive, context sensing mobile devices form a new kind of a socio-digital system where the mobile devices are active participants and can initiate interaction between the devices as well as with people. As traditionally in social interactions, the interaction between different co-located physical nodes is essential regardless of are such nodes humans or devices. In short, the setting of Social Devices resembles socially interactive pervasive computing between different nodes, or an ad-hoc smart space.

The rest of this paper is structured as follows. Section 2 outlines the general overview and the needs of Social Devices, and the solution concepts in the SDP. The implementation and architecture of the SDP are described in Section 3. Section 4 gives an illustrative example how the SDP operates. Related work is discussed in Section 5, whereas Section 6 discusses Social Devices and the SDP in general, pointing directions for further work. Finally, Section 7 draws conclusions.

2. SOCIAL DEVICES OVERVIEW

2.1 Motivation and Background

Social Devices aim at enriching various kinds of co-located interaction situations. When people meet face-to-face, Social Devices can augment the social interaction that takes place. Also device-to-device interaction can be enriched: Social Devices can make otherwise invisible device interaction explicit to users. Finally, Social Devices enable more intuitive interaction between users and devices, especially when devices need to give feedback to the users.

Any interaction that is enriched or enabled by Social Devices is called an *action*. A key characteristic of actions is to provide human users some visible value intelligently. The actions can be performed autonomously by the devices themselves, although the actions can also require human interaction. The actions differ largely in nature but typically involve interaction of heterogeneous devices near each other in a situation that is not mission-critical.

Figure 1 illustrates various actions that can take place between Social Devices. Meeting and greeting between people offers plenty of situations for Social Devices. For example, the devices of two businessmen can exchange contact cards and at the same time announce each other's information aloud. Social Devices may also help in situations where one cannot remember a person's name by greeting aloud on the street. Social Devices can also make invisible device interaction more explicit, for example, a laptop and a mobile phone can speak aloud their synchronization progress. As another example, a mobile phone may say aloud to a car navigator where to go in addition to setting the destination automatically based on a calendar entry. The user is then aware of the ongoing navigation and can interrupt if necessary by, e.g., specifying a new destination address. Interaction can also involve a large number of devices, for example, mobile devices can make a massive wave by flashing screens in a rock concert. Finally, actions can be instructive by nature, such as in a case when an elevator instructs a visitor to find the correct meeting room based on the upcoming entry in the mobile device calendar.

A precondition of Social Devices is the availability of computationally capable smart devices that are aware of and can interact with other devices and users in their proximity. A prime example of a social device is a mobile phone, which most people carry nowadays everywhere, although social devices can be very heterogeneous. Some devices are highly personal, such as mobile phones, whereas some are impersonal, such as meeting room screens; and some devices are very stationary, such as elevators, whereas some do not have a fixed location but move around other devices freely, such as family laptops or tablets. Finally, the devices have heterogeneous resources with varying levels of quality; such resources can include speakers, microphones, and screens.

In Social Devices, actions are executed in a very dynamic environment. Potential social situations arise as people interact and carry on their everyday tasks. Since mobile devices move in and out of range of other devices, the devices that can participate in an action change constantly. Furthermore, device states change dynamically, e.g., devices can be turned to silent mode or run out of battery.

Several other research areas are related to the concept of Social Devices. Social Devices resemble smart spaces [26]: both utilize the proximity of devices to provide services. In contrast to smart spaces, Social Devices are not tied to a particular location nor particular devices, and the actions focus on providing user-visible behavior rather than possible background services. However, some of the social devices can be stationary, such as a meeting room screen, and then Social Devices resemble more of a smart space. In contrast to pervasive computing [26, 32] or Internet of Things in general, Social Devices provide human visible actions, rather than performing actions and forming compositions in the background. In fact, central in Social Devices is the interruption of social interaction that is in a sense just the opposite to pervasive computing that has the vision of indistinguishable form or minimal user distraction [26], and actually Social Devices can be used to complement pervasive computing: Social Devices can make the background tasks performed by pervasive computing visible. In contrast to Web Services or service-oriented architecture (SOA) [20], the key in Social Devices is to utilize resources and capabilities of various physically co-located devices and the proximity of the devices in a user observable manner. Finally, instead of devices autonomously forming collaborating networks and agreeing on operations, the users should always have a control of the actions their devices participate in. Further, at least advanced users could create new actions to make devices interact.

2.2 Requirements and Characteristics

There are several practical problems to be solved in order to provide collaborative actions for Social Devices. In the following, we discuss these problems, the respective high-level solutions and thus derive requirements for the implementation of the SDP. The prototype for the SDP is described in the next section.

Firstly, the devices need to have identities, and the SDP needs to keep track of the device identities. The need for device identities is highlighted by the fact that devices col-

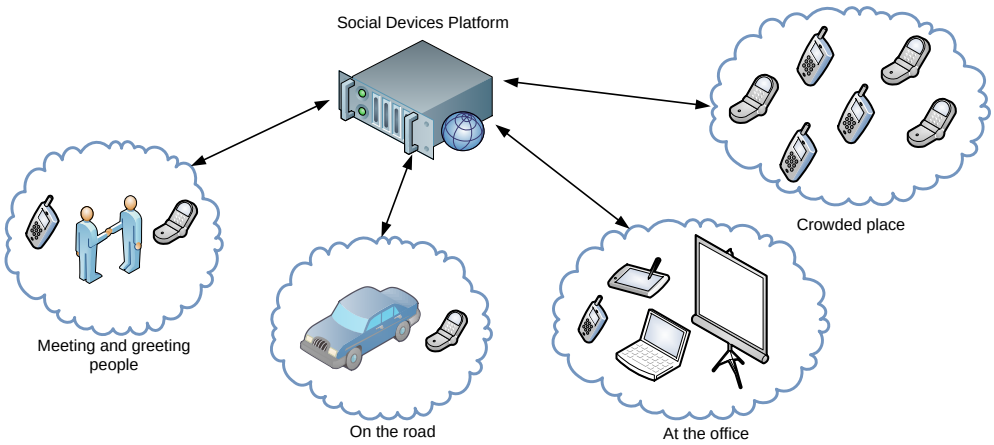


Figure 1: Example uses of Social Devices Platform (SDP)

laborate in a way that is often personal to the user and very much related to user preferences. Some users may prefer their mobile devices not to be too verbose, e.g., not to speak aloud the synchronization progress. From a technical point of view, the approach for device identity management in the SDP is to use a centralized registry where devices need to register their identity and capabilities. Moreover, as users are essential for Social Devices, the platform needs to maintain user models that indicate which device is the most personal for a user and what other kinds of devices a user has. The information about the most personal device can be used to indicate the proximity of users. The user model can also contain personality and social network information that can, for example, be used for suggesting actions in different contexts.

Secondly, devices need to know which other devices are nearby. This is because proximity is a precondition for Social Devices: co-located social interaction requires that users (and devices) are near each other. Thus, the proximity of devices needs to be discovered with respect to each device that is to participate in an action. The SDP maintains proximity information in a form of a mathematical graph, where nodes denote devices and edges denote the mutual distance of the devices. The devices measure other devices nearby and report this to a centralized server that calculates the mutual distances of the devices and updates the proximity graph. Thus, the approach has been taken to allocate much of the computing to the server side, leaving devices simple rather than relying on the devices to keep track of the proximity.

Thirdly, there is a need to know what kinds of different collaborative actions are available between devices. For this purpose, the SDP manages *action descriptions* centrally. Actions are described in two parts: the *action body*, that is, the description of what the devices should do during an action, and the *action precondition*, that is, the description of what is expected from the devices participating in the action. In order for the action description to do anything meaningful, it needs to know which kinds of device capabilities can be utilized. For example, an action *Dialog* may need to operate on devices that are able to speak aloud by synthesizing text. These expectations are characterized via *interfaces* that the devices need to fulfill, e.g., the ability to talk is characterized with a *TalkingDevice* interface. Hence, interface specifications for services in Social Devices are what WSDL is for Web Services. A social device can provide any number of interfaces: the owner of a social device can install and enable different capabilities to her own liking. By enabling interfaces and modifying interface settings, users can also adjust what kind of privileges they want to give for the SDP. For instance, from the settings for a *CalendarDevice* interface, a user can allow the SDP to use the work calendar in actions with colleagues.

Additionally, there is a need to know which kinds of devices can participate in an action. In general, such rules are captured in the action precondition, which can be any logical expression about the devices. An action precondition may require certain capabilities from the devices, e.g., that the devices implement certain interfaces. For example, an action for a dialog may require that devices have the capability to talk, i.e., the devices implement the *TalkingDevice* interface. Further, the action preconditions are not only about static capabilities of devices, but they may constrain the dynamically changing device properties. For example, it may not be enough that a device provides the *TalkingDevice* interface, but the device also needs to be willing to talk and in a relatively silent environment. Such dynamic conditions that affect whether a device can participate in an action are monitored via device *states*. For example, a boolean-valued state *isWillingToTalk* can indicate the willingness to talk. For this purpose, the devices continuously report their states to a central server; the central server then has all the necessary information to decide whether an action can take place. A state can be based on general preferences of a user or a specific state of a device as in the case when a phone is turned to silent mode.

Further, the condition when actions are executed needs to

be decided automatically at runtime. This is because social devices are autonomous in the sense that direct user interaction is not necessarily needed to start an action. Continuous searching for devices which satisfy the action precondition is computationally hard. Therefore, the SDP has taken the relatively simple approach that a *trigger* marks the need for action execution. In particular, certain actions are attempted to be scheduled when a certain trigger is received. A trigger can be an update to a state value of a device, an external event, or a change in the proximity group. Triggers and the potentially resulting actions are managed as descriptions by a centralized server, which also receives the triggers and decides whether and which action to initiate. Consequently, the intelligence of the decision about initiated actions can be built and refined in one location, on the server; the devices only need to identify the condition to raise a trigger.

In addition to identifying when to execute an action, one needs to decide at runtime which concrete devices participate in the action. The main task is to relate the potentially available devices to the preconditions stated in the action description and to decide to which roles the devices should be assigned. When a trigger for action execution is raised, there may be several potential actions and several devices that potentially fulfill the preconditions. Even the same set of devices can participate in an action in different roles: for example, in different roles in a dialog. Thus, the configuration problem in Social Devices is as follows: to find an action that matches a runtime trigger and a set of devices and device roles that fulfill the action preconditions. However, in practice the configuration problem is often eased by the action precondition restricting the number of potential devices, e.g., by requiring that the device that raised the trigger should participate in a certain role in the action, and that other participating devices are in its proximity.

Finally, the action needs to be executed. For this, the action coordination or orchestration problem requires defining a control flow for a set of independently operating devices. Since each device has been designed to run its own software — apart from mission-specific coordination tasks such as making a phone call in a cell phone — there needs to be additional features to manage the interactions taking place between the devices. In the SDP, in order to create a generic system where various cooperation possibilities are available, we decided to use a coordination approach by defining a platform into which coordination is built in.

To summarize our high-level solutions, we rely on a centralized approach rather than on device autonomy and intelligence. This is in contrast to, e.g., agent-based systems or ad-hoc networks. For example, proximity management and decisions about actions and device configurations are not the burden of the devices. As a consequence, quite simple devices can also be social devices, since complex calculation or special resources are not needed.

3. IMPLEMENTATION

3.1 Architecture

Section 2 described the characteristics and central concepts of Social Devices, and consequently, the requirements for the SDP. Figure 2 illustrates how the SDP architecture answers to these requirements. Initially, the actions that Social Devices can perform are defined beforehand (Step 1 in

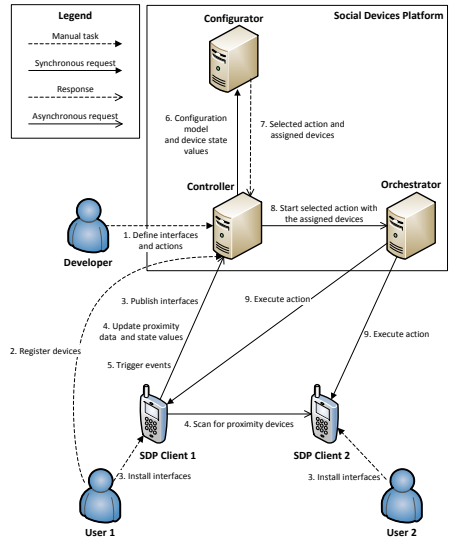


Figure 2: An overview of the SDP architecture.

Figure 2). In addition, owners of the Social Devices can register their devices and install new capabilities by installing interface implementations (Steps 2 and 3). At runtime, devices report the other devices in their proximity and their current state (Step 4). Whenever a device identifies the potential need for an action, it raises a trigger (Step 5). As a consequence, the SDP tries to find an action and a set of devices that fulfill the preconditions of that action (Steps 6 and 7). If such a configuration of devices is found, the execution of the action is started, and the execution is orchestrated among devices (Steps 8 and 9).

As shown in Figure 2, the architecture of the SDP consists of several components. On the device side, there is the *Client* component. On the server side, there is the *Controller* component, including the *Proximity* subcomponent, and the *Configurator* and the *Orchestrator* components. Each component has specific responsibilities and interacts with different actors.

3.2 SDP Client

The SDP Client contains all the necessary components for a device to operate as a social device. The main responsibility is the ability to communicate with the Controller and Orchestrator. As a first step, the device needs to register itself to the Controller and install and publish its interfaces. In other words, the device indicates that it implements the operations defined by the interface. As mentioned, enabled interfaces and their settings also define what kind of operations users allow their devices to perform and therefore also reflects to the privacy. When a device has published an interface, the device can participate in actions that require the

respective interface. The device also needs to keep the state values up-to-date.

The SDP Client can include advanced features. The user can install and enable new interfaces on a device. When a new interface is installed and enabled on a device, the device publishes this information to the Controller. The SDP Client running in a social device can trigger certain events based on the context of a device. When an event is triggered, the event is sent to the Controller.

Another key responsibility of SDP Clients is that the low-level proximity information is collected. For example, in the current implementation the Bluetooth received signal strenght indicator (RSSI) value is used for measuring the closeness of devices. The SDP Client devices keep their Bluetooth turned on and visible and periodically search for other devices. The devices then communicate the devices they were able to find along with the signal strengths to the Proximity subcomponent (in the Controller), which keeps track of the proximity at the overall level. However, as described in the following section, it is not necessary that every device collects proximity information, since the proximity information is calculated as symmetric relations and managed by the Proximity subcomponent.

The current implementation that we are using in our prototype is developed in Python for Linux devices and mobile phones. The client software has a plug-in type of architecture that allows the installation of new interfaces. However, the client software depends on the device types. Thereby, each kind of device needs an own specific client. The clients can differ in terms of how much and intelligent functionality they provide; some devices, such as a speaker, a screen, or a car navigator, can be in quite passive roles in actions. These kinds of more passive or simple devices are not required to discover nearby devices nor trigger events, since it is not necessary for all devices to be able to send triggers. Consequently, the client software can in its simplest form be quite lightweight, since the only requirement is to communicate with the Controller and the Orchestrator using a RESTful interface and to execute the implemented operations of an interface when the Controller instructs.

3.3 Controller Component

The Controller acts as a front-end for the SDP and is also responsible for managing the interaction between the other components within the SDP. Additionally, the Controller acts as a centralized registry for managing all the data required by the SDP. The data include actions, interfaces, triggers, devices, and user models. Proximity information, however, is managed by the Proximity subcomponent. The Controller provides both a web user interface (UI) as well as a REST application programming interface (API) for users and devices to access and update parts of the data.

Figure 2 shows an overview of the main interactions between the Controller and the other actors. Firstly, developers can define new interfaces and actions in Python and add them to the Controller either through a REST API or a web UI. The Controller then parses the interface and action descriptions and stores the parsed data in a database. Parsed data include the interface name, state values related to an interface, and the action precondition.

Secondly, the Controller allows users to download and install the SDP Client on their devices and thereby register their device with the SDP. Besides the interface, the Controller stores the state values of each device. The state values are used to determine whether a device can participate in an action and in which roles.

A key responsibility in the SDP is managed by the Proximity subcomponent that is responsible for maintaining up-to-date proximity information of the devices registered to the SDP. The information is maintained in the form of a mathematical graph $V = (N, E)$, where N is a set of nodes and E is a set of edges.

As noted, SDP Clients communicate the Bluetooth signal strenght between devices to the Proximity subcomponent, which in turn computes the distance based on the signal strength. This distance is attached to the edges between the nodes. Besides distance, each edge is attached a timestamp declaring when the edge was created (or refreshed, if it already existed). The Proximity subcomponent is configured to remove old edges. Currently, the time window for edges to be valid is four times the length of the period the device clients carry out Bluetooth discovery.

The Proximity subcomponent offers a REST interface for querying a proximity set with respect to a device. For example, it is possible to search for devices within a 20 meter radius from device d. The search returns a set of nodes of the induced subgraph of neighbors centered at device d within a given radius (20 meters). Naturally, this is only an approximation of the graph, since not all nodes are connected, and thus some devices might be closer to d than they appear in the subgraph. The Proximity subcomponent has one more responsibility: when a new proximity set is formed, it sends a so-called GetCloseTrigger to the Controller.

The Controller also provides a REST API through which the SDP Client can update the proximity data. In addition, a similar REST API can be used for triggers, although state value updates can also be interpreted as triggers. Regardless of the origin of a trigger, the Controller processes the triggers and forms a request that is sent to the Configurator. The request consists of a configuration model and the state values of the devices that are in proximity of the triggering device. The configuration model is generated at runtime based on the proximity group of the triggering device and a set of actions that have been linked to the triggered event. The Configurator uses the request to select an action and assign devices to each of the roles of the action. The tasks performed by the Configurator are described in more detail in the next section. Finally, the Controller sends a request to the Orchestrator to start executing the selected action.

3.4 Configurator Component

For the coordination of social devices, the SDP needs to be able to determine whether an action is possible for the devices in a proximity set so that the devices can be validly assigned to the action roles, i.e., adhering to the action preconditions. In addition, the SDP needs to select the action to be executed, typically from several possible actions and several possible ways how the devices can be assigned to the actions. For instance, for a device to be assigned to a specific role in an action, the device must implement all of the interfaces required by the role. Furthermore, the device must satisfy the state value conditions that have been specified in the precondition for the action role. This challenge can be formulated as a configuration problem where, given an action and a configuration model, the task is to find a valid configuration of devices assigned to the action roles.

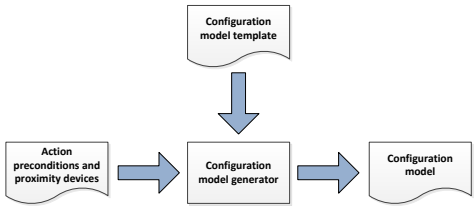


Figure 3: An illustration of the configuration model generation process.

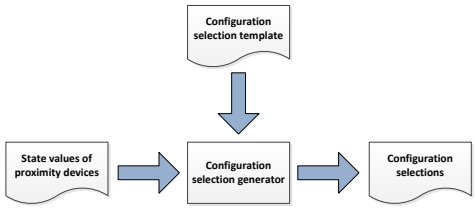


Figure 4: An illustration of the configuration selection generation process.

The configuration model captures, for example, action preconditions and interface definitions.

The Configurator provides configuration as a service for finding a valid configuration given a configuration model and a set of configuration selections. Since the proximity groups are dynamic, the configuration model needs to be generated at runtime based on information of the triggered actions and the proximity group. The configuration selections, on the other hand, consist of device state values. Based on the configuration model and the configuration selections, the Configurator finds a valid configuration if one exists and returns the configuration to the Controller.

In the current implementation, the Configurator utilizes the *smodels* inference engine [27], which is a general-purpose inference tool based on the stable model semantics of logic programs. Therefore, the configuration model and configuration selections are represented using Weight Constraint Rule Language (WCRL), a general-purpose knowledge representation language. The Controller uses a template-based generator for generating the configuration model and selections. The input for the configuration model generator consists of the preconditions of the set of actions linked to the triggering event and the devices in the proximity group of the triggering device. As shown in Figure 3, based on the input and the configuration model template, the Controller generates a configuration model. Similarly, the Controller generates the configuration selections based on the state values of the proximity devices and the configuration selection template as shown in Figure 4. The Configurator receives both the configuration model and selections and uses them to derive a valid configuration, i.e., an action with devices assigned to the specific roles of the action.

Configuration problems can be computationally expensive due to the combinatorial explosion of the number of configurations. In the worst case, the time required to find a valid configuration grows exponentially with the number of variability points. In the SDP, the variability points consist, for instance, of proximity devices and action roles. Typically, however, the number of proximity devices and the amount of roles in an action are limited. Our initial test runs indicate that finding a valid configuration in the context of the SDP seems feasible using a standard PC. For instance, finding a valid configuration for 1000 devices and two actions, with two or three roles, takes approximately 2 seconds. In general, the size of a proximity group is less than 100. Additionally, since the triggering of actions in the SDP is not time critical and does not necessarily involve user interaction, the response times only need to be within a few seconds at most. As such, the response times of finding a valid configuration are feasible.

3.5 Orchestrator Component

In software development, coordination languages [3, 11] have originally been used to define joint actions of actors constituting a software system. Similar to our case, each actor is responsible for its own local operations, whereas additional coordination, expressed using an additional coordination language, defines how cooperation between the actors actually takes place. In our approach we consider devices within the same proximity set to be actors in an action, and these actors are being coordinated by a central server.

For us, one of the main design drivers for the implementation was the use of already existing implementation mechanisms, allowing us to focus on the essential properties of coordination. Although we originally experimented with various approaches, we decided to compose the final implementation using Python instead of using a specific coordination language. However, the fashion in which Python is used closely reflects the properties of coordination languages.

The Orchestrator is a centralized server that is responsible for executing the actions, the *action body* parts to be more specific. The action executions are tied to a predefined set of devices which the Configurator component has selected. The coordination of the devices is based on services, which users can install to their devices and which can be enabled and disabled according to the users' wishes and the usage context of the device. Each service has an interface for operations, which defines what the service is for and how the actor device behaves when an operation is invoked.

The service interfaces have their counterparts on both the server and the actor devices as illustrated in Figure 5. On the server side, the services are implemented in Python and are therefore easily callable from the action, as we are using Python as the coordination language. On the device side, the implementations of these services are coupled to the platform and its programming language. Moreover, because the way of implementing features varies among different platforms, the service implementations and the quality may vary as well. In other words, each actor device is responsible for its local operations, as mentioned.

The Orchestrator has a REST API for initiating the action executions. When action execution begins, all the actor devices are allocated for the duration of execution. This design decision was made to offer a more transactional way for the devices to behave, as they can rollback the changes in case of an error. It is also clearer to the users if their devices are participating to one action at a time. When users are

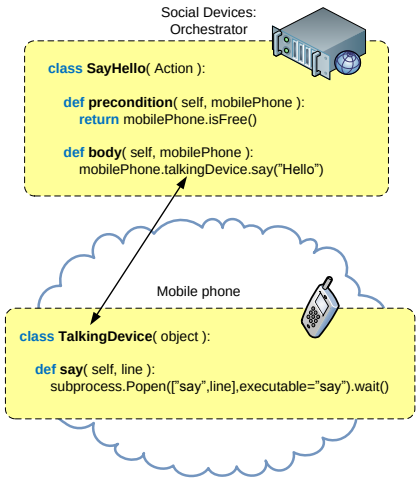


Figure 5: Execution of service operations

aware of what their devices are doing, they feel that they have more control over their devices and the execution of the action. On the other hand, several actions can be executed at the same time inside the mobile cloud, although this requires that separate devices will be selected to separate actions, as one device can only take part in one action at a time.

For initiating the action, a set of parameters is given for the API. The parameters are used to dynamically initialize the action and the actor devices. In the initialization process, the Orchestrator allocates the actor devices by ensuring that the devices are still available and by allocating the services that the devices are providing. After a successful initialization process, the action execution starts and the interface returns a response.

The Orchestrator uses a synchronized way for calling the actor devices' service operations. This design decision allows the platform to relay return values directly from actor devices to action execution processes and hence the operation calls act like regular method calls in Python. This makes it illustrative for users to create their own actions and develop new services.

In some cases, the whole action execution can depend on a single operation executed by an actor device, in case of which the response value can be used by other operations. To avoid only partially functioning actions and to give better feedback for users, the Orchestrator supports relaying exceptions from failed operations of actor devices to the Orchestrator, and furthermore, all the way to other devices executing the same action as well. When an exception is raised, all the actor devices get a message describing which part of the system failed and in what operation and then get released from the action. On the other hand, if any errors occur while executing the action definition on the server

```
class GreetEachOther(Action):
  @actionprecondition
  def precondition(self, d1, d2):
    return proximity([d1, d2], 2.0) and
           d1.hasInterface(TalkingDevice) and
           d2.hasInterface(TalkingDevice) and
           d1.isFriend(d2)

  @classmethod
  def getTriggers(cls):
    return ['GetCloseTrigger']

  @actionbody
  def body(self, d1, d2):
    s1, s2 = smallTalkServer.getSentenceAndReply(d1, d2)
    d1.talkingDevice.say("Hello, " + d2.getOwnerName() + "!")
    d1.talkingDevice.say(s1)
    d2.talkingDevice.say("Oh, hello " + d1.getOwnerName() + "!")
    d2.talkingDevice.say(s2)
```

Figure 6: A simplified version of action definition GreetEachOther.

side, an exception is raised and relayed to the actor devices by using the same functionality. The functionality can also be used to relay user interruptions and interface events. For instance, in the current SDP Client, users can cancel the actions by shaking the device.

4. EXAMPLE: GREETING DEVICES

4.1 Example Action

This subsection describes a simple entertaining action of greeting devices. Assume that Alice, Bob, and Carol have gathered together, and after a while Dave joins them. Now, someone's (perhaps Carol's) personal device says aloud: "Hello Dave! How was your trip to London?", and Dave's device replies: "Oh, hello Carol! The trip was great!". This example illustrates how Social Devices can enrich a common social interaction scenario: two or more friends meet, and their devices greet each other in a personalized way. This kind of personalized content and location information can be gathered from the social media, such as Facebook or Foursquare, and even from calendar entries if users allow.

In order for the SDP to implement this example scenario, several things must have happened. Firstly, all stakeholders must have registered themselves and their devices to the SDP. The devices must have installed the necessary talking capability for this action (in particular, the implementation for the *TalkingDevice* interface). Information of the friendships and necessary personalized content (status updates or calendar entries) must be available. Moreover, someone must have defined and uploaded the action *GreetEachOther* (Figure 6).

Figure 6 contains the action description for the action *GreetEachOther*. The action description has two parts: the action precondition and the action body. The action precondition must evaluate to *true* for the action body to be executed. In this example, the precondition requires that two devices (devices in roles *d1* and *d2*) must be less than 2 meters apart and both must be able to talk (have implemented the *TalkingDevice* interface). In addition, since we do not want to greet strangers, the users of the devices must be friends. In general, the precondition being true does not guarantee the execution of an action, but it only states that it is possible for the action to get executed.

Generally, actions are attempted to be scheduled when a certain trigger is received. In our example, the action *GreetEachOther* is attempted to be executed when the *GetCloseTrigger* is received. The attempt means that the SDP tries to find the devices *d1* and *d2* for which the precondition evaluates to *true* after which the SDP tries to execute the action.

The action body describes what the participating devices *d1* and *d2* do. In the example, the personalized content is generated by the *smallTalkServer*, which is able to produce small dialog fragments. For example, Dave has made a check-in with a status update in London by using Facebook, so the server created the sentences: "How was your trip to London?" and "The trip was great!".

Other auxiliary functions are omitted from the example for simplicity. Moreover, some simplifications have been made to the precondition.

4.2 Resulting System Behavior

The different parts of the action description *GreetEachOther* are used by different components in the SDP architecture. The Configurator parses the precondition to be able to find the devices that fulfill the precondition. The Controller uses the class method *getTrigger* and saves the trigger-action-pair (GetCloseTrigger, GreetEachOther) in its data structures. The action body is used by the Orchestrator.

When Alice, Bob, and Carol gathered together, their devices noticed this (for example, using Bluetooth). This was communicated to the Proximity subcomponent, which had (at the moment) edges between the nodes representing Alice's, Bob's, and Carol's mutual distances. When Dave — after a while — approached the group, the approach was recognized similarly (using Bluetooth), and this was communicated to the Proximity subcomponent. The change in the proximity information caused the Proximity subcomponent to send the GetCloseTrigger to the Controller, including information of the newly formed set of devices.

The Controller had registered earlier that the *GetCloseTrigger* is related to the action *GreetEachOther*. Using the auxiliary function *getActionAndRoles* (which is omitted here), it might have been able to set some devices to roles already. Then, using the Configurator, it would have been able to find the devices for the rest of the roles. At this point, the Controller had the action to be executed (*GreetEachOther*) and the devices (Carol's phone and Dave's phone) assigned to the roles. These two pieces of information were then sent to the Orchestrator, which orchestrated the action execution.

5. RELATED WORK

Social Devices, and the SDP in particular, are based on principles adopted from existing technologies. In fact, one design driver has been to utilize existing work in the design of the SDP. In general, the SDP follows some approaches that are similar to a service-oriented architecture (SOA) approach to software development [20, 21]. However, the interaction between components in the SDP adheres to the REST principles [10]. In particular, the two central servers, the Configurator and the Orchestrator, have been influenced by and are related to earlier research.

The Configurator is based on the concepts of knowledge-based product configuration, which aims at satisfying different customer requirements through mass customization [25]. Instead of explicitly enumerating all products, a product is configured from a standardized set of well-defined components, which interact with each other in a predefined way [7, 25]. To achieve this, configuration knowledge, typically represented in a configuration model, needs to capture the rules on how these sets of parts can be combined. Example application areas of product configuration include the computer hardware industry, the telecommunications industry, the automotive industry [8], and traditional services [29]. The concepts have also been applied to software [13, 18] within the broader research topic of software variability [28]. Within the domain of software architecture, the need for dynamic adaptation has been identified. For example, [17, 30] use explicit models of components and connectors as runtime artifacts, allowing architecture-based adaptation. Furthermore, dynamicity in software variability has received increasing attention, such as in a form of a dynamic software product line [12]. Our work builds directly on the results from these domains and provides a significant difference due to the application in an autonomous platform. Especially, the dynamic generation of a configuration model is an open area of research to which we provide an initial solution.

The approaches for coordinating multiple devices have mainly been focusing on information presentations (e.g. [6, 15, 19]), or for multimedia resource synchronization (e.g. [23, 24]). However, our work is different, since we are not aiming to offer only automated services or new kinds of interfaces. For example, in [6], we find similarities in the approach for coordinating the devices, but the aim is different. As [6] and [19] focus on generating user interfaces and coordinating them on the devices, the system philosophy is more user-centric than ours. We, on the contrary, aim to make devices interact and socialize independently, and make the operations visible for the users. When the majority of approaches focus on coordinating the devices in predefined locations, such as smart spaces or homes, our focus is in coordinating the devices wherever they are in the proximity of each other in any location. Several approaches have also been proposed for the modeling and specification of collective actions (e.g. [2, 16]) and for coordinating computational resources (e.g. [1, 3, 4, 5]). We are revitalizing the idea by applying it to mobile clouds, where actors correspond to individual devices forming the cloud, and whereas a central server is responsible for coordinating the execution of mobile devices. In previous research, the closest relative to our approach is constituted by coordination languages for mobile agents (e.g. [22]). However, our work is different from these, since we are treating complete mobile devices as agents. Consequently, the granularity of the coordination is fundamentally different from agent-based approaches.

6. DISCUSSION

The first prototype implementation, the SDP, has been implemented but there are several further research challenges, as discussed in the following.

The decomposition of the SDP architecture (Figure 2) aims at a clear separation of responsibilities with the use of RESTful interfaces; this has allowed us to develop each component separately. For example, the use of a configurator as a service has also allowed us to experiment with different kinds of intelligent tools without largely affecting other parts of the SDP components. Furthermore, we have focused mostly on the server side whereas the SDP Client

only provides basic functionality. However, the decomposition may not be optimal from, e.g., a performance point of view. Many of the architectural pieces are quite efficient but not yet optimal, especially due to the nature of extension discussed in the following.

The current approach for configuration returns one arbitrary, valid configuration and there is practically no means to affect what kind of configuration will be returned. However, the capabilities and the quality of service (QoS) of participating devices affect the overall quality of an action. As an example, the audio quality depends on the quality of the device speakers. Further, contextual data may also affect the quality of service of a service composition: for instance, the user-perceived audio quality depends on the distance from a display or loudspeaker. Finally, device users have different kinds of preferences. Thus, given a set of devices with varying capabilities and level of quality of service, the problem becomes to find the best or good enough devices for the action roles. The configuration of an action is typically an under-constrained problem to which there are several valid configurations. One advanced approach is to use recommender systems as a means for finding devices so that user preferences and device capabilities are taken into account. Recommender systems are information systems that have been introduced in various domains for proposing items to users based on user preferences using techniques from the field of artificial intelligence [9]. Our initial feasibility studies indicate that technically recommender systems could be applied instead of configurators by representing the configuration problem as a recommendation problem. Furthermore, despite the computational complexity of the recommendation problem, the response times also seem to be quite feasible. However, recommendation adds challenges, such as whose preferences should be taken into account in social situations and how user preferences should be captured.

Although Social Devices are not critical in the sense that an action needs to recover or terminate properly, the highly dynamic nature can result in that actions do not execute properly, which can become frustrating and inconvenient. Therefore, the reliability of the SDP would benefit from various additional supporting services. In particular, the SDP Client provides several opportunities for advanced context-aware behavior and self-management capabilities.

The action and interface definitions follow certain predefined conventions and are in Python due to its simplicity and, in particular, the use of existing parsers. Currently, however, there are no means to support and ensure that the correct conventions are used when defining the actions and interfaces in Python. An integrated development environment (IDE) would offer an approach to easily construct actions. One approach is to use a plug-in for the Eclipse open source IDE (www.eclipse.org) that then connects to the RESTful API of the SDP to upload actions. Furthermore, it seems to be possible to ease action definition by a domain-specific language, with even a graphical notation, that is then transformed into Python. Such an approach could even be used for enabling end-user programming of actions. Additionally, such conventions would facilitate application deployment, using different programming languages, to platforms that do not readily support Python. Nevertheless, Python seems to be quite a feasible coordination language for the actual action coordination requirements.

The Implementation of the SDP is currently device-centric, although social interactions as well as devices are personal. Users even have several devices and sometimes share devices, e.g., with family members. To enhance personalization and contextualization of social devices, the concept of users, including social relations to other users and devices, is required. For example, the interaction with a friend's device is most likely different compared to one with a colleague at work.

Privacy and security are also issues that require further investigation. Currently, users are able to enable and disable capabilities from their devices, and adjust settings to their liking. However, as the platform becomes more complex, strict privacy policies will be needed.

Currently, the SDP uses low-level contextual data [31], such as proximity and device state obtained through device sensors, to trigger and determine which actions can be executed. This kind of low-level contextual data may not in itself be sufficient to determine more high-level contexts such as specific social interactions. Instead, to be able to sense more complex social contexts, the SDP needs to have a way to infer high-level contexts by, e.g., combining data from several low-level sensors or by using application data such as a calendar. The identification of specific social situations can also help to increase the privacy of users by not allowing certain actions to be executed in certain situations.

The concept of Social Devices intervene with established norms in social interactions between people. It would be essential to thoroughly understand how users perceive the actions and what kinds of actions users want to have. The intervention of social interactions can in certain situations raise different kinds of feelings. Some user experience research has already been made for the concept of Social Devices: the concept itself has received quite a polarized response — some have questioned the concept completely while others have been quite enthusiastic. For example, it was considered tempting that Social Devices would violate certain social norms. Only a few have stayed neutral. Nevertheless, the contribution of this paper has been to introduce the concept of Social Devices while more studies in the real social context need to be carried out.

7. CONCLUSION

The way people communicate and socialize has changed due to social media services. Whereas these services are useful in many ways for supporting remote social interactions, they offer only limited advantages for face-to-face and co-located situations. In this paper, we introduced the concept of Social Devices that offers a new kind of socio-digital system for co-located devices and humans to interact with each other. The interactions are based on actions, which are predefined processes where devices are used in certain roles, and where each device is offering certain services according to its capabilities. In contrast to smart spaces, the actions are not tied to any specific location nor devices, and the devices interact in a more ad-hoc manner. On the other hand, the processes are made visible for the users, and moreover, users may interact in these actions as well.

The SDP was introduced as a prototype implementation for the concept of Social Devices. The SDP offers components for the main practical problems that have emerged with Social Devices. These problems are: tracking the proximity of devices, finding a suitable configuration for a set of devices within close proximity, and orchestrating the oper-

ation executions on the devices. While the SDP offers solutions for the main problems, more research is still needed in many areas. For instance, personalizing contents of the actions for users in different contexts requires further studies. Also understanding how people thoroughly understand the idea of Social Devices and the actions requires studies in real context. However, as we designed the client to be lightweight, implementing it to other mobile platforms happens in a near future, allowing us to focus on studies on a more larger scale.

8. REFERENCES

[1] T. Aaltonen, J. Helin, M. Katara, P. Kellomäki, and T. Mikkonen. Coordinating aspects and objects. *Electr. Notes Theor. Comput. Sci.*, 68(3):248–267, Aug. 2003.

[2] R.-J. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Trans. Program. Lang. Syst.*, 10(4):513–554, Oct. 1988.

[3] B. Chapman, M. Haines, P. Mehrota, H. Zima, and J. Van Rosendale. Opus A coordination language for multidisciplinary applications. *Scientific Programming*, 6(4):345–362, Oct. 1997.

[4] P. Ciancarini. Coordination models and languages as software integrators. *ACM Comput. Surv.*, 28(2):300–302, June 1996.

[5] J. Darlington, Y. ke Guo, H. W. To, J. Yang, H. Wing, and T. J. Yang. Functional skeletons for parallel coordination. In *EURO-PAR'95 Parallel Processing*, pages 55–69, 1995.

[6] C. Elting. Orchestrating output devices: planning multimedia presentations for home entertainment with ambient intelligence. In *Smart Objects and Ambient Intelligence sOc-EUSAI*, pages 153–158, 2005.

[7] B. Faltings and E. C. Freuder. Configuration (Guest Editor's Introduction). *IEEE Intelligent Systems*, 13(4):32–33, July 1998.

[8] A. Felfernig, G. Friedrich, and D. Jannach. Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering*, 15(2):165–176, Apr. 2001.

[9] A. Felfernig, G. Friedrich, and L. Schmidt-Thieme. Recommender systems. *IEEE Intelligent Systems*, 22(3):18–21, May/June 2007.

[10] R. T. Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] D. Gelernter and N. Carriero. Coordination languages and their significance. *CACM*, 35(2):96–107, Feb 1992.

[12] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, Apr. 2008.

[13] L. Hotz, K. Wolter, T. Krebs, J. Nijhuis, S. Deelstra, M. Sinnema, and J. MacGregor. *Configuration in Industrial Product Families: The ConIPF Methodology*. IOS Press, Sep. 2006.

[14] A. Klein, C. Mannweiler, J. Schneider, and H. D. Schotten. Access schemes for mobile cloud computing. In *First International Workshop on Mobile Cloud Computing at MDM '10*, pages 387–392, 2010.

[15] C. Kray, A. Krüger, and C. Endres. Some issues on presentations in intelligent environments. In *EUSAI '03*, pages 15–26, 2003.

[16] R. Kurki-Suonio and T. Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In *PDSE '98*, pages 94–102, 1998.

[17] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, Oct. 1996.

[18] T. Männistö, T. Soininen, and R. Sulonen. Product configuration view to software product families. In *Proc. of the SCM-10 workshop at ICSE '01*, pages 14–15, 2001.

[19] B. A. Myers, J. Nichols, J. O. Wobbrock, and R. C. Miller. Taking handheld devices to the next level. *Computer*, 37(12):36–43, Dec. 2004.

[20] M. Papazoglou. Extending the service-oriented architecture. *Business Integration Journal*, 7(1):18–21, Feb. 2005.

[21] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, Nov. 2007.

[22] F. Peschanski, A. Darrasse, N. Guts, and J. Bobbio. Coordinating mobile agents in interaction spaces. *Sci. Comput. Program.*, 66(3):246–265, May. 2007.

[23] M. Raatikainen, T. Mikkonen, V. Myllärniemi, N. Mäkitalo, T. Männistö, and J. Savolainen. Mobile content as a service a blueprint for a vendor-neutral cloud of mobile devices. *IEEE Software*, 29(4):28–32, July/Aug. 2012.

[24] J. Rekimoto. Multiple-computer user interfaces: "beyond the desktop" direct manipulation environments. In *CHI '00 extended abstracts on Human factors in computing systems*, pages 6–7, 2000.

[25] D. Sabin and R. Weigel. Product configuration frameworks—a survey. *IEEE Intelligent Systems*, 13(4):42–49, July 1998.

[26] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, Aug. 2001.

[27] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, June 2002.

[28] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Softw. Pract. Exper.*, 35(8):705–754, July 2005.

[29] J. Tiihonen, M. Heiskala, K.-S. Paloheimo, and A. Anderson. Applying the configuration paradigm to mass-customize contract based services. In *MCPC '07*, page 7.5.3 on the proceedings CD, 2007.

[30] A. van der Hoek. Design-time product line architectures for any-time variability. *Sci. Comput. Program.*, 53(3):285–304, Dec. 2004.

[31] G. Chen and D. Kotz. A survey of context-aware mobile computing research. In technical report *TR2000-381*, Dartmouth College, 2000.

[32] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.

V

Publication V

N. Mäkitalo, T. Aaltonen, and T. Mikkonen. First Hand Developer Experiences of Social Devices, In *Advances in Service-Oriented and Cloud Computing*, Volume 393 of the series *Communications in Computer and Information Science*, C. Canal, M. Villari (Eds.), pages 233 – 243, 2013, ISBN 978-3-642-45363-2, Springer.

First Hand Developer Experiences of Social Devices

Niko Mäkitalo, Timo Aaltonen, and Tommi Mikkonen

Department of Pervasive Computing
Tampere University of Technology, Tampere, Finland
PL 553, 33101 Tampere, Finland
{niko.makitalo, timo.aaltonen, tommi.mikkonen}@tut.fi

Abstract. Contemporary Internet connected devices, such as tablets and mobile phones, have excellent computing power, which creates a possibility for complex, cooperative multi-device platforms. We have introduced a concept of Social Devices and its reference implementation Social Devices Platform. The system offers an intuitive way to build interactions between co-located people and their devices, and then trigger these when people meet face-to-face. In this paper we study how developers experience the concept and the platform. We hired a four-person team to design and implement a multiplayer game, and afterwards interviewed the team members about their experiences. Based on their feedback we evaluate the system. Moreover, we raise some open questions that require attention and more research in the future.

1 Introduction

Custom-built native apps have become one of the dominant ways people use software. In the mobile space, the time span of the users' actions is usually significantly shorter than in the desktop space; the users wish to perform rapid, focused actions instead of long-lasting sessions; actions must be simple yet focused, and they must be accomplished with ease, using only a minimal number of keystrokes or finger presses, often while the user is walking, driving a car or is somehow otherwise distracted by other activities. The different usage modalities and smaller screen sizes have a significant impact on application design; generic web pages geared towards laptop or desktop computer users are not usually ideal for mobile use.

While numerous apps for mobile devices are meant to be social – think about mobile Facebook and Twitter clients, Instagram, and Foursquare – the actual means for programming follow the traditional device centric development approach. For instance, in the context of iPhone, apps are defined as individual applications that are separately activated by the user, and communication patterns follow the practices that have been developed for conventional networking.

The benefits of using an already established application model are many. Users are accustomed to installing and activating applications, and appear to be willing to do so. From the developer perspective, development tools and the

programming model are already familiar, and although at times some of the design details appear cumbersome and impractical, the fact that these issues are similar in most settings have taught us to circumvent them in designs.

However, smart devices have excellent computing power and connectivity and at the same time are used for various purposes. Moreover, we have learned to accept that mobile devices play more and more proactive role in daily activities. This creates the possibility for complex, cooperative multi-device programs, for which current programming paradigms are not well-suited. Hence, we have tackled the above problem by introducing a new paradigm: an action-oriented programming model for pervasive computing [1]. Actions are proactively initiated pieces of functionality, which synchronize and coordinate joint behavior of several devices. The action-oriented programming model is realized within the cloud-based Social Devices Platform. So far, the technical feasibility of the approach has been demonstrated in our earlier papers [3], together with the description of applications that have been developed using our platform. However, the developer perspective has not been addressed. At the same time, the developer experience is a key issue in obtaining a large number of applications that are available for end users.

In this paper, we present experiences gathered from outside developers who have been using the platform to develop an application during Spring 2013. The paper includes both the developers' opinion about the concept as well as experiences in programming using it. The application was developed in cooperation with Demola[1], an innovation instrument targeted for fostering innovation and experimenting with radical ideas.

The rest of this paper is structured as follows. In Section 2, we discuss the background of this work. In Section 3 we introduce the developer team and their application. In Section 4 we present the results regarding developer experience, and list claims regarding the way of developing applications with the Social Devices platform. In Section 5 we briefly address related work, and in Section 6 we pinpoint some open issues. In Section 7, we draw some final conclusions.

2 Background

The concept of *Social Devices* and its reference implementation *Social Devices Platform (SDP)* was first introduced in [3]. The aim of Social Devices is to increase, facilitate, and enrich social interactions between people in various kinds of co-located and face-to-face situations. For example, when people meet, the devices can greet each other aloud to help people to remember each others' names, or the devices can automatically chance contact information when a group of businessmen meets. Also device-to-device interaction can be enriched: Social Devices can make otherwise invisible device interaction explicit to users. For instance, a mobile phone can say aloud to a car navigator where to go in

[1] `http://www.demola.fi`

addition to setting the destination based on a calendar entry. Moreover, Social Devices can be used for suggesting and proactively initiating social multiuser applications when like-minded are nearby. For example, when friends meets in cafeteria Social Devices can proactively suggest them to view photos if one of the friends has added a new album to Flickr and enabled photo sharing feature on her phone. Another obvious example of social applications are games that can be suggested for co-located people, for instance when people are traveling by the same bus or are in the same bar.

The Social Devices Platform currently runs in Amazon cloud, consisting a number of cloud services, as can be seen in Figure 1. The client side currently consists of Android devices and Python capable devices, such as Linux laptops and MeeGo phones. As the system infrastructure is cloud based, it abstracts the differences of the devices. However, instead of hiding or ignoring the different resources of each device, Social Devices accents the differences by regarding them as capabilities, which describe what a device can do. For instance, the device may have a TalkingDevice capability meaning that the device is equipped with text to speech translator. In addition to the existing capabilities, the platform allows developers to create their own new types of capabilities, which other users can then install in their devices.

The interactions between devices are defined in terms of Actions, which is a novel modular unit for describing how several co-located devices operate together. The Actions are defined with classes of Python programming language, and they contain a precondition and a body methods. The precondition is used for defining when the Action can actually take place and what is required from the participating devices. The body part is used for defining the device coordination logic with the help of device capabilities. Naturally, the platform also offers an easy way for developers to define their own Actions.

3 Experiment

To get input from developers who have no previous experience regarding the concept of Social Devices, we hired a four-person developer team from Demola. The team had a project manager, a graphic designer, and two software developers. The project manager and the graphic designer had only limited knowledge about software development, and also the two developers had no previous experience of pervasive system or mobile cloud development. The team was given free hands to develop what they wanted, and they came up with an idea about a set of mini Olympic Games, where users' avatars are zombies. Each zombie comes from different country, and hence has its stereotypic characteristics. The game was named as *Apocalympics*. See demo[2].

An example setting for a gaming scenario could be as follows: Alice and Bob don't know each other, although they travel daily by the same bus. Bob is interested in playing games and also likes zombie splatter movies. Alice also

[2] `http://youtu.be/ZzhUiwO-vl4`

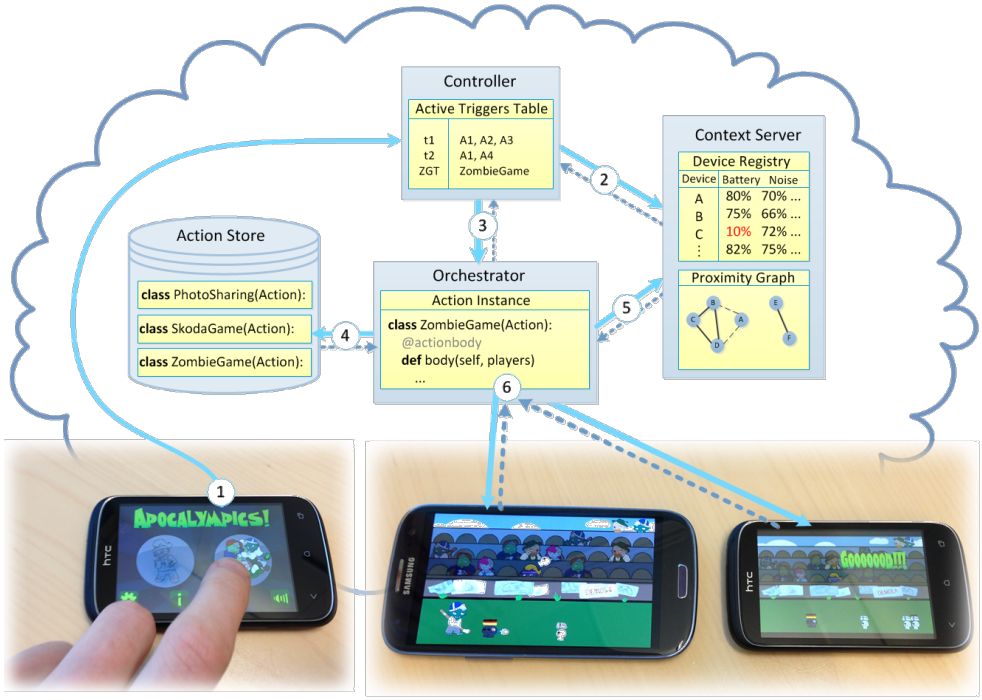


Fig. 1. The SDP architecture and work flow in Zombie game scenario.

likes to play some games every now and then. Now, when Alice gets bored in the bus, she can indicate this to Social Devices Platform: By pressing a ZombieGame icon on her phone, she can challenge someone nearby to play with her. The Social Devices Client then sends a ZombieGameTrigger to Controller component (see phase #1, in Figure 1). The system knows what other Social Devices are near Alice as the clients gather proximity information that is based on Bluetooth signal strengths (RSSI), and periodically report this and other device state information to the Context Server.

The Controller component uses the proximity and state information, while it tries to find participants for the Action (see phase #2). As Bob happens to be in the same bus and has several likes of zombie movies in Facebook, the Controller chooses him as one of the players. Carol, who is also in the bus, is not selected to participate as her battery is almost dry. After collecting the players, the ZombieGameTrigger with the participants is send to the Orchestrator component (see phase #3). The Orchestrator fetches the Action body (see phase #4), and then allocates the devices for the ZombieGame action (see phase #5). Finally, it starts coordinating the devices (see phase 6). Now, if Bob wants he can of course decline from the gaming challenge.

Figure 2 shows how the team defined the game with the terms of Action. The total number of code lines was 240, but from Figure 2 some boilerplate

```
class Apocalymbics(Action):
  # Player, Game and Constant class deifinitions omitted (11 rows)

  @actionprecondition
  def precondition(self, players):
    return proximity(players, 15.0) and\
     haveFacebookLikes(players, ['Gaming', 'Zombies'])

  @actionbody
  def body(self, players):
    # initializations, game and country selection omitted (66 rows)

    # Mainloop exists until less that 2 players
    while (len(players) > 1):
      # Mini Game initializations omitted (14 rows)

      # GAME PLAY LOOP FOR EACH MINI GAME STARTS HERE
      if game == Game.gameSkullThrow:
        random.shuffle(players)

        for round in range(3):
          for player in players:
            player.device.zombieGame.startRound(round)

        # Loop every player's turn of current round
        for currentPlayer in players:
          # Inform clients which player is in turn
          for player in players:
            player.device.zombieGame.updateTurn(currentPlayer.number)

          # Get player's throw length
          currentPlayer.throwAngle = Constant.uninitialized
          while (currentPlayer.throwAngle == Constant.uninitialized):
            currentPlayer.throwAngle = currentPlayer.device.\
             zombieGame.getThrowAngle()

          # Inform everyone how far the player managed to throw
          for player in players:
            player.device.zombieGame.showThrow(currentPlayer.number,\
             currentPlayer.throwAngle)

          # Wait until every client has shown the throw
          for player in players:
            ready = False
            while not ready:
              ready = player.device.zombieGame.isReady()

        # Winner Screen and next game choosing omitted (32 rows)

    # Game ends
```

Fig. 2. The definition of the Apocalympics Action

code was omitted for brevity. The omitted parts mainly contained capability method calls that were related to the game initialization on the devices, such as selecting the country for the avatar. Also code for showing the winning screens and game selection were omitted, as well as three developer defined classes: Player, Game and Constant.

The precondition (lines 5-7) shows that the Action can take place if the participating players are less than 15 meters apart from each other, and that they

have liked Gaming and Zombies on Facebook. The server side game and coordination logic is defined in the body part of the Action (lines 9-50). For example, on lines 22-23 each player's device is commanded to start a new round on SkullThrow mini game. On lines 32-35 the game waits until the player in turn has thrown the skull, and on lines 39-40 this throw is then shown to other players. The communication behind the capability calls is taken care by the Orchestrator component, and developers don't need to care about its dirty details.

4 Developer Experiences

In this section we review the experiences of the hired team, and evaluate the Social Devices concept and platform based on the team's feedback. The study was conducted by interviewing the team members after the project. The heterogeneous background of the members gave us an opportunity to get insight from different perspectives. With the project manager and the graphic designer we focused more on general and concept level questions. With the two software developers we focused more on the technical details, and tried to find out their understanding about Social Devices Platform development aspects.

4.1 Understandablity

The team was given a short five minute introduction of the Social Devices concept and platform in the first meeting. All members of the team agreed that the overall idea was easy to grasp. Everyone also agreed the the system was easy to explain for other people as well, for example in their networking pitch. However, the level of understanding seemed to vary between the members. On the one hand, it seems that the project manager and designer did not grasp all of the technical details of the system. On the other, it seems that this kind of technically detailed understanding was not even needed by them, as the concept seems to offer enough higher level abstractions. The developers, on the contrary, agreed that the system was also technically easy to grasp after we gave them a ten-minute presentation of the development aspects and technical details. One thing supports this claim is that the developers managed start implementing the game for Android phone almost immediately after the technical introduction, and on the following day had starting screen implemented. Moreover, the developers didn't have experience on Android development, and it follows that most of their time was spent on studying these things.

Claim 1: Social Devices offers appropriate abstractions for developers.

4.2 Acceptability

When asking from the team about the social acceptability of interactions that start proactively, the members agree that there might be some "shyness" among people, and add that probably people would start using proactive applications

first with their family and friends. Based on their feedback, it seems that games would be an easily acceptable starting point for this type of proactive applications since they are not too serious, and many people also very willingly want to try out new types of games. They also say that applications that automatically and proactively help working in group (e.g. sharing schedules and notes) would be something that they would personally want to use.

The team had gotten enthusiastic feedback from other people while they had been introducing the system for third parties. People had generally liked the idea about using Bluetooth for detecting proximity of others, but some had also been a bit worried about their battery lives. According to the project manager some of his contacts want to hear about how the concept is handled in the future, and gave us contact information of a game studio that had gotten interested in our platform. The overall acceptability of the Social Devices concept is not in the scope of this paper, but instead will be reported separately based on our ongoing research[3].

Claim 2: Social Devices is a socially inspiring concept.

4.3 Coordination aspects

According to the developers it was straight forwarded to coordinate the devices with the help of Social Devices platform and communication with the devices was made very easy. The developers also believe that the this kind of approach to coordinate the devices can be utilized in several other systems, in which a centralized communication point is needed to coordinate devices. However, they point out that a need for fast communication may limit the cases where the current implementation of the system can be utilized. In fact, the latency in communication was one of the reasons why the team chose to create turn–based game.

Currently the latency in device coordination is due to the cloud–based coordination paradigm. To reduce this lag we have considered complementing the system with Personal Network Area based coordination paradigm, which could be implemented on top of Bluetooth protocol for instance. In this paradigm one of the participating devices would be selected as a coordinator, which then would coordinate the other participants as well at itself. However, this would require support for Bluetooth or some other PAN protocol from the device, which all of the Social Devices don't necessarily have.

Claim 3: Social Devices and Action–Oriented Programming Model offer appropriate means for coordinating functionalities in several devices.

4.4 Programmability

Generally the developers liked the methods that Social Devices offers for implementing interactions. The concept of Action was described clear and the de-

[3] `http://www.cs.tut.fi/ihte/projects/CoSMo`

velopers seemed to understand its purpose and features well. Constructing a similar application from scratch would have required concentrating on difficult connectivity and synchronization issues, whereas now, these are hidden by the concepts of the programming model.

The device capabilities also seemed to make sense as a unit of modularity. One of the developers describes them: *"The capabilities are pretty flexible and easy way for anyone to quickly enable games, applications, features etc. from available interfaces. It's like having a little library of things inside an application"*.

Moreover, during their project the developers had couple of ideas how the concept of Action could be complemented. The first idea was that new devices could join in the Action during its execution. As we have had similar ideas, we would have wanted to implement this feature for the platform already during the project. Unfortunately, this wasn't possible due to timing reasons and thus remains as future work. The second idea was to complement the concept so that it would support concurrency inside Action processes: the body part of the Action could then contain multiple threads from which the devices could be coordinated parallel with other threads. The idea is intriguing but still requires further research. The downside of this kind of support for concurrency is that defining device coordination could become more complex.

Claim 4: Social Devices' programming concept action is a clear unit of modularity.

Claim 5: Social Devices' programming concept capability is a "flexible and easy way for anyone to quickly enable games, applications, features".

4.5 Reusability of code

Based on our own experiences and also what can be seen from the team's approach to implement the game, it is hard to design generic capabilities that can be utilized in several different kinds of actions. For example the developed ZombieGame capability can only be utilized in different variants of the current game since so much of the game logic is implemented inside the capability, and not inside the Action. In future we try to research how the capabilities could be designed to be more generic.

4.6 Deployment aspects

In the current implementation of the Social Devices Android client the capabilities can be dynamically loaded during the run time and enabled by the user's choices. The developers regarded the idea of loading the capabilities dynamically very good. However, they eventually encountered difficulties in using this feature which were mainly related to the limitations of Android platform: Android requires defining its Activities in Manifest-file before the application is compiled. This prevents defining and creating new interaction windows from capability code on runtime. Our own experiences are very similar to theirs, and

hence this aspect requires also further research. This feature would also cause problems on other more closed platforms, like iOS and Windows Phone for instance.

The heterogeneity of devices also caused some problems. With cheap low–end devices the lack of memory caused their game to crash at times, but the developers were unanimous that this problem was not related to the Social Devices platform. They had also encountered problems with some 10" Android tablet which they were unable to solve. With other Android tablets and smart phones the game seems to work fine.

5 Related work

Previously, the approaches to coordinate multiple devices have focused mainly on information presentation techniques (e.g., [2,4,5]), and multimedia synchronization (e.g., [6,5]). However, the approach in our work is different as we are not focusing on automating services in pre-defined locations, like in smart spaces for instance, but rather at coordinating devices wherever they are in near proximity of each other. We are also not focusing on generating user interfaces and coordinating them on devices as [2,4]. Instead, our focus is to make the devices interact and socialize independently, and to inform users about the ongoing operations.

6 Open Questions

In this section we raise some open questions of the Social Devices concept and current implementation of the system. The aspects raised here require more research and will remain as future work.

Question 1: How should joint behavior of devices be programmed?

Action has proven to be an excellent abstraction for coercing multi-party applications. However, still some aspects require more studying. How should the triggering condition be described; Is the current precondition decent or should there be a more easily computable function? What about body: Should there be threads? Is there a need for dynamic number of participants? How should the Action be modified to enable it? Does introducing a dynamic number of participants lead to a need for merging two running Actions to one (for example, two zombie games could be merged).

Question 2: What would be a sufficient set of device capabilities for developers?

Capability describes a functionality that a device has. Our goal is to develop universal capabilities which can be utilized in many Actions. What would be a

sufficient set of capabilities for developers? For example, the zombie game has been developed based on a single dedicated capability: ZombieGame. Which universal capabilities could it be based on?

Question 3: How should an automatic triggering of apps be carried out?

Triggering is carried out half manually in the example. User simply pushes a button on her device, then the system uses proximity and state information to find possible participants for the Action. How much of this could be automated? What kind of strategies could be used for retrying to trigger the Action?

Question 4: What are the contexts where the apps can be triggered?

Social Devices applications are meant to be triggered in various kind of social contexts. How to define an appropriate one for each Action to take a place? What kind of different contexts there exists where the Actions could be triggered? How to deduce the context, and how to protect users' privacy?

Question 5: How to evaluate and measure the quality of apps?

The goal of the Social Devices Platform is to offer easy manner for developers to create new applications. On the other hand, the quality of application is very important thing to consider when proactively triggering applications for users. What aspects need to be considered when measuring the quality of the developed SDP applications?

Furthermore, we plan to test the system in a code camp with students, and hence need to set some kind of criteria for evaluating their applications. What aspects needs to be taken into account in this evaluation process? For example, is reusability of the capabilities relevant aspect? Should the amount of features (and the code) implemented on server side (in Action) versus the client side (in capabilities) be taken into account in the evaluation? Or should the applications be evaluated more like regular mobile applications, and try to conceal the distribution and platform specific aspects totally from the developers?

7 Conclusions

Instead of reflecting the interactive capabilities between different devices, the development of mobile apps follows the conventional development fashion. However, there are also different programming models that allow focusing on interactions, as we have demonstrated with our earlier work [3]. In this paper, we listed experiences from developers who were new to our platform to gather feedback on the feasibility of the model as well as the maturity of our platform.

In the future, we plan to execute a more excessive experiment in the form of a one-week code camp with students. The data reported in this paper will be used to improve the platform as well as the instructions that will be given to the participants. In addition, we plan to work on improving the methodology used for the evaluation process.

8 Acknowledgements

We thank the demola team: Trent Pancy, Mikko Järvelä, Teemu Avellan and Sonja-Maria Juslin. The research was funded by Academy of Finland[4] (264422).

References

1. Timo Aaltonen, Varvara Myllärniemi, Mikko Raatikainen, Niko Mäkitalo, and Jari Pääkkö. An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud. In *Asia-Pacific Software Engineering Conference (APSEC) (to appear)*, 2013.
2. Christian Elting. Orchestrating output devices: planning multimedia presentations for home entertainment with ambient intelligence. In *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*, sOc-EUSAI '05, pages 153–158, New York, NY, USA, 2005. ACM.
3. Niko Mäkitalo, Jari Pääkkö, Mikko Raatikainen, Varvana Myllärniemi, Timo Aaltonen, Tapani Leppänen, Tomi Männistö, and Tommi Mikkonen. Social Devices: Collaborative Co-Located Interactions in a Mobile Cloud. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, MUM '12, pages 10:1–10:10, New York, NY, USA, 2012. ACM.
4. Brad A. Myers, Jeffrey Nichols, Jacob O. Wobbrock, and Robert C. Miller. Taking handheld devices to the next level. *Computer*, 37(12):36–43, December 2004.
5. Jun Rekimoto. Multiple-computer user interfaces: "beyond the desktop" direct manipulation environments. In *CHI '00 extended abstracts on Human factors in computing systems*, CHI EA '00, pages 6–7, New York, NY, USA, 2000. ACM.
6. Bo Xing, Karim Seada, and Nalini Venkatasubramanian. Proximiter: Enabling mobile proximity-based content sharing on portable devices. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, PERCOM '09, pages 1–3, Washington, DC, USA, 2009. IEEE Computer Society.

[4] `http://www.aka.fi/en-GB/A/`

VI

Publication VI

M. Raatikainen, T. Mikkonen, V. Myllärniemi, N. Mäkitalo, T. Männisto, and J. Savolainen. Mobile Content as a Service A Blueprint for a Vendor-Neutral Cloud of Mobile Devices, In *IEEE Software*, Volume 29, Issue 4, pages 28 – 32, July, 2012, IEEE Computer Society.

VII

Publication VII

N. Mäkitalo, H. Peltola, J. Salo, and T. Turto. VisualREST: A Content Management System for Cloud Computing Environment, In *Proceedings of 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'11)*, pages 183 – 187. Oulu, Finland, August 30 – September 2, 2011, IEEE Computer Society.

VisualREST: A Content Management System for Cloud Computing Environment

Niko Mäkitalo, Heikki Peltola, Joonas Salo and Tuomas Turto
Department of Software Systems
Tampere University of Technology
P.O.Box 553, FI-33101 Tampere Finland,
Email: {niko.makitalo, heikki.peltola, joonas.salo, tuomas.turto}@tut.fi

***Abstract*—The amount of digital content is increasing at an accelerating speed. This content is distributed to a heterogeneous set of users' devices and handling the content manually is an unsustainable solution and soon leads to a increasing problem. We have studied the most crucial concepts of content management systems and how the distributed environment affects them. Moreover, we have investigated technologies that can be used to implement a content management system in a distributed environment. Based on these, this paper introduces the VisualREST content management system that provides a uniform way to manage the content stored in all of the devices with the same user–friendly and efficient principles. As with other cloud computing systems, also VisualREST abstracts away the physical structure and complicated processes from user perspective.**

***Keywords*-Cloud Computing; Mobile Devices; CMS; REST;**

I. Introduction

The amount of digital content is increasing as more new ways to produce digital content become available. Managing the content manually is unsustainable because of its challenging and time-consuming nature. In addition, the memory and network bandwidth of the users' devices is often inefficiently used.

In this context, the users' devices form a distributed environment where the different devices form networks. For instance, the computers may be multiprocessor computers that are capable of heavy processing or, on the other hand, lightweight mobile devices that have very limited battery life and constrained resources. The most important common feature is the ability to connect with other computers in the system.

We have studied how content can be managed in a heterogeneous and distributed environment. On the basis of this research, we have summarized the most crucial features and concepts that are required of a content management system. The impact that a distributed and heterogeneous environment causes for the content management is especially surveyed. We also have studied what technologies can be used for developing content management system in this specific environment.

This paper introduces a new content management system named VisualREST [1]. The system is designed and implemented with the special characteristics of the distributed and heterogeneous environment in mind. The main goal of the system is to provide a uniform way for users to manage the content stored in all of their devices with the same user–friendly and efficient principles. As cloud computing and distributed systems in general, also VisualREST tries to abstract away the physical structure and complicated processes from the user perspective.

The rest of the paper is organized as follows. Section II outlines the most crucial features of content management. Section III reviews the backgrounds of the most relevant technologies that have been used for the implementation. Section IV gives an overview about VisualREST content management system. Section V introduces the interfaces of the VisualREST from the content management perspective. Section VI evaluates the system and Section VII concludes the paper.

II. Motivations

Managing digital content efficiently usually requires special solutions which depend on the way that the content is used. Moreover, the nature of the content as well as the environment of use are related to the way that the content needs to be managed. Because of the complex and multidimensional nature of managing content, no general guidelines exist and the terminology varies with different sources. The following discussion introduces the terminology of content management that we use and the main features of content management in a distributed heterogeneous environment.

A. Content: Metadata and Essence

According to Mauthe and Thomas [2, p. 4] content can be any kind of audiovisual, visual, sound, or textual information. However, the term content itself is used with different connotations and can therefore be confusing. We use terminology defined by Society of Motion Picture and Television Engineers (SMPTE) and the European Broadcasting Union (EBU). According to this terminology content consists of the metadata and the essence [3].

1) Essence: The core part of the content. Mauthe and Thomas describe essence as: "the raw programme material itself, represented by pictures, sound, text, video, etc." [2, p. 4].

2) Metadata: Descriptive information about the essence and its representations. Metadata can be related to content, location or media/material [2, p. 4].

B. Main features of content management

Mauthe and Thomas define that a system which comprehensively manages both essence and metadata is called a content management system [2, p. 5]. Our previous research [1] has indicated that the following are the most crucial features of content management systems (CMS features):

1) From the end user and content interaction perspective, the content needs to be stored to a location that is safe and where it can be accessed regardless of time or place [4]. Content management system needs to take care of the whole life cycle of the content [5].
2) Searching for the content and navigating the results should be easy and intuitive [5], [4].
3) A distributed content management system needs to monitor where content is referenced and furthermore, place the content in the right physical location according to this information [6].
4) As much metadata as possible needs to be extracted and produced from the content and attached to it right after the content has been created [5].
5) The system needs to support describing the content with heterogeneous and arbitrary metadata [7].
6) In a heterogeneous environment, content management system needs to offer a uniform interface for all the resources [6].
7) A distributed content management system in a heterogeneous network needs to offer an efficient way to access content with user specific configurations and settings [6].

These content management features have also been discussed in more details in [1].

III. Technical background

In this section we introduce the major concepts behind VisualREST. We start with REST and then discuss XMPP.

A. Web and REST

REST, as described by Fielding in his thesis [8], is a general architectural style for building network-based software. Most often, however, REST is utilized in the context of web applications. In this context REST becomes a resource-oriented architecture where resources are exposed by servers and consumed by the clients using HTTP methods [8]. A resource is accessed via its URL and its state is transferred using its representation. A key characteristic of a RESTful interface is the clear division of application state between the client and the server.

Three major properties of REST are the use of layered client-server style, the uniform interface and the possibility to negotiate a suitable representation for a resource. The first property allows the introduction of intermediaries such as caches and gateways. The second permits a wide range of clients to utilize the system. Finally the negotiable representations allow humans and programmed clients to simultaneously use the system. This architecture is, however, distinctly based on shared data repository. This means that the client-server relationship is enforced and the web service just acts as a mediator for the data.

REST, as realized by the HTTP protocol, has the pull style built in into the architecture. For modern web applications, many authors consider this to be the major drawback of REST. It should be noted, however, that this is by design as discussed in [8]. It is argued that although event-based integration style would be beneficial for a client wishing to monitor a single resource, such a push-based model would be too unwieldy considering the scale of the Web.

B. XMPP

The Extensible Messaging and Presence Protocol (XMPP) is a protocol designed for near-real-time and request-response services streaming XML messages [9]. XMPP and its extensions provide support for establishing presence, authentication, one-to-one and multi-party messages and notifications. These services have been used to build systems for instant messaging, systems control, and lightweight middleware [10].

The streaming of XML messages is achieved using long-lived bi-directional TCP connections and the messaging elements are defined as an XML dialect. The XMPP is built on a client-server architecture where the clients connect to their respective provider and these in turn communicate with each other in order to provide the requested service, such as instant messaging. The messages can be bi-directional and be stored in a message bus. The bi-directionality makes it possible to treat each XMPP client as a first-class citizen in the architecture. In addition, XMPP allows messaging over federation barriers thus enabling message-passing style over organizational boundaries.

Although the semantics of the long-lived bi-directional TCP connections that XMPP utilizes can be emulated with HTTP connections [11], XMPP is mainly utilized outside the Web context. Here the long-lived TCP connections enable the possibility of nearly real-time communication between the clients without the need for polling. Thus services built on the XMPP stack are a good fit whenever we wish to send notifications of a changed state or monitor a particular set of entities.

IV. VisualREST

In the following we give a short overview of the VisualREST content management system and introduce the different roles of the clients. The overview of the system is illustrated in Figure 1.

A. Overview

VisualREST [1] is the content management system that facilitates users' access to the essence and the metadata of their content items that are distributed across many devices.

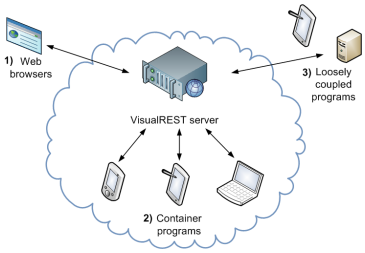


Figure 1. Overview.

The users can register their desktop, laptop and mobile devices to VisualREST and access their content from a single entry point, the VisualREST server. The server provides a web interface and Atom Feeds [12] for viewing the content. It also keeps track of different versions of the content. New versions are automatically created when content is updated, so content is never lost in VisualREST.

VisualREST's architecture is roughly based on the client-server model. There is one server that coordinates the connections and processes with the client devices. The actual communication between the server and the client is essentially client-server based (HTTP).

B. Roles

The VisualREST server's role is to act as a middleman between the clients, storing the metadata of all the content in one shared repository. Clients of the VisualREST can play three different roles depending on the task they wish to achieve. Furthermore, a client can play multiple roles at the same time, as long as it fulfills the requirements of each role it takes. These roles are (Figure 1):

1) User using the service with a browser: The role can be used, for instance, to manage the profile, for editing the access rights, and for browsing and accessing the content.

2) Container program providing content to the system: The container programs are used for importing content to VisualREST and upload the essence to the server.

3) A loosely coupled program using the REST interface: Developers can create new clients for different needs. These clients are used for managing the resources by using CRUD operations and visualizing the representations of the content from VisualREST. For instance, it is possible to add new arbitrary metadata to content through the REST interface.

C. Container program

An instance of the container program runs on user's device. Its main task is to import content to VisualREST. The program monitors the changes in the content, and whenever these changes do occur, it versions the content and extracts the initial metadata, which is then sent to the VisualREST server: In VisualREST, each content item has the initial metadata, which is a set of metadata that defines the existence of a content item. Moreover, users can use additional metadata for describing a content item.

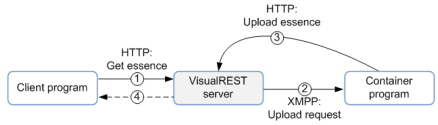


Figure 2. Requesting content from the server.

Compared to other clients, the container program's role is more tightly coupled to the server, because they are used for storing the essence of the content, and then transferring it to the server when explicitly asked to. On the interface level this means that, in addition REST interface, the container program also needs to implement the message passing functionality.

V. Application Interfaces

This section describes the two interfaces that VisualREST offers for applications. First we introduce the REST interface, followed by the complemented interface for more tightly coupled client programs.

A. REST interface

Basic features such as creating and modifying resources like content, users, user groups etc. are implemented in a RESTful manner using CRUD operations. The server also enforces access rights when required. The following summarizes the REST interface operations that are the most crucial from content management perspective.

1) Importing content to VisualREST: The initial metadata of a content item is transferred to the server with HTTP PUT and POST methods.

2) Transferring essence of content: The essence of a content item is transferred from container program to the server with HTTP POST method.

3) Adding and modifying arbitrary metadata: Arbitrary metadata associated with the managed content is added and modified with HTTP PUT, POST and DELETE methods. Arbitrary metadata can be added and modified by container programs, as well as by any other type of client program.

4) Searching for content: Content is searched using queries with search parameters. These parameters, which can be related to any metadata that has been added, are attached to the query part of URI address. The path part of URI defines a context. The context can be seen as a filter, which narrows down the query to a more specific set of content. The searching of the content is done with the HTTP GET method.

5) Fetching metadata and essence: The metadata and essence of a content is fetched with the HTTP GET method.

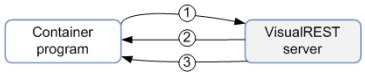


Figure 3. Transferring content list.

B. Complemented interface

The following functionality could not be done efficiently or without breaking the REST architectural style, and therefore the architecture has been complemented with the message passing architectural style, implemented in XMPP.

1) Uploading essence of content: Figure 2 illustrates the upload process. When a client program requests essence of content from the server (step 1), access rights to the content are checked. If the client is authorized to access the content, the server requests the essence from a container program (step 2). After the essence has been uploaded to the server (step 3), it can be returned to the client that requested it (step 4). Because of this convention, the client doesn't need to know where the essence is really located at.

2) Generating thumbnails of pictures: The server can request thumbnails from the container program with a XMPP message. The thumbnails are sent to the server using the REST interface after they have been generated by the container program. This saves bandwidth as the possibly large essence is not transferred.

3) Parsing the content list: The container program keeps track of its content items. Whenever content is modified or added, the container program sends the content list to the server. Figure 3 illustrates the steps done in transferring the content list to the server: As the container program sends the content list (step 1), it receives a HTTP response implying that the process of parsing the content list has been started (step 2). When the process is ready the server sends an XMPP message to the container program notifying of the success or the failure of this process (step 3).

VI. Lessons learned

In this section we evaluate the current implementation of VisualREST. First we discuss how VisualREST meets the content management features that were presented in Section II. Secondly, we evaluate how well the used technologies serve their purpose.

A. Content management features of VisualREST

The three most important concepts of content management are searching, describing and ensuring the availability of the content [2]. Next we discuss more closely how VisualREST meets these concepts.

1) Search features: In order to meet CMS feature 2, VisualREST offers an intuitive and efficient REST interface that both users and other applications can use. According to the REST guidelines and CMS feature 6, the interface needs to offer unique identifiers and a uniform interface for each resource. VisualREST uses URIs for this purpose. The URIs form a hierarchy with levels that can be used as a context for handling sets of resources. In queries these contexts form basis for the search, which narrows down the search results.

VisualREST can be accessed directly by the end user as it provides a graphical user interface for web browsers. In addition VisualREST can serve the resources in structured data formats such as Atom feeds, which can be viewed by web browsers, feed readers, and most importantly, are easy to parse with XML libraries. Custom client programs can be built to visualize the resources in use-case specific ways.

2) Describing content: According to CMS feature 4 as much metadata as possible needs to be extracted and produced from the content right after it has been created. Curtis et al. have solved this with a separate tool that can generate metadata from a content [5]. In VisualREST the container program, as described above, uses somewhat the same principles: after noticing that new content has been produced, container program versions it, extracts the initial metadata and sends it to VisualREST server.

Currently, however, the metadata that is generated by automated processes is not very beneficial for users searching for a specific content [5]. CMS feature 5 requires that content should be describable with heterogeneous and arbitrary metadata. In VisualREST, the users can add their own metadata types and use the metadata types added by other users to describe the content. These metadata types also have basic comparison operators that can be used to define value ranges.

3) Availability of content: In order to meet CMS features 1 and 3, the content in VisualREST has been divided into two parts. The metadata is always available from the VisualREST server for authorized users. The essence, on the other hand, is stored in the container programs.

This arrangement can be beneficial in a world of slow, limited or expensive mobile Internet connections, because uploading essence unnecessarily to the server can often be avoided. For instance, the content is quite often modified on the device that has produced it, but other users, however, are usually only interested in the final version. On the other hand the critical essence from a content–user interaction perspective can always be transferred to server right after creation (CMS feature 1).

The role of the VisualREST server in this scenario is to transmit essence upload requests and then convey the essence that is uploaded by the container programs. In addition, the server observes the online status of the devices and by using this information gives clear responses to the users. Compared to the Web in general, one of the main goals of content management systems is to provide clear responses and explanations in cases when content is not available.

The availability of the content is a multidimensional problem, and meeting the CMS features 1 & 3 depends on many different concerns such as the networking environ-

ment, the type of the device, and the users' usage habits. The availability in a distributed heterogeneous environment is a compromise among these concerns. In the future we plan to study how the essence could be transferred directly between devices.

B. Technology perspective

The REST guidelines offer an intuitive and efficient way for managing and searching content. REST guidelines can be used for the design and the implementation of the interface for content and all the other resources of the system. User-friendly addresses also help referencing resources with the same intuitive principles.

The REST guidelines seem to fit well for content management purposes. However, in cases where the REST architectural style cannot offer native support for implementing a desired feature, the architecture has been complemented with the message passing architectural style.

During the evaluation process, minor conflicts with the REST guidelines emerged: The REST guidelines forbid the server to maintain the client's status. VisualREST server, however, needs to maintain the online status of the container programs in order to give proper information about the availability of the content. Currently, the container program updates through the REST interface a timestamp describing the online status of the container resource.

VII. Conclusion

Content management in a distributed heterogeneous environment is a currently relevant problem. The amount of devices capable of producing and storing large amounts of digital content is continuously growing. Moreover, users are willing to share their content among other users.

We have studied the most crucial requirements of content management systems. On the basis of our research, we collected a set of features that a content management system should meet in a distributed heterogeneous environment.

In this paper we introduced the VisualREST content management system that aims to offer a uniform way for the end users and the client programs to manage content with the same user-friendly and efficient principles. REST was a successful design choice for our approach as it made the interfaces clear and intuitive. With a RESTful interface we could also abstract away the physical structure and homogenizing the environment.

In the evaluation process VisualREST was discovered to meet the content management features well. Also the chosen technologies fit to their purposes mainly well. REST does, however, have limitations that have been also visible in our approach. Whenever we needed features that REST does not natively support, we complemented the architecture in a way that was not against the REST guidelines. A message passing architecture implemented in XMPP has been one of these technologies and in the future we will study how to take even more advantage of this complementary architecture.

VIII. Acknowledgment

We would like to thank Subhamoy Ghosh, Mikko Ylikangas, Mikko Raatikainen and Tomi Männistö from Aalto University, Helsinki, Finland, and also Tapani Leppänen, Juha Savolainen, Timo Aaltonen, Vlad Stirbu and Rod Walsh from Nokia Research Centrer, Tampere, Finland, for providing key inputs for the contributions in this paper. This research was part of the Cloud Software Program of TIVIT, and was partially funded by TEKES.

References

[1] N. Mäkitalo, "RESTful content management system for distributed environment," Master of Science thesis, Tampere University of Technology, 2011.

[2] A. Mauthe and P. Thomas, *Professional content management systems: handling digital media assets*. West Sussex, England: John Wiley & Sons, 2004.

[3] SMPTE/EBU, "Task Force for Harmonized Standards for the Exchange of Programme Material as Bitstreams–Final Report: Analyses and Results," 1998.

[4] A. Spedalieri, A. Sinfreu, G. Sisto, P. Cesar, I. Vaishnavi, and D. Melpignano, "Multimedia content management support in next generation service platforms," in *Proceedings of the 3rd international conference on Mobile multimedia communications*, ser. MobiMedia '07. ICST, Brussels, Belgium: ICST, 2007, pp. 14:1–14:7.

[5] K. Curtis, W. P. Foster, and F. Stentiford, "Metadata – the key to content management services," in *Proceedings of the Meta-Data'99*, ser. The Third IEEE Meta-Data Conference, Bethesa, Maryland, USA, 1999.

[6] C. D. Cranor, R. Ethington, A. Sehgal, D. Shur, C. Sreenan, and J. E. van der Merwe, "Design and implementation of a distributed content management system," in *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, ser. NOSSDAV '03. New York, NY, USA: ACM, 2003, pp. 4–11.

[7] P. Bolettieri, F. Falchi, C. Gennaro, and F. Rabitti, "Automatic metadata extraction and indexing for reusing e-learning multimedia objects," in *Workshop on multimedia information retrieval on The many faces of multimedia semantics*, ser. MS '07. New York, NY, USA: ACM, 2007, pp. 21–28.

[8] R. T. Fielding, "REST: architectural styles and the design of network-based software architectures," Doctoral dissertation, University of California, Irvine, 2000.

[9] *RFC 3920 - Extensible Messaging and Presence Protocol (XMPP)*, Jabber Software Foundation, October 2004.

[10] P. Saint-Andre, K. Smith, and R. TronCon, *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*, 1st ed. Sebastopol, CA, US: O'Reilly Media, 2009.

[11] I. Paterson and P. Saint-Andre, "XEP-0206: XMPP over BOSH," XMPP Standards Foundation.

[12] *RFC 5023 – The Atom Publishing Protocol*, NewBay Software, Google, October 2007.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3749-3
ISSN 1459-2045