

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

**MODELOS EMERGENTES PARA EL DESARROLLO DE
APLICACIONES MÓVILES SOCIALES: PEOPLE AS A
SERVICE Y SOCIAL DEVICES. PRUEBA DE CONCEPTO.**

**EMERGING MODELS FOR THE DEVELOPMENT OF SOCIAL
MOBILE APPLICATIONS: PEOPLE AS A SERVICE, AND
SOCIAL DEVICES. A PROOF OF CONCEPT.**

Realizado por
JOSÉ ANDRÉS CORDERO BENÍTEZ
Tutorizado por
CARLOS CANAL VELASCO
Departamento
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA
MÁLAGA, noviembre 2014

Fecha defensa:
El Secretario del Tribunal

Resumen: Actualmente los dispositivos móviles han ganado una gran importancia en nuestra sociedad y están presentes en cualquier lugar. Estos dispositivos tienen una gran cantidad de información sobre sus dueños y datos contextuales de sus alrededores, y muchos canales de comunicación. No obstante con los modelos actuales los dispositivos móviles no pueden obtener ventaja de esta situación.

Por estos motivos, han comenzado a surgir recientemente nuevos modelos de aplicaciones móviles. People-as-a-Service (PeaaS) permite interacciones entre dispositivos móviles cercanos, usando perfiles sociológicos de sus dueños para crear comunicaciones entre dispositivos interesados. Social Devices es un modelo reciente que implementa el concepto de PeaaS, aprovechando que los dispositivos están casi siempre con el usuario crea una nueva línea de interacciones entre los dispositivos cercanos con el objetivo de aumentar las relaciones sociales. Para ello cada dispositivo es capaz de proveer y consumir servicios de dispositivos cercanos, manteniendo la información del usuario en el propio dispositivo y compartiéndola solamente cuando esté autorizado. Este concepto también puede integrarse con tecnologías como Internet-de-las-Cosas y la Computación Ubicua, permitiendo que los dispositivos se conecten con otros objetos cercanos tales como ascensores, semáforos o tu propio vehículo.

En particular existe una plataforma que implementa ambos conceptos, y que es el objeto de estudio de este trabajo: OrchestratorJS. La plataforma aún se encuentra en desarrollo y no es demasiado estable, además de no contar con ninguna documentación. El objetivo de este trabajo es explorar la plataforma y crear una documentación básica que pueda ser utilizada por cualquiera que esté interesado en ella, actuando como punto de partida para futuros trabajos.

Palabras clave: Social Devices, People-as-a-Service, Cooperación basada en proximidad, Computación Ubicua, OrchestratorJS.

Abstract: Nowadays mobile devices have gained a great importance in our society and they are present everywhere. These devices have a huge amount of data about their owners and context data of their surroundings, and many communication channels. Nevertheless with current models mobile devices cannot take advantage of this situation.

For these reasons, new models for mobile applications have appeared recently. People-as-a-Service (PeaaS) allows interactions between mobile devices in their surroundings, using sociological profiles of their owners to create communication between interested devices. Social Devices is a recent model that implements the PeaaS concept, taking advantage the devices are almost always with the user it creates a new kind of interactions between nearby devices with the main target of improve social relations. For this purpose each device is able to provide services to nearby devices and consume them, keeping the user's data in the device and sharing it only when it is authorized. This concept could be integrated with technologies like Internet-of-Things and Ubiquitous Computing, allowing devices to connect with nearby objects like elevators, traffic lights or your own vehicle.

In particular, there is a new platform that implements both concepts, which is the study object of this work: OrchestratorJS. The platform is still under development and is not too stable yet, furthermore there is no documentation available. The main target of this work is to explore the platform and to create a basic documentation which could be used for anybody interested on the in, acting as a starting point for future works.

Keywords: Social Devices, People as a Service, Proximity-based Cooperation, Ubiquitous Computing, OrchestratorJS.

Index

INDEX	7
INTRODUCTION AND GOALS	9
CHAPTER 1: PEOPLE-AS-A-SERVICE	13
CLOUD-ENABLED MOBILE DEVICES	13
PEOPLE-AS-A-SERVICE	15
PEAAS ARCHITECTURE	16
CURRENT IMPLEMENTATIONS	18
THE INTERNET OF PEOPLE.....	22
INTEGRATION WITH SOCIAL DEVICES AND INTERNET OF THINGS	26
CHAPTER 2: SOCIAL DEVICES	31
THE ACTION-ORIENTED PROGRAMMING MODEL	32
REQUIREMENTS AND CHARACTERISTICS OF SOCIAL DEVICES	37
ARCHITECTURE	39
ORCHESTRATOR.JS.....	40
TOOLS.....	41
GETTING THINGS WORK	49
ANDROID CLIENT.....	52
CHANGES AND ADDITIONAL FEATURES	54
GOALS.....	56
EXAMPLE: THE COFFEE BREAK MACHINE	57
CHAPTER 3: RELATED TECHNOLOGIES	61
ANDROID PROGRAMMING	61
MONGODB	67
NODE.JS.....	70
CHAPTER 4: GETTING STARTED WITH SOCIAL DEVICES	75
DEPENDENCIES.....	75
INSTALLATION	76
GETTING STARTED	79
EXECUTING A FIRST ACTION	82
STARTING THE DEVELOPMENT.....	83
CHAPTER 5: APPLICATION HITHERE!	89
CHAPTER 6: APPLICATION EGREETINGS	97
CHAPTER 7: APPLICATION FOLLOWME	103
CHAPTER 8: APPLICATION HAVEYOU MET	109
CHAPTER 9: APPLICATION TREASUREHUNT	115
CONCLUSIONS AND FUTURE WORK	119
CONCLUSIONES Y TRABAJO FUTURO	121
ANNEX 1: NPM MODULES	123
REFERENCES	127

Introduction and goals

Nowadays smart devices have an excellent potential and connectivity, and can be used for multiple purposes: entertaining, socializing or sharing information. We have applications capable to do everything with the help of web services and cloud computing. However, we have currently some limitations.

These days everything is *something-as-a-service*. We are used to Software-as-a-Service and Platform-as-a-Service that have been well accepted by the industry. Now, when the usage of mobile devices have become customary for the majority of population, probably it is the moment of a new model. Mobile devices are becoming the most important piece in the communications, and they contain a lot of information about the owner. We could use this advantage to create a new type of communication, where devices interact with other devices as people would do.

The goal of this work is to learn about a new model, called People-as-a-Service [18][19], where mobile devices can deploy services to interact with other devices in their surroundings. In particular, this work studies the platform called *Social Devices*, developed as a research work by the University of Tampere in conjunction with Nokia [26]. This platform is currently in development and there is no documentation or further information available.

The main target of this work is to analyse the potential of the platform *Social Devices*, to study all the capabilities and create a base documentation about it that could be used for future works. I have created some case studies to study the potential and capabilities of the platform, which are completely documented to prepare the way for future projects. All the effort has been concentrated in researching and understanding the platform, documenting everything properly to facilitate the work for future projects using this platform and technology.

The platform is not finished and there is a long way to integrate everything, I have been collaborating with Niko Mäkitalo and his team in the University of Tampere improving some aspects of the platform and the Android client. The code is public to everyone and can be accessed in the next link:

<https://github.com/nikkis>

This work uses NodeJS¹ and Python² as basic technologies to develop new capabilities and functionality in the platform. The server service, called OrchestratorJS is currently developed in NodeJS, and the capabilities in Python. On the other hand, the client is focused in mobile devices, so we use Java as main technology to develop the client functionality for Android devices.

¹ <http://nodejs.org/>

² <https://www.python.org/>

This document is divided into three main parts:

- The first part includes the chapters from 1 to 3, which are a brief introduction to the technologies used in this work: *Cloud Computing*¹, specially the model *People-as-a-Service* and the *Action-Oriented Programming Model* [1], which are used as basic architecture for the Social Devices platform. The server is developed using Node.js [35], and connected to a MongoDB [32] Database. *Android Programming* [4] is the technology used in the client.
- The second part contains all the relevant information to understand the Social Devices Platform and OrchestratorJS. I explain in Chapter 4 how the platforms behaves, their architecture and the base of knowledge. Also I explain how to prepare and install the environment to start the development.
- The third part includes the chapters from 5 to 9, one chapter for each case study. I have developed 5 small applications that can be used as a proof of concept to see what the platform can offer. Each chapter contains all the documentation to use the application and details about its implementation.

Social Devices has currently two functional platforms implementing the concept. In chapter 4 we will see the first implementation, called *Social Devices Platform* (SDP) as an introduction to the new platform *Orchestrator.js* (OJS) [25].

¹ http://en.wikipedia.org/wiki/Cloud_computing

Motivation

This project is aligned with the strategic objectives of the research groups SCENIC, from the University of Málaga, and QUERCUS, from Extremadura on Mobile Cloud Computing and related with the research projects *SeaClouds*¹ and SOFIA², in which is currently working the University of Málaga as many other universities across Europe. These research groups are focused in cloud technologies and mobile computing.

Cloud Computing is a new concept and there is a long future for it. It reduces time-to-market and provides on-demand scalability at a low cost to the business and a new concept is being developed: People-as-a-Service.

PeaaS is a new technology in development and there are only a few platforms that integrates the concept. The purpose of this project is to create a proof of concept of what People-as-a-Service can bring to us, focused in the platform Social Devices, currently in development as a PhD, by Niko Mäkitalo in the University of Tampere.

The target of this project is to do the proper investigation into the platform and elaborate a well know-how about it, exploring the possibilities and preparing all the documentation for further research and developments, as there is no documentation about it. The platform is currently in development and is not completely stable.

The concept of PeaaS will be also interesting for people, because this new technology will allow new ways of interacting between people and devices, providing new capabilities in how we use them. Combined with Internet of Things, people will be able to interact with the surroundings, the devices will know the context data of the user, and change its behaviour depending on it, making the work for the user without forcing him to send commands to the device.

Social Devices is a new platform that can be used for this purpose, it was introduced in 2011 as a joint work by Nokia Research Center, Aalto Univesity, and Tampere University of technology [26]. Smartphones have not only a lot of information about their owners, but also capabilities that enable them to resemble humans (they can translate text into speech, for example).

¹ <http://www.seaclouds-project.eu/>

² <http://sofia.lcc.uma.es/>

Chapter 1

People-as-a-Service

The cloud computing paradigm has not been well adopted by the industry, overthrowing all the initial expectations. Nowadays there are no many big or complex services developed to the cloud, and migration is not considered yet as an option by many organizations [18].

However cloud computing has become a real success for small devices, like mobile phones and tablets. Currently there are many applications with back-end components running in the cloud which serves them. Applications intended to be used by mobile devices but too complex to be executed on it, with the low throughput and resource limitations of this kind of devices.

The cloud-based model also solves the data ubiquity problem [2], offering mechanisms for scaling and adapting computing resources depending on the user's demand, and allowing transparent syncing capabilities for access to the same information anywhere from any device.

Currently when we speak about mobile devices and cloud computing we think in the mobile device connecting to the cloud to consume a service. However, taking into account the new emerging technologies we have to change this point of view, and start thinking in the mobile device as service providers.

Due to the mobile devices' level of incursion in society we can treat them as a digital mirror of its owner identity. From a social perspective, every device contains a huge amount of information that can be used with other people.

Cloud-Enabled Mobile Devices

Mobile devices conform a complex network of distributed nodes with high processing capabilities, where each of them can collect information from its sensors about the surrounding environment.

In this context, mobile devices as a whole could be considered as a 'big cloud' [18] environment of computational and information resources. The concept of *cloud-enabled mobile devices* emerges from those factors, and it can be summarized with the following ideas:

- **Mobile devices can act as service providers.** Instead of considering mobile devices as dumb terminals for consuming services, the cloud-enabled mobile devices concept implies the service provider role of such devices. Using technologies based on the most common web-services protocols like SOAP or REST for providing standard-based service interfaces.

- **Services deployed in mobile devices consume available resources and information.** The deployment and execution of services in cloud-enabled mobile devices treat the information and available mobile resources as sources of data that could be consumed and conveniently served.
- **Resources, information and patterns of use of mobile devices can be understood as the digital identity of their owners.** The way in which people use their mobile phones defines accurately their sociological profile. These devices are increasingly used for reading and answering emails, taking photos, messaging, interacting in social networks, or browsing the web. How, when, and where is all this activity done is logged in the device, and such information, that represents the virtual behaviour of each owner, could be exploited by new social-oriented mobile applications and services.

Having all the above in mind, Cloud-Enabled Mobile Devices (CEMD) could be defined as a conceptual perspective of mobile computing that highlights smartphones as the digital extension of their owners, and as unique devices able to depict the sociological profiles of their users, capable enough to provide and consume services and resources between them in a not necessarily centralized way of connection.

Alternative to Client/Server Mobile Architectures

Traditionally mobile applications are held by a full back-end service running in a server. However, current mobile performance and energy management allows us to develop applications which can dispense with the server part that makes the heavy-load work. We can start to think in alternatives like Mobile P2P, where applications connect with other devices instead of a centralized server.

Mobile Peer-to-Peer architectures perfectly fit with the proposed support environment for the cloud-enabled mobile devices concept. Such architecture model and its communication protocols could be enriched to include extra properties or attributes in node descriptions related with the nature of the services which are exposed to each one (Guillén et al., *Architecting Infrastructures for Cloud-Enabled*, 2013).

Supporting the Distributed Processing of Individual Social Profiles

The cloud-enabled mobile devices concept needs a way to communicate devices with each other, sending and retrieving data from others. They must be able provide and consume services from other devices.

However, the personal information must be communicated in a secure and trusted environment. The cloud-enabled mobile devices concept implies the privacy-friendly treatment of the personal information, encouraging a service model based on the distributed processing of the information, avoiding its centralized storage.

The information gathered by the different social networks about their users can be unified in a unique profile, which can be used as the key and the visible face of their

owners from the other user's perspective. Mobile devices gather enough information about their owners to become as the digital representation of them in the virtual society.

People-as-a-service

People-as-a-Service [19] (PeaaS) is a new service model built on a cloud-inspired infrastructure that considers each mobile device as a provider, which can be used to deploy, instantiate and execute services.

The concept of People-as-a-Service is new and not totally developed yet. This model is the result of the joined work of the QUERCUS¹ group from the University of Extremadura (UNEX) and the SCENIC (Service and Component Engineering for Internet Computing) group from the University of Málaga (UMA), which the collaboration of *Gloin*, a company settled in Cáceres who has developed applications like *BeeFun*², *nimBees*³ and focused in Cloud Computing.

These devices can be used to generate a sociological profile of its owners, which can be shared and securely provided to third parties as a service. The access to the information is controlled, so owners can contribute to collective sociological information without risk. This profiles can be of great interest and benefit to researchers and fields like smart cities. The People-as-a-Service (PeaaS) model establishes the evolution of this kind of technologies.

The target of the PeaaS model is to exploit the potential of mobile devices to offer sociological profiles and offer them as a service. The user's identity is stored in the device, under the control of the owner, but allowing the external access to consult the information.

PeaaS rests on these four pillars [19], which try to overcome the limitations of current social computing models:

- **Mobile Devices as Interfaces to People:** Mobile devices are considered as interfaces to people, connecting people to the outside world. These devices are the main component of PeaaS.
- **Sociological Virtual Profiles:** Mobile devices can collect information about their surroundings. This means the device has information about the behaviour of the owner and can be used to create a sociological virtual profiles of the owner.
- **Sociological Profiles as a Service:** These owner's profiles are very useful to external entities. Sociological profiles must be provided as a service to those who might wish to access that information, allowing interactions between these profiles.

¹ <http://www.unex.es/investigacion/grupos/quercus>

² <http://www.beefunapp.com/>

³ <http://nimbees.com/>

- **User privacy:** One of the most important points of this model is to guarantee that the individual profile is always kept exclusively in the owner's device, not in centralized servers that often violate privacy. Furthermore, PeaaS makes all accesses of information completely visible and controllable by the user.

The user can know who and when accesses the information and what information is read. The privacy policies can be set up according to their preferences.

The information provided to external entities must be completely anonymous, so others can't identify the device's owner. Personal information about the owner will be only given under explicit permission by the owner and only in specific circumstances.

The model has the following capabilities to be able to comply the four pillars and provide support for the services:

- **Execution environment for services:** PeaaS-based mobile services are deployed and run in a common execution environment that is platform independent. This environment provides access to the mobile device's resources, like network.
- **Networks of mobile devices:** The PeaaS model provide mechanisms for implementing private or public networks of mobile devices.
- **Discovery service:** Mechanisms for discovering other connected devices in the surroundings.
- **Device permission control:** The service execution environment allows device owners to define access policies to the device's resources for each of the deployed services.
- **Device-to-device communications:** The model allows peer-to-peer communication between services deployed in different mobile devices.
- **Crowd-oriented services:** Executing crowd-oriented services, like public opinion research surveys, is another possibility of the PeaaS model.

PeaaS Architecture

This abstract architecture [Figure 1] assures that PeaaS implementations comply with the four pillars, it can be profiled to match any special requirement to any PeaaS application.

The central component in this architecture is the mobile device, which contains the user's profile and all the mechanisms to provide the service to other devices and consume them. The architecture is implemented at the application level, but could be more efficient closer to the OS and hardware.

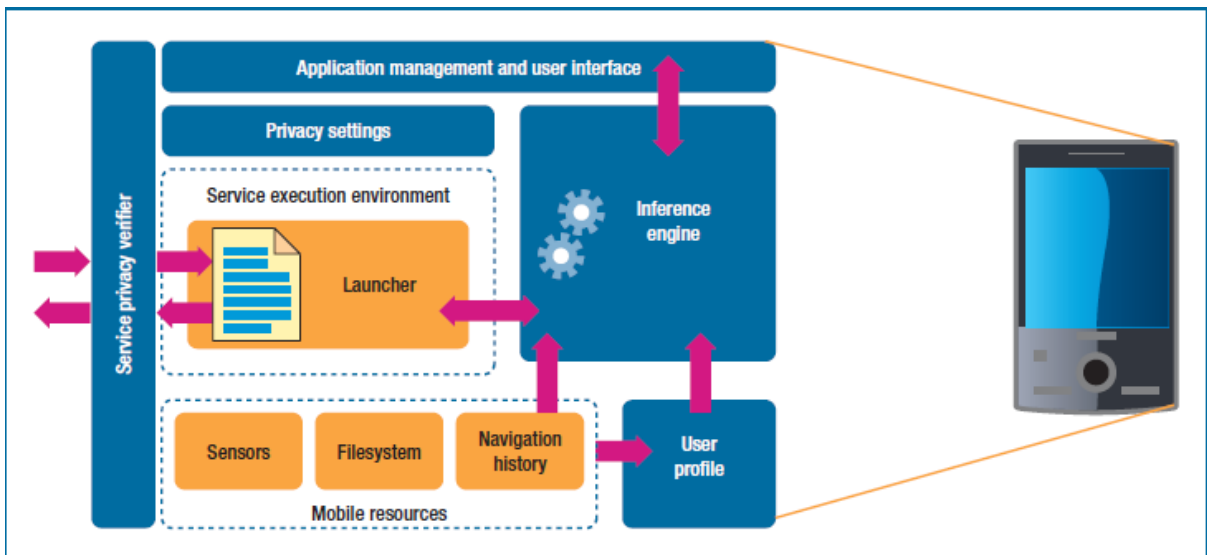


Figure 1. A High-level People-as-a-Service architecture.

The first pillar is achieved with the *inference engine*. The information can be requested according to sociological criteria, like age or gender. This means that the device should be able to evaluate if the owner's profile is suitable for certain interactions, which is achieved through the *inference engine*.

When the owner is suitable with the criteria, the device can channel the information to the user through the user interface or create a communication between external entities automatically without the device owner's intervention.

The *user profile* and the *mobile resources* are evaluated by the *inference engine* to determine if the profile is suitable or not with the requirements. The *user profile* contains the owner's identity, and the *mobile resources* provide simple contextual data like GPS position.

The user profile retrieves information monitoring the mobile resources and the user's interaction with the device. This information is used to create a sociological profile, which is stored in the device and used in the services.

The PeaaS architecture includes a *service execution environment*, which gives the capability to offer the sociological profile as a service, allowing external entities to interact with mobile device owners via the *inference engine*. Usually the communications are established using standard service protocols such as SOAP or REST.

The user privacy is probably the most important pillar that has to be guaranteed. All information about the device owner is kept exclusively in the device, with the *privacy settings* component. This component contains the privacy policies and determines which services can be deployed and executed, who can invoke the services, which mobile resources each service can access, and which type of information each device can return. These policies can be configured through the *user interface* by the user.

Finally, the *service privacy verifier* applies the privacy policies, blocking all service requests that don't comply the policies and notifying the device owner about conflicts.

Current implementations

Although People-as-a-Service is a recently and innovative concept, however there are two platforms developed in Spain by Gloin¹, a start-up which aims to exploit the model commercially.

Introducing nimBees

NimBees² is a platform developed by Gloin aimed at developing an API to facilitate the implementation of a new kind of mobile device apps, based on the new PeaaS model. The idea of this platform is to take contextual information of the mobile device and the owner and to use it to execute applications or services interested in this data. The API is aimed to facilitate the information and be the support of applications.

With this kind of applications owners can gain economic or social benefits exchanging their information. Of course, the mobile device owners can decide and manage what information they want to provide and to whom. All the data is saved and processed on the user's mobile device, so the privacy is never compromised and consumes less battery than other platforms.

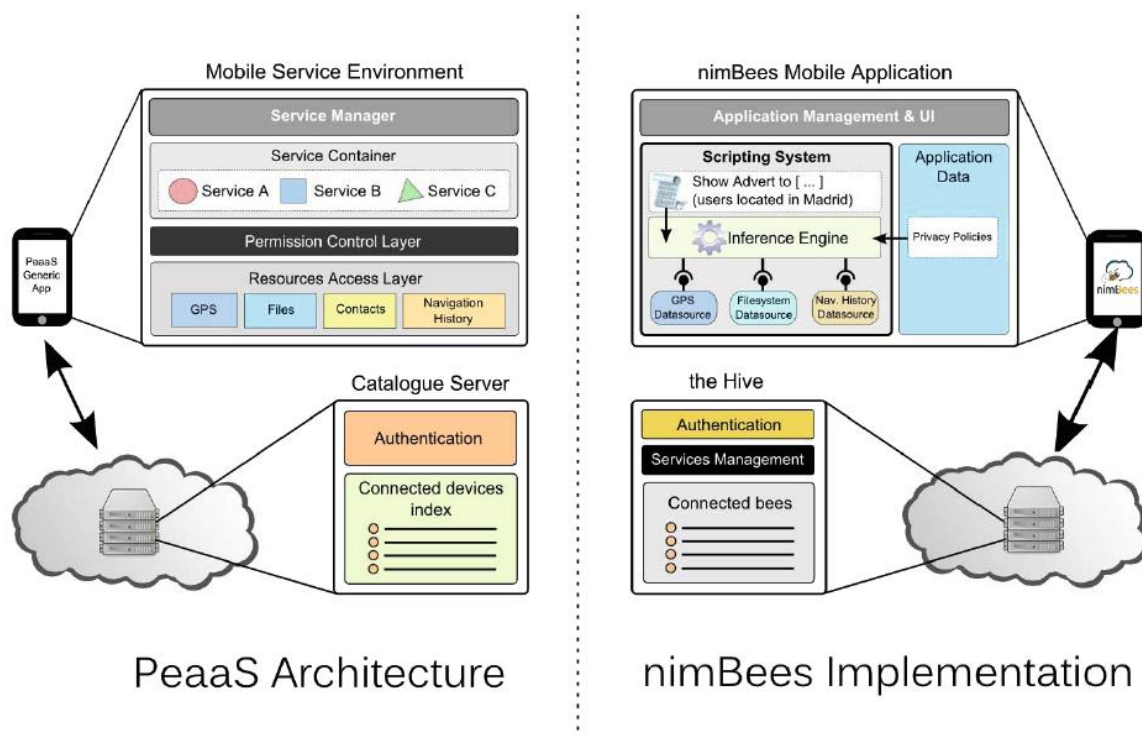


Figure 2. nimBees implementation of PeaaS architecture.

¹ <http://www.gloin.es/>

² <http://www.nimbees.com/>

For example, an advertiser can use this platform to create an application that sends advertising to special targets, based in their geographical position, age, interests, social status, etc.

The nimBees architecture [Figure 2] is a specialization of the one defined for the PeaaS model. Hence, the server and mobile architectures are designed as follows:

- **Catalogue server.** Based on the concept of the catalogue server, a non-mobile element with a fixed and well defined service endpoint where every new mobile device added to the PeaaS network must be connected, in order to authenticate and provide information about itself, and to retrieve information about the rest of connected devices. This catalogue server is called **The Hive** in nimBees.
- **Mobile Service Environment.** They are basically mobile applications that must be installed on every mobile device that wants to be part of a PeaaS network, providing an execution platform in which PeaaS-designed services can be deployed and executed. This is divided in four layers of abstraction:
 - o **Service Manager:** This layer constitutes the core of the mobile service environment. It provides the capabilities required for deploying, removing, running or stopping services in the service container, and a basic communication interface to establish a link with the catalogue server. It also provides an interface to the user of the device to configure the mobile service environment properly. In nimBees the inference engine is also included in this level. It consists on a component that monitors user activity to extract behavioural patterns that enrich the personal information profile. The information generated by this component is used to determine whether the owner matches the user profile required by the advertiser for a certain service execution.
 - o **Service Container:** This layer provides the execution environment where the PeaaS-based services are allocated, similar to a virtualization engine. The lifecycle of the allocated services is managed by the service manager layer. In nimBees this layer has been implemented as a scripting engine that provides an execution environment in which run the nimBees services, which are written using an own scripting language.
 - o **Permission Control Layer:** The access permissions to the resources and services provided by the mobile device are verified by this layer, thereby avoiding access to certain resources by unauthorized services. Users can block the access to certain resources or information in every moment and for any service.
 - o **Resources access Layer:** This layer abstracts PeaaS-based services from platform-specific implementations of each resource and service offered by mobile devices, providing a standard interface to be used homogeneously by the services. In nimBees the access to the device's

resources is provided to the deployed services in form of “sources of information”. Such source can be accessed by the scripting language with the proper syntax.

In conclusion, nimBees is a new platform that uses the power of People-as-a-Service concept to provide a new communication channel to the users, which can be used for advertising companies to communicate with specific groups of users in a new and more powerful segmentation capabilities.

Introducing BeeFun

Mobile marketing is a great niche for PeaaS, Google and Facebook are clear examples of how collecting sociological information from advertising targets has become extremely important to companies [10] [16], but with their model is difficult to gain user satisfaction. They try to create apps as tools for launching personalized recommendations to device owners, while digital advertisers can get segmented data about the users and show targeted and relevant advertising to them.

However, the sociological profile is always centralized on company servers, providing no transparency to the user and without a privacy guarantee about his data. Here is where PeaaS presents an alternative applied to mobile marketing that could benefit both consumers and advertisers.

BeeFun¹ is a mobile app developer for Android and iOS that makes use of the nimBees messaging infrastructure. It was developed by Gloin and allow users to interact with new kinds of targeted messages based in criteria such as age, city of residence, sex, current location, etc. It is a potential example for this market and the capability to provide rapid feedback on its benefits.

What BeeFun offers is the ability for advertisers to anonymously query advertising targets' sociological profiles to retrieve collective information about their preferences, areas of interest, and so on. This completely segmented data lets advertisers refine their marketing strategies and potentially increase their campaigns' effectiveness. Furthermore, the interoperation possibilities supported by PeaaS, and consequently by BeeFun, let advertisers send out interactive ads to the app's users on the basis of their profiles.

Interoperation between advertisers and targets is carried out through the services that advertisers can deploy in the devices. BeeFun is at the moment a merely a proof of concept and the applications and services have to be written in his proprietary scripting language, which contains basic instructions to ask the inference engine about the user's sociological profiles, request notifications and visual content to display on the user interface, and return and execution response to advertisers.

¹ <http://www.beefunapp.com/>

All the services that advertisers can deploy in mobile devices are maintained in an external catalogue of a *Hive* system, which registers all mobile devices connected to the network and manages the deployment and execution of services and applications in the devices.

The most important thing of this system, and which differentiates the PeaaS model from others (like Google or Facebook), is the accordance with the fourth pillar. The sociological profiles are always kept in their devices, with only some of the information, voluntarily given by the users) saved in the hive. This basic information allows the advertisers to interact with targets, because it would be difficult to find the targets without a centralized system.

The hive system lets the system preselect potential targets on the basis of this information. Then, the devices' inference engines still must evaluate the target's suitability.

Conclusions

Through this implementation of PeaaS we can explore new forms of mobile marketing that allow advertisers to reach their targets more precisely. It allows accessing detailed sociological profiles of the targets and retrieving collective information about them. Targets are able to provide their opinions about products and ads while preserving their privacy, what is an innovation in this field. The technology allows the generation of a view of collective interests, which could be relevant for market studies.

By focusing on people and their mobile devices as information sources, we can continue to think of a new advanced computing models that provide numerous benefits. Mobile-centric computing architectures can prove their great capability to generate sociological profiles through applications such as BeeFun.

Applying this model in fields with a high social impact could be extremely beneficial for society. PeaaS could let organizations identify patterns that could be contributing to health disorders among large communities. They could use this information to plan strategic actions that could help increase individuals' life expectancies. In government, PeaaS could allow retrieval of collective information about citizens' political views and preferred courses of action to enhance living standards and promote democracy.

However, the success of PeaaS requires a wide adoption by mobile device users, they need to receive some benefits in exchange for participating. PeaaS gives users more control over their privacy, but perhaps they could receive benefits from the companies using their information. For example, discounts for people using the PeaaS platform. Additionally, a third-party company that uses the platform to reach users could monetarily compensate those users or give them its products or services for free or a discounted price.

The Internet of People

There is growing interest in developing applications on the Internet of Things. The main objective of these applications is to integrate technology into people's everyday life in order to be a service to them. However there are many things to improve in the implementation. Usually, the user needs to set parameters within the application, and re-configure them when the context changes.

Nowadays Internet is available almost everywhere, connecting everyday physical objects to the Net is getting ever easier. One of the main goals behind such smart integration of devices is to simplify people's lives by having technology work for them seamlessly [6]. For instance, you can remotely switch on your house's air-conditioning system using your smartphone to get a comfortable temperature when arriving home.

However, how IoT technologies are currently integrated with human still leaves much room for improvement [17]. We need to develop suitable mechanisms to properly adapt to people's context and make the technology able to work for people. Currently people are forced to change their context to fit technological requirements.

Technology should 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. In this desirable scenario, the air-conditioning system will know where we are in our way back home, instead of switching on at specific preconfigured time.

Enabling such scenarios requires moving from the Internet of Things to the *Internet of People* (IoP) [30]. In this infrastructure, smartphones play a central role, reflecting their current user as the main interface allowing people to be connected to the Internet. However, the smartphones have to meet a serie of requirements and capabilities [28]:

- The smartphone needs to be capable of learning about its owner and their context, constructing a digital profile.
- It needs to transparently negotiate and propose interactions with other devices on the Internet, reacting to stimuli and handling relationships.
- 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 the digital profile.

The infrastructure to support the IoP concept can be achieved with a combination of the Social Devices¹ and People-as-a-Service 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 their context and sociological profile. The combination of these two contributes to constructing an IoP by allowing

¹ <http://socialdevices.github.io/>

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, opening the way to the development of new kinds of human-friendly services and applications.

As smartphones contains a lot of information about their owners, and sensors to retrieve contextual information about the surroundings of it and, consequently, about the owner, they are the perfect device to be the central piece of this new concept [30]. Smartphones can reduce the complexity of user interaction with IoT-linked devices. They can know the user's circumstances, as they are almost always with the user. The contextual data can be used to know information about the user and to learn about him and to detect how the person behaves when they are at certain locations or surrounded by certain other people.

One of the most important aspects of People-as-a-Service is the fact that all the information about the user is kept in the owner's device, and is securely provided as a service to third parties directly from the smartphone. The user can limit who can access to that information and know who and when is receiving it. The user can control his sociological profile and all the information easily, just modifying or deleting it from the device.

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 profile.

Current IoT technology needs people-centric enhancements. As set out in (Miranda et al., 2014) [30], the goals for any pervasive computing context should be:

- **Be social:** Interactions in which things, devices and people participate must be social. The IoP should allow for heterogeneity by supporting the different types of devices that people use, allowing them to interact with people and other devices. Users need to be empowered to adjust their preferences and policies about when and with whom their devices are socializing.
- **Be personalized:** Interactions between devices must be personalized to the sociological profiles and contexts of their users, providing a transparent mechanism for this customization. Users must be able to adjust their preferences in order to control how their profile is used by other IoP stakeholders.
- **Be proactive:** The triggering of interactions must be proactive, not manually commanded by the user. The IoT should allow for device heterogeneity to allow them all to interact more proactively, without the need of manage all connected things. Users must be empowered to adjust their preferences to control how proactive their surrounding things are, this is done by proactivity policy. It is very important because can introduce security risks.
- **Be predictable:** Interactions must be predictable and triggered according to a predictable context that the user had previously identified, and with a specific

behaviour defined. Users must be able to identify and tag that context, define the expected behaviour and set the privacy policies for sharing their information. The user has to know how an interaction can be stopped immediately and how to prevent a misbehaviour in future.

Internet of People Middleware architecture

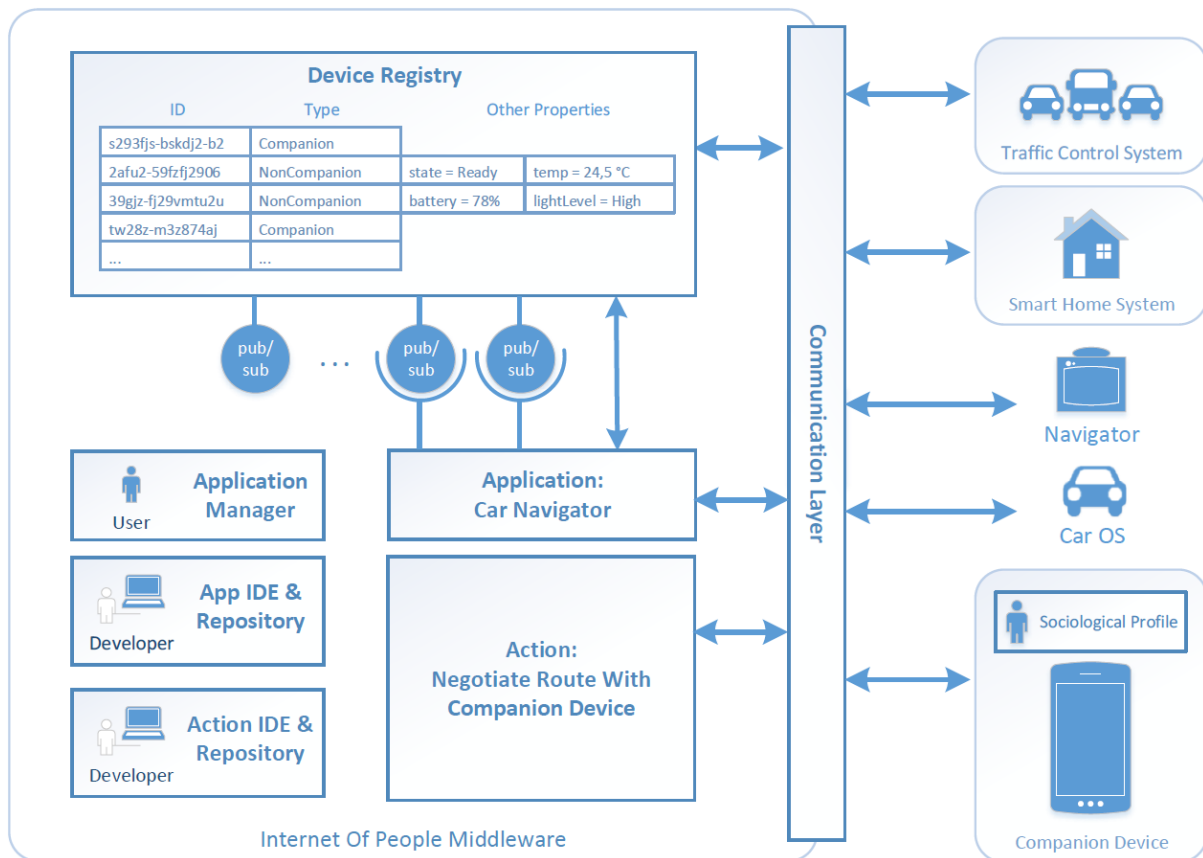


Figure 3. Architectural description of the IoP middleware.

In this architecture, the central component is the Device Registry, who maintains the information about the different devices managed by the system. The devices are classified into *Companion* and *NonCompanion*.

Companion devices are those that maintain contextual and sociological information about their owners, while *NonCompanions* are “things” which can register their capabilities and additional information about their state.

The *Action Repository* 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 middleware.

The *Application Manager* 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 of mood.

Sociological Profiles

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 contextual data such as location. Devices can also process contextual information such as a proximity graph of surrounding people/devices/things.

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.

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 of users, and in which context the information should be provided.

Finally, the Sociological Profile provides an interface for device owners to personalize the way in which the device behaves under specific circumstances.

Conclusion

Internet of Things is having a successful growth, however, people's interaction with this kind of systems is still far from friendly. The term Internet of People has been used before, but it referred to traditional Web systems designed only for humans to use. Now the idea of this term is to bring the Internet of Things closer to people.

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.

To construct more human-like, predictable, proactive and social interactions between people and "things", the IoP middleware offers a Web-based IDE and tools¹, which we will discuss in next chapter.

¹ <http://orchestratorjs.org>

Integration with Social Devices and Internet of Things

People-as-a-Service and Social Devices combined provides new interactions and capabilities to users when they are co-located, but IoT can be also integrated with them, providing new capabilities and communications with everyday objects near the user. In some years 50 billion of devices could be connected [11].

Some IoT-related scenarios have been already implemented and introduced using Social Devices, for example The Social Coffe Machine [25]. This coffee machine asks to the user, based on his context, if he wants to have a coffee, and then the machine invited other people to join the coffee break.

Vázquez and López-de-Ipina [50] have been working on this subject too, they introduced the concept of semantic device as a system “*able to spontaneously discover, exchange and share context information with other fellow semantic devices as well as augment this context information via reasoning in order to better understand the situation and perform the appropriate reactive response.*” [48][49].

They have developed four prototypes of what they call *Social Devices*, integrated with The Internet of Things to make some objects capable to interact with users. They reutilize the concept of *Semantic Gadget*, which was introduced by Lassila and Adler as devices capable of performing “*discovery and utilization of services without human guidance or intervention*” [24].

This concept emphasizes even more the benefits of the communication with other devices in order to determine the appropriate behaviour. Internet is the enabling technology in this case, since the knowledge provided by local devices can be complemented with information obtained from the Internet (which is provided by other devices, services or humans).

Flexeo’s SensorMap: a wireless sensor network

This is the first prototype by Vazquez and Lopez-de-Ipina. The goal of this project was designing a system for remotely monitoring wireless sensor networks through the Internet, and integrating all the data into business processes in order to execute data-mining operations and determine correlations among data. Scenarios related to “health at home” and remote monitoring of industrial facilities were designed in order to test the resulting system.

As an additional outcome, a mash-up of sensor collected data and Google Maps was developed, called SensorMap. This subsystem illustrated the potential applications of publishing up-to-date sensor data on the Internet in order to dynamically integrate different sources of information for providing higher level services.

The next [Figure 4], from [50], shows a sensorized chair with two pressure sensors (at the seat and at the back) connected to a wireless sensor network node in order to provide information about its use to local and remote objects. [Figure 5] contains a

screenshot of the SensorMap application showing the state of the chair (if someone is sat on and/or leaned backwards).

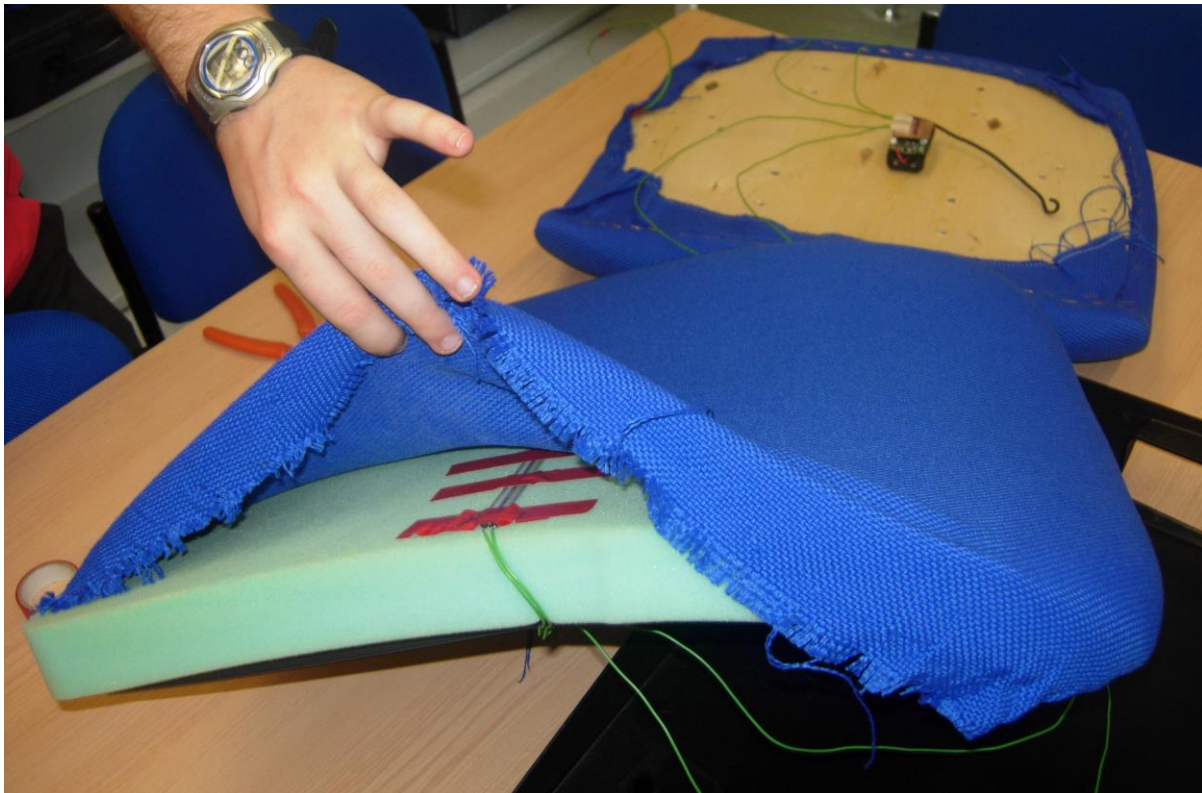


Figure 4. Pressure sensors in the chair.

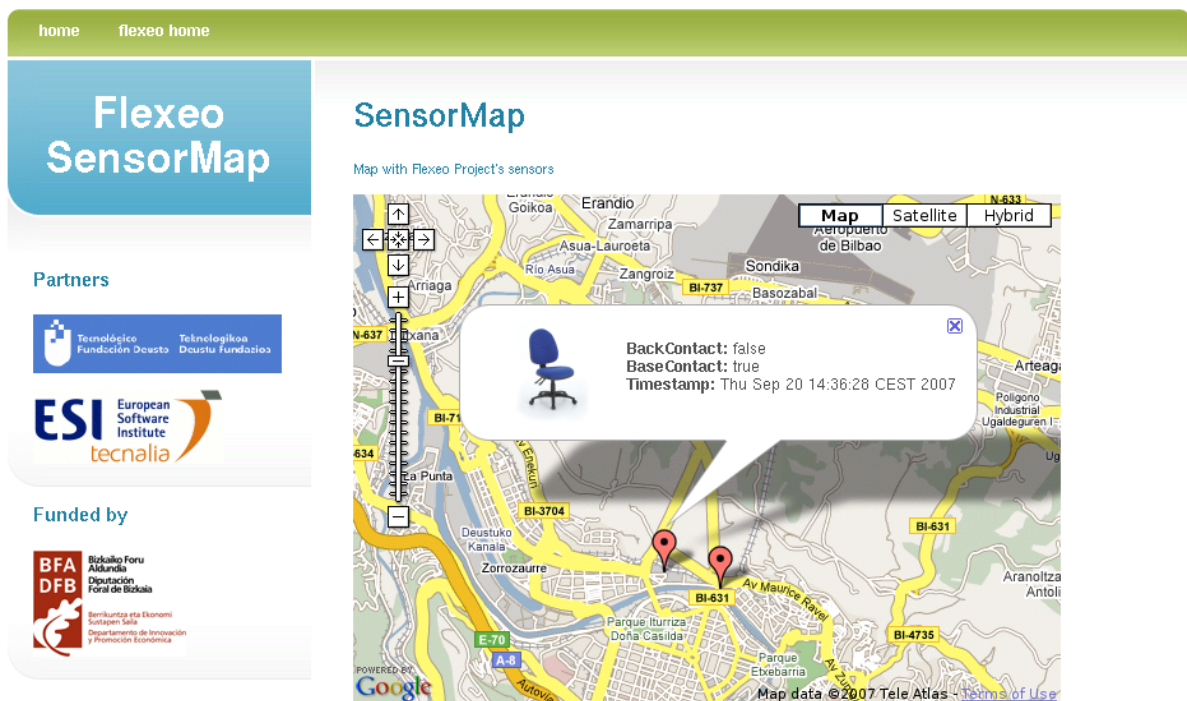


Figure 5. Flexeo's SensorMap showing the state of a chair.

RealWidgets: from the desktop to the real world

With RealWidgets they wanted to embody the functional capabilities of these digital entities into real world tiny wireless displays, in order to have small “windows” deployed everywhere opened to the information from the Internet.

The RealWidget is formed by an OLED display, with high resolution and contrast while small energy consumption, integrated with a CrossbowMote2 wireless sensor network node. A computer acted as a gateway between the Internet and the wireless sensor network, running a RealWidget Management Application that connected to the required sites on the Web, downloaded the information, analyzed it and finally sent the content to the appropriate widget as configured by the user. [Figure 6] illustrates a RealWidget showing information about the liquid level in a remote chemical container that was monitored by a wireless sensor network over the Internet.

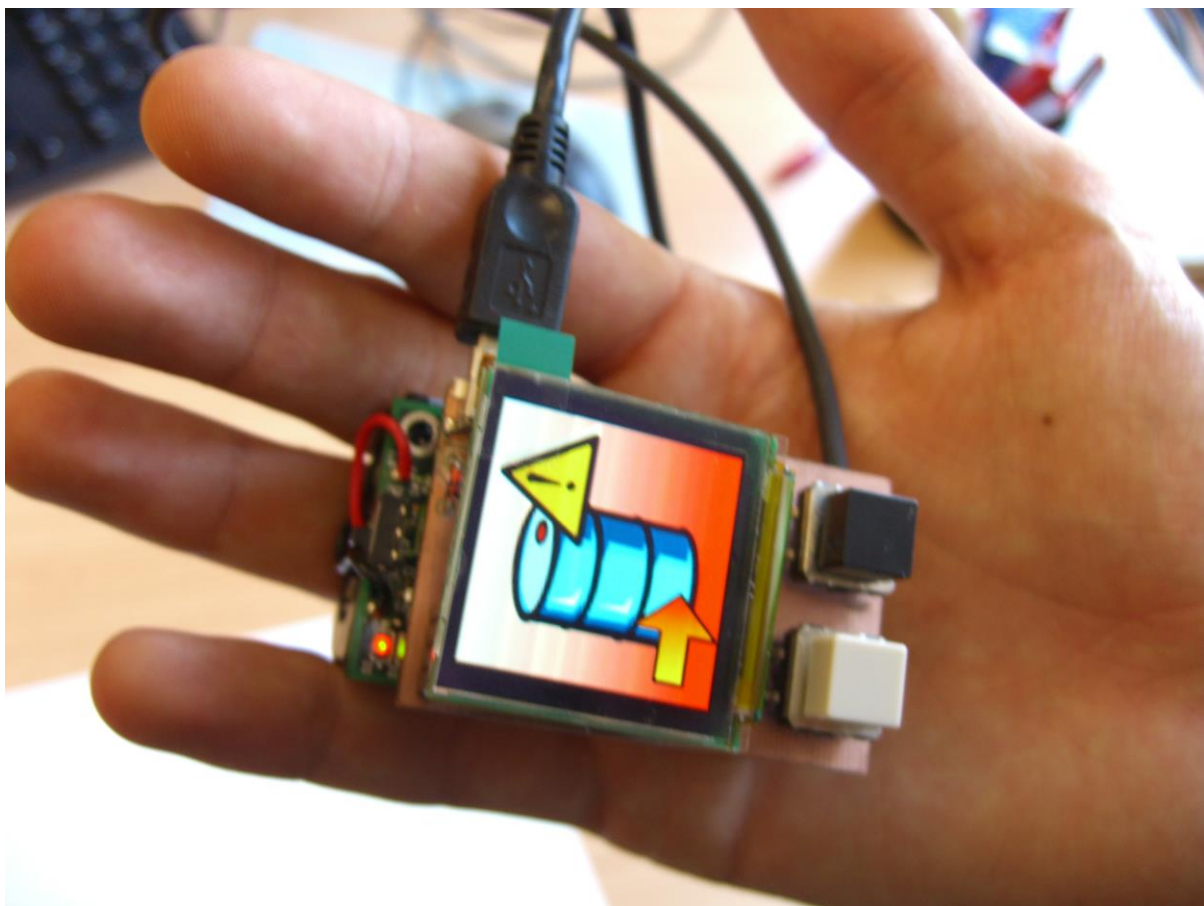


Figure 6. RealWidged alerting about the liquid level in a tank.

Two buttons were provided for interacting with the RealWidget, basically for managing the energy by postponing the information for a later time, or immediately discarding it.

SmartPlants: autonomous objects that interact with their environment

SmartPlants is a completely decentralized architecture for designing semantic devices that collaborate by sharing information about the environment, and spontaneously react to changes in the context. This was one of the targets of the research: to create an artefact that could be attached to real objects, augmenting their perceptions and providing them with intelligent capabilities.



Figure 7. The SmartPlant prototype.

The artefact includes a wireless sensor network that provides temperature and light measures about different nearby locations, asking the user to be moved to most suitable place using a synthesized voice. The plant is also able to download ontologies from the Internet in order to interpret particular expressions and predicates provided by surrounding sensors.

Aware-Umbrella: a reactive device integrating local and global communication

The Aware-Umbrella was the last prototype created by Vazquez and Lopez-de-Ipiña. This integrates local and global information, which means the device gets contextual information provided by other local objects in the environment as well as remote services on the Internet, integrating all this knowledge to realize a higher level of context-awareness.

In this scenario a smart umbrella can obtain current weather information from surrounding sensors, as well as the weather forecast for the next hours from the Internet. The smart umbrella reacts when the user is leaving home without taking it, by issuing a synthesized voice alert.

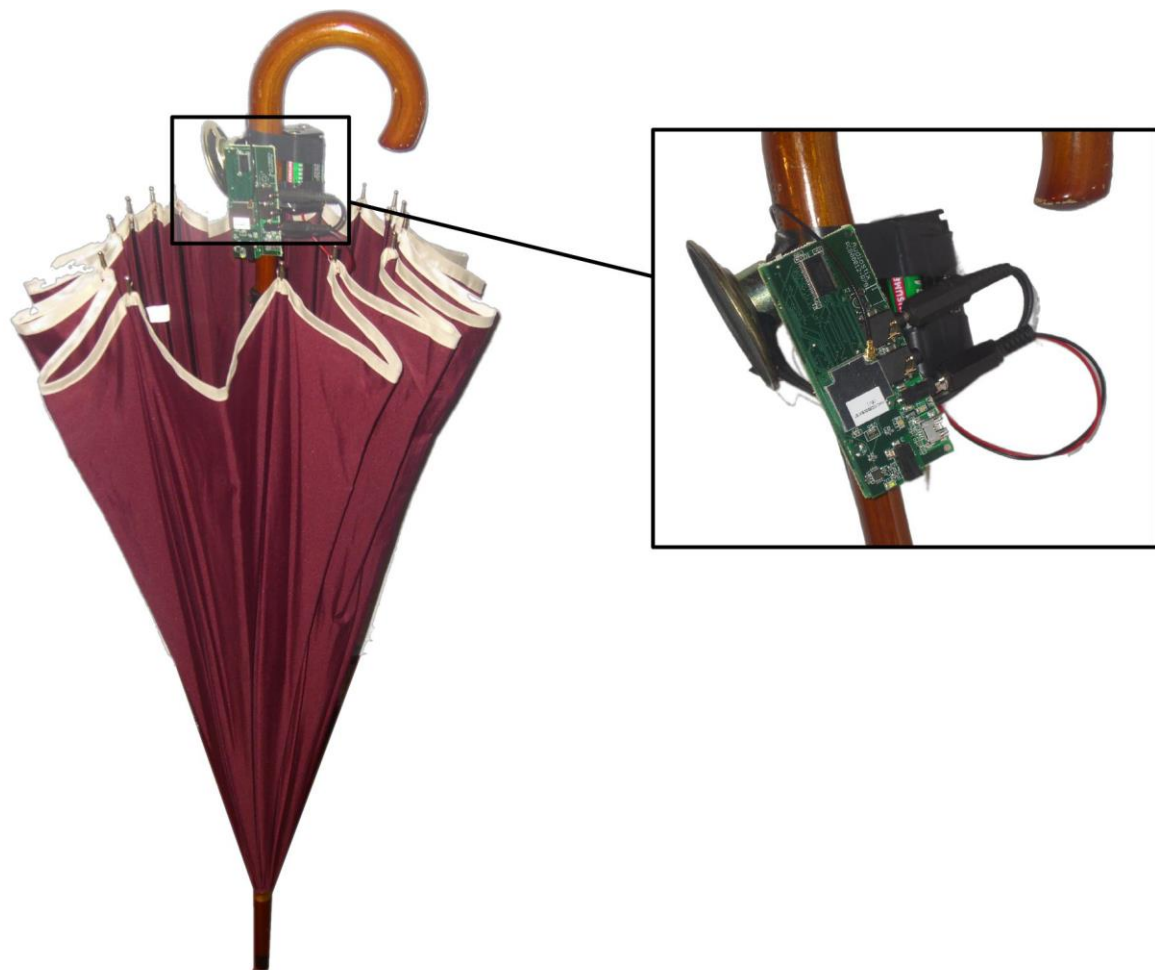


Figure 8. The Aware-Umbrella prototype.

The Umbrella checks the door state to know if the user is leaving home, it can get information about the weather forecast and in case of rain will alert the user to take the umbrella.

Chapter 2

Social Devices

Social Devices is a new concept focused on enriching local interaction between people using the technology, supporting the face-to-face social interaction instead of the remote ones. The concept was first introduced in 2011 as a joint work by Nokia Research Center, Aalto University, and the Tampere University of Technology [26]. The goal of Social Devices is to allow heterogeneous multi-user and multi-device applications to be created for any type of device.

This new concept breaks with the current communication model, where people can interact with other users remotely, in any part of the world. Social Devices is a Mobile Cloud Computing service that offers a platform for connecting mobile devices with each other, introducing co-located social interactions based on proximity.

Social Devices Platform (SDP) is a prototype implementation of the concept. This platform allows the interaction between devices and also between humans and devices. The system is proactive, what means, each device is an active participant and can initiate interactions with people or other devices. This platform uses the *Action-Oriented Programming Model*, where any interaction enabled by Social devices is called an action.

The target of Social Devices is stimulate social interactions when people meet face-to-face, but also device-to-device interaction can be enriched and devices are able to interact between them [37]. These interactions would be invisible to the user, he only will receive the feedback of the communication.

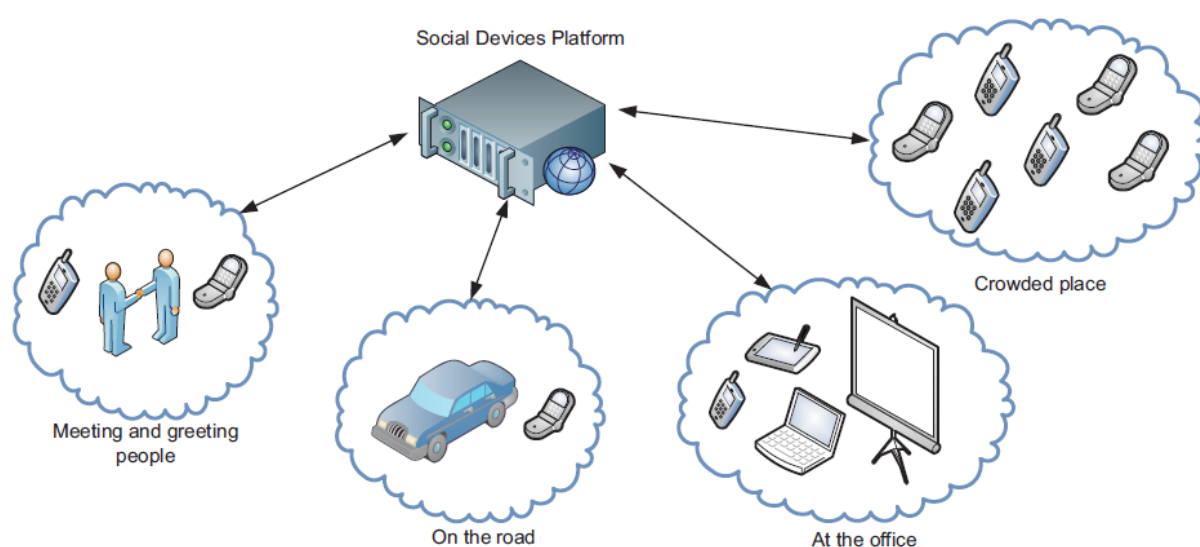


Figure 9. Some examples of interactions made possible with Social Devices.

Social Devices can add new functionality to our social interactions. For example, when two businessmen meet, their devices can exchange contact cards automatically. The devices can also greet the name of other person when meet in the street.

Social Devices is supported by the *Action-Oriented Programming Model* [1] (Aaltonen, Myllärniemi, Raatikainen, Mäkitalo, & Pääkko, 2013). To help the understanding of the platform we need to know this new programming model and the motivation behind it.

The Action-Oriented Programming Model

Nowadays, smart devices are capable and connected and can be used for everything. Devices can perform tasks at the background and even controlled by other devices. They can act according some situations happening in the surroundings with the help of the multiple sensors they have. However, available applications typically operate on one device at a time.

Probably we need to start thinking about new multi-device applications that can create coordinated interactions between devices in a pervasive manner. For example, your device could trigger an action to show a photo to your friends at the same moment, so all of your friends would see your photo at the same moment. This coordinated interaction between devices can be realized with mobile cloud computing, and it would be called an *Action*. Actions can start proactively and pervasively, even without user interaction, if the given preconditions are met.

To support this scenario we require additional effort with traditional programming models, this is because the traditional programming constructs do not directly support necessary primitives such as actions and triggers.

With *action-oriented programming model* we can develop applications that make possible the use of actions to execute multi-device tasks in a coordinated, proactive and pervasive way.

In the *Action-Oriented Programming Model* a program consists of one or more actions, one of which executed by an initiating trigger.

An action developer can easily define *actions*. Within an action, the developer should be able to define coordinated behaviour among several devices. The actions will be executed based in *preconditions* and *triggers*.

An action defines the behaviour, and the *capabilities* provide the functionality. The capabilities are defined in *interfaces*, which can be developed according each device. An interface consists of operations, and they have to be registered by the device.

When an action is executed, it starts one or several operations enabled in the device. If the required operation is not enable, the device will do nothing. This makes the user able to decide what capabilities want to be executes or not.

Actions

An *action* is the basic unit of modularity, it usually will have *parameters*, one or several *preconditions* and a *body*. An action can also define *roles*, and encapsulate the joint behaviour of multiple devices.

- **Parameters** can be the same as common parameters in a normal function in any programming language. We can send information of any type from one device to others through this parameters.
- The **precondition** must evaluate to True for each device with the parameter combination. A precondition could be, for example, that the other device must be in my contact list to be able to start the action.
- Defining **roles** is optional. If we have an action that shares a picture with my friends, the “sharer” device will have a role of server, and the other devices will be the clients that show the picture in their screens.
- Finally, an action has a **body**, which defines the behaviour, describing how operations are invoked in the devices assigned to each role. In our example, the server role would send the picture to surrounding devices, and they would show it on the screen.

Concept	Description
Action	The basic unit of modularity, encapsulating the joint behaviour of several devices. It contains the body, where is all the code that defines the behaviour and all the invoked operations.
Action precondition	A Boolean expression that defines all the requirements to be accomplished before the action can be executed.
Action role	Actions can be specific for roles, each participant can be assigned to a role, which determines the operations that will be executed.
Device	A device that is able to participate in actions. Roles are assigned to devices, and they can participate in a synchronized and coordinated manner.
Device capability	Capabilities are operations that can be executed by a running action. They are associated with the tasks of functionalities of the device.
Interface	Defines the contract with all the operations associated with a specific device capability.
Trigger	A trigger is a small condition that initiates the scheduling of a certain action. It can be used to avoid checking the preconditions regularly.
Scheduling	The scheduling is a task that attempts to start the execution of an action. When an action precondition is satisfied, it tries to find a set of devices and assign them into the action roles.
Dynamic information	The dynamic information is needed to be able to recognize triggers and evaluate preconditions. This information is out of the scope of the Action-Oriented Programming Model.

Table 1. The key concepts in the Action-Oriented Programming Model.

Triggers and Scheduling

The Action-Oriented Programming Model provides *triggers*, which can initiate an action proactively. When a trigger is received the related action is attempted to be executed. Triggers are not strictly necessary, but can help to evaluate complex preconditions only when the calculated state has changed.

The use of triggers requires a mapping between the trigger and the actions to be executed. These relations between triggers and actions need to be registered in a *triggering table*, which is hosted by a *trigger registry*. There can exist a many-to-many relation.

The *scheduling* is a process that selects one action from the set of actions related to a trigger when it is triggered. This process involves finding the action and evaluating the preconditions. It is possible to assign some devices or trigger to specific roles.

Incoming triggers can be stored in an *active triggers table* to wait for rescheduling if preconditions are not immediately satisfied. Triggers are not restricted and can be sent by any source, such as services or devices.

Artifacts and Infrastructure

In Action-Oriented Programs the concepts need to be encapsulated into artefacts see [Table 2].

Artefact	Description
Action description	A piece of code that defines the action, its precondition, device roles, and the action body.
Interface description	A piece of code that defines an interface, which consists of a set of operations.
Trigger description	A piece of code that defines the trigger: a trigger notifies when the action is tried to be scheduled.
Device capability	Software resource that implements a certain interface for a given device. By installing a certain device capability to her device, the user indicates her ability to participate in actions that utilize this capability.

Table 2. The Programming Artifacts of the Action-Oriented Programming Model.

Also, the Action-Oriented Programming Model must be able to schedule and execute the actions, so a supporting infrastructure is needed. There are several services implemented in the infrastructure, see [Table 3], like a registry for actions, devices and triggers, as well as a service for scheduling and executing of actions.

Service	Description
Action registry	A registry for all possible actions and interfaces that are available.
Device registry	A registry for all devices and their installed capabilities. This registry may also contain additional dynamic information about the devices, such as device proximities and states, as well as contain information about users and device owners.
Trigger registry	A registry that consists of a triggering table, which holds information about possible triggers and corresponding actions; and of an active triggers table, which holds information about trigger instances that have been observed, are still alive, and have not yet been successfully scheduled.
Scheduling	Management of the scheduling task.
Execution	Management of the action execution, that is, synchronization and coordination of the devices.

Table 3. The Services in the Action-Oriented Programming Model Infrastructure.

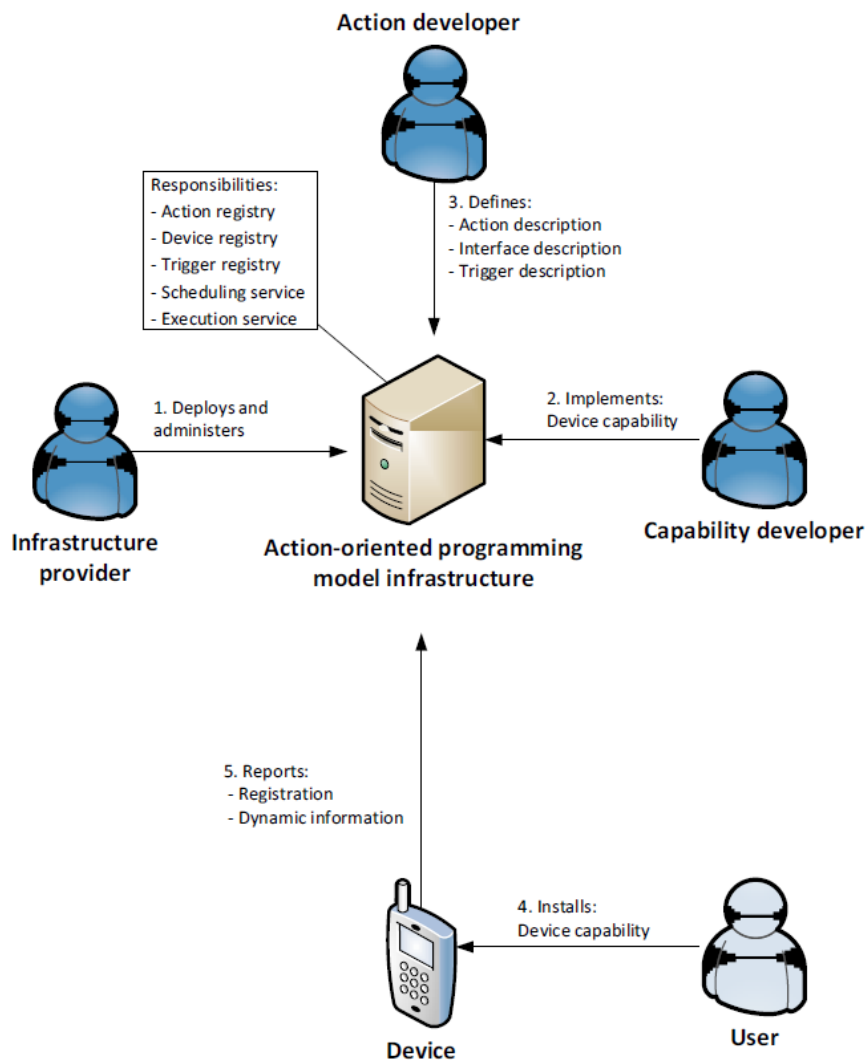


Figure 10. Actors, artefacts and responsibilities in the action-oriented programming model infrastructure.

There are several actors and they interact with the artefacts in different ways [Figure 10].

- The *infrastructure provider* deploys and administers the Action-Oriented infrastructure.
- The *capability developer* is responsible for implementing the capabilities for the devices, accordingly to the defined interfaces.
- The *action developer* uses the interfaces to execute the necessary capabilities in each device. The developer doesn't know the identities or characteristics of the devices, and how the capabilities are implemented.
- The *user* installs the desired capabilities to the device.
- The *device* could be considered as an actor too, it will register itself to the infrastructure at runtime, and constantly reports dynamic information about its state.

The order of the illustrated events can be different, because the action developer can reuse existing device capabilities instead of writing code for new device capabilities to support multiple mobile platforms.

The three basic steps to create an Action-Oriented program would be the following:

- Define or implement the *interfaces* for the necessary capabilities of the device.
- Define the *actions* to be executed, the program itself, and specify the needed preconditions for starting the action.
- Define the *triggers* that initiate the actions.



Requirements and characteristics of Social Devices

Our devices have a lot of information about us, maybe our mobile phone knows where we want to drive when we enter in the car, and it can communicate our destiny automatically to our GPS system. Also at the office, employees' devices can synchronize each user calendar to establish the day and time of a meeting, and then communicate the result to each employee. The concept can be taken beyond, for example, our device knows we have a meeting in the third floor, so it could send that information to the elevator when we enter in it.

The key of Social Devices is to utilize resources and capabilities of multiple physically co-located devices and the proximity of the devices in a user observable manner. Some of these devices can be stationary, like an elevator or the screen in a meeting room. This technology would fit well in conjunction of other technologies like pervasive computing¹ [7] or Internet of Things [17].

However, Social Devices requires some devices capabilities to be able to work in each device. The device must be a computationally capable device and be able to interact with other devices and users in their proximity, for example via Bluetooth. This requirement is easy to accomplish for devices like mobile phones, laptops or computers, but is difficult for other kind of devices like elevators, GPS systems, cars, etc.

The capable devices need to have identities, so the platform would be able to keep track of the device identities. Devices collaborate in a way that is often personal to the user and related to user preferences. There is a centralized registry where devices need to register their identity and capabilities. The platform allows to register multiple devices for each user and select which of them is the principal, the most used.

Another requirement is the need to know which other devices are nearby the device. Social Devices interactions work in a co-located way, and require that users and devices are near each other. Thus, the proximity of devices needs to be discovered and stored in the Social Devices Platform, who maintains proximity information in a form of a mathematical graph, where nodes denote devices and edges denote the distance of the devices. This information is acquired by the devices and reported to the centralized server, who updates the proximity graph.

There is also a need to know what different collaborative actions are available between devices. The platform manages *action descriptions* centrally. As we have seen, an action has a *body* and a *precondition*. The body describes what the action will do, and the precondition is a set of requirement to be comply the device to be able to execute the action. Each device has a set of enabled capabilities that can be used by the running actions, having a required capability could be a precondition to execute a determined action. Also we may need to know which kinds of devices can participate in an action, what will be also captured in the action precondition.

¹ **Pervasive computing**, also called ubiquitous computing, is a concept where computing is made to appear everywhere and anywhere.

Capabilities are a way to manage permissions in Social Devices. The user can enable or disable different capabilities. For example, there is a capability able to speak aloud by synthesizing text, if the user doesn't want the device can speak, he can disable the capability. However the user won't be able to execute actions that require that capability.

The condition when actions are executed needs to be decided automatically at runtime, but the preconditions are not only about static capabilities of devices, but they may constrain dynamic properties changing on the device. Dynamic conditions that affect whether a device can participate in an action are monitored via *device states*. The devices continuously report their states to the central server, so it has all the necessary information to decide if the action can take place or not. States can be based on general preferences of the user as, for example, if the device is in silent mode.

All the conditions have to be evaluated without user interaction, but continuous searching for devices and evaluating preconditions is computationally hard. Here is where *triggers* take place to mark the need for action execution. Certain actions are attempted to be scheduled when a certain trigger is received. A trigger can be an update in the state of the device, a proximity change or an external event. The devices only need to identify the condition to raise a trigger, which will be received by the central server, who decides whether and which action must be initiated.

Also when executing an action, one has to decide which concrete devices will participate in it. The main task is to relate the potentially available devices to the preconditions stated in the action description and to decide the roles that will be assigned. Sometimes the same set of devices can participate in the action with different roles, for example in a dialog application.

Finally, the action needs to be executed, but each device has a different software and hardware features. The action coordination or orchestration requires defining a control flow for each device, regardless the different software or hardware specifications. In the SDP there is a generic system where various cooperation possibilities are available defining a platform into which coordination is built in.

These requirements and characteristics are necessary primarily for user and private devices, because they must allow the owner to change their preferences and save the privacy of the user. However, for public Social Devices, like an elevator, there is no need to evaluate preconditions or capabilities, cause is a simple device which only will accept requests of nearby devices and act accordingly.

Architecture

This is the basic architecture of Social Devices, the devices are registered in the Controller component in the server, and the user can install the interfaces and capabilities he wants. Once connected, the device will look for nearby devices and report them to the server, what will trigger some events. The server then evaluates if there is an action available to be executed for this device, and orchestrator executes it.

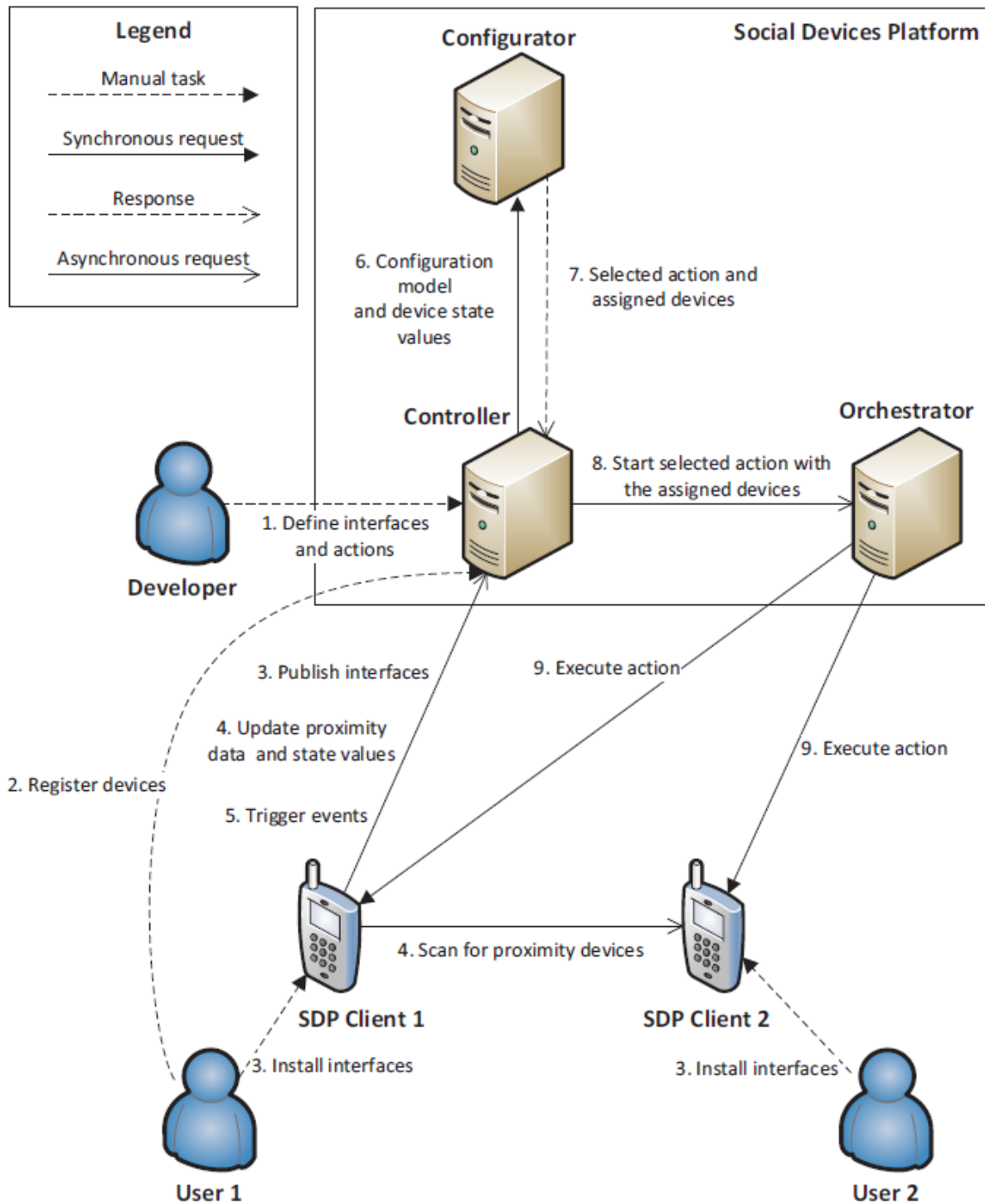


Figure 11. The basic architecture of Social Devices, and connections done step by step.

Orchestrator.js

We have seen a small description about *Social Devices Platform*, who implements the concept of Social Devices. However we have seen it as an introduction to the new platform, called *Orchestrator.js* and implemented by the same group of developers. This new platform is the focus of this work, and it is more powerful than SDP.

Orchestrator.js (OJS) is a completely new middleware that supports Social Devices concept. It is implemented as a single cloud service that offers a complete set of tools to implement and prototype Social Devices applications. It is specially designed to implement interactions between devices and people, and for observing situations when these interactions should proactively get triggered.

Social Devices applications differ from traditional mobile apps, but Orchestrator.js can be easily hosted on any regular computer and be used for quickly developing multi-device applications. The main task of Orchestrator.js is to maintain persistent connection to devices, store applications and contextual information of devices, and offer means for triggering applications.

Architecture

Orchestrator.js is based on [Node.js](#)¹ and the communication is based on [Socket.IO](#)² Protocol. This offers highly efficient input/output operations between Orchestrator.js server and its devices. The server is formed by the following components:

- The **Orchestrator Core** creates new action instances and makes sure that the actions work properly. It maintains the connections to devices and coordinates the devices by handling incoming and outgoing events based on the logic defined in actions.
- The **Application Manager** stores and executes the applications, allowing users to start/stop them, and stores user-related settings that may be required by the application. The applications are like daemons or child processes that run constantly and monitor changes with help of observers. When the app detects changes, it uses some scheduling algorithm defined by the developer to determine whether some interaction should take place.
- The **Device Registry** maintains contextual state information about each device, which is stored to [MongoDB](#)³ database. Moreover, the registry maintains the *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.

¹ **Node.js** is a server-side JavaScript platform built on Chrome's JavaScript runtime.

² **Socket.IO** is a new non-standardized protocol for relying events between device and server.

³ **MongoDB** is an open-source document Non-SQL database.

- 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 behaviour of devices and interactions between users by utilizing device capabilities.
- The **Capability Repository** contains descriptions of each capability, like Java's interfaces. The functionality is not written here, this only defines what methods the capability must implement, and what parameter each method takes. The developers must implement the capabilities like they wish and using whatever language, as long as they follow the capability description.

Tools

OrchestratorJS 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 this console. It can be used directly from mobile phone's browser and also be embedded inside mobile apps with *WebView UI* components.

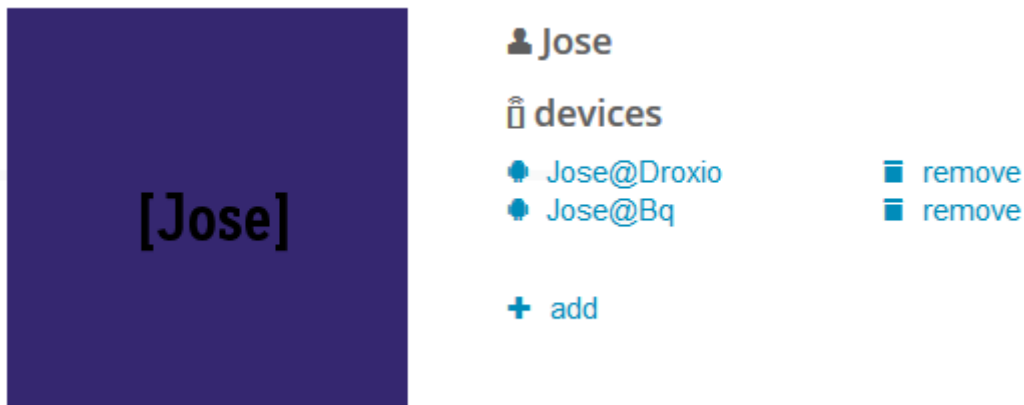


Figure 12. Web Console view where the user can add, modify or remove his devices.

From this view, users can add, remove and manage their devices. In [Figure 12] we can see 2 devices registered to my account. Each device has an identifier, composed by the username and the device name.

We can see all the registered devices in the Device Monitor view [Figure 13]. In this view we have all the information available by the platform of all the devices. For example, we can see the Bluetooth Address, UUID, Owner, Type of device, enabled capabilities, and the context data.

Only registered devices can be used in the platform, so when a new user wants not use the application, he has to register the device using the Web Console, and enable all the desired capabilities.

Devices can be modified and removed, so if the user doesn't want to continue using the platform, he can remove his registered devices. All the information about that device will be totally removed from the platform, and won't be possible to use it anymore.

Identity	MAC	BLE UUID	Owner	Type	Capabilities
Jose@Droxio	d0:e7:82:36:cf:8e	059bb767-9803-44b2-85ea-e7d66d9fdd3e	Jose	android	
Jose@Bq	00:09:88:9a:cf:2e	687203e8-427c-4870-a4f8-cbfcc7e4a3c4	Jose	android	

Figure 13. Device Monitor with my registered devices.

All the enabled capabilities appear in the view, they are codified as a coloured square. We can enable the desired capabilities in the server-side from the same view where we register new devices.

Required Settings

Device Identity
Jose@Droxio

Username

Device name

android
 gadgeteer
 arduino
 ios

Optional Settings

Bluetooth MAC

Device Capabilities

- Apocalymbics
- Apocalymbics2
- BluetoothCapability
- CoffeeCapability
- ContactCapability
- DialogCapability
- GpsCapability
- HashTagCapability
- LocateCapability
- NotificationCapability
- PhoneNumberCapability
- PlayerCapability
- PlayerDevice
- SocialNetworkCapability
- TalkingCapability
- TestCapability
- TreasureCapability
- UriScreen
- VisitingCardCapability

Figure 14. Register new Device view. From this view we can add all the information about the device, and enable the capabilities we want to use.

The Web Console also offers a Web-based IDE for implementing the applications without install any software. Developers can define capabilities, actions and observers directly from their browsers, and trigger them manually for a predefined set of devices.

We can see, create and modify actions from the *Actions* → *Definitions* menu. We will see a view like [Figure 15]. All the defined actions will appear on the right menu, also create a new Action is possible with the first option.

On the left we have a big editor where we can introduce the body of the action. This editor is very powerful, it will colours the syntax while we are coding and show the line numbers. This IDE allows us to program actions entirely without the need of an external software. We can save the action directly to the server pressing the *Push* button.



Figure 15. Example of the *Actions Definition* view, with the *Apocalymbics* action loaded in the editor.

We have two fields below the editor which allows us to execute the action in a device. Parameters can be devices or simple data like Strings, Integers, etc.

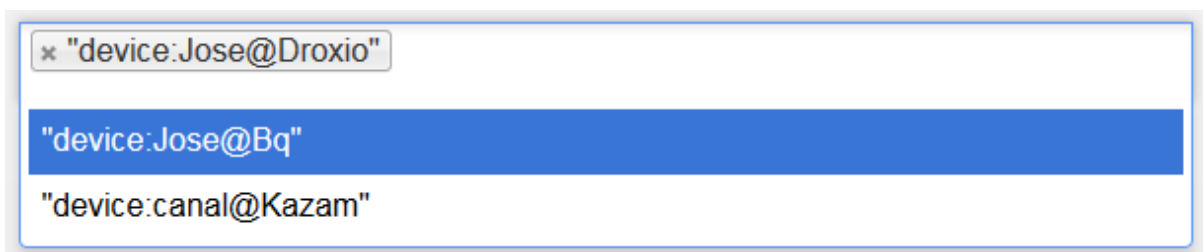


Figure 16. We can select devices to execute the action directly in the dropdown list.

Capabilities are implemented with the native platform specific tools, like with Eclipse (Android), however, the OrchestratorJS platform offers a tool for defining the capability descriptions. This view [Figure 17] is pretty similar to the Actions Definition view, with all the capabilities on the right menu and the editor on the left.

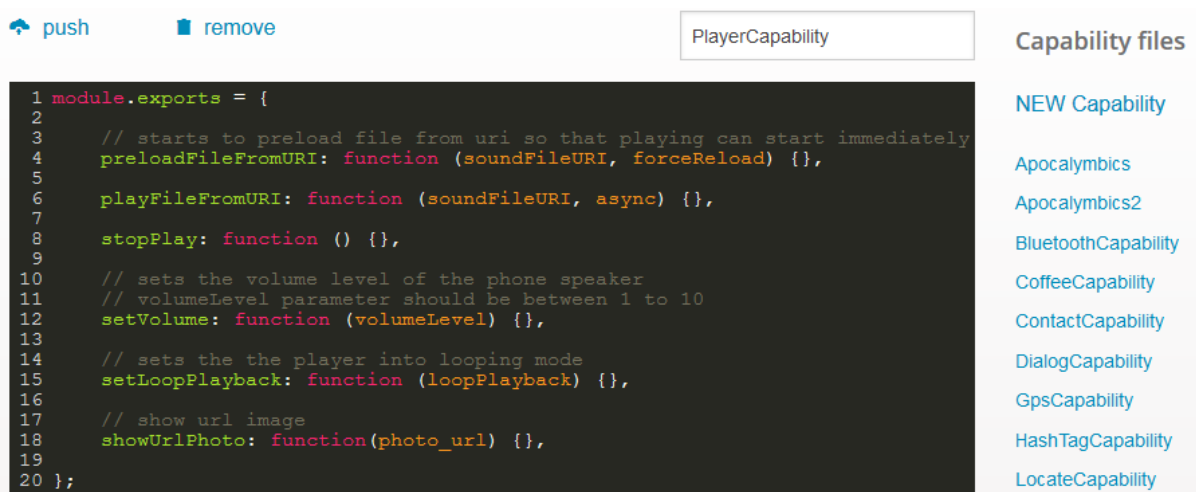


Figure 17. Capabilities interface example, in this case with the *PlayerCapability* loaded.

In this interface we can define all the functionality for this capability. We can add comments to explain them. All of the functions defined in the interface must be implemented in the device [Figure 18].

```
public JSONObject setVolume(JSONArray JSONmethodcallParameters) throws Exception {
    JSONObject retVal = new JSONObject();

    try {

        float volumeLevel = (Float)JSONmethodcallParameters.get(0);
        volumeLevel = volumeLevel / 10.0f;

        p("volumeLevel: " + Double.toString(volumeLevel));

        if(PlayerCapability.mediaPlayer != null) {
            p("inside iffi");
            PlayerCapability.mediaPlayer.setVolume(volumeLevel, volumeLevel);
            p("set is done..");
        }

    } catch (Exception e) {
        p("Error while setting volumeLevel for mediaplayer..");
        e.printStackTrace();
    }
    return retVal;
}
```

Figure 18. Implementation example of the function *setVolume* of the *PlayerCapability* in Android.

OrchestratorJS is available for Android, iOS, .NET Gadgeeter and Python using these capabilities. We only have to implement the capabilities in each device in their native language, and the Platform will call the functions.

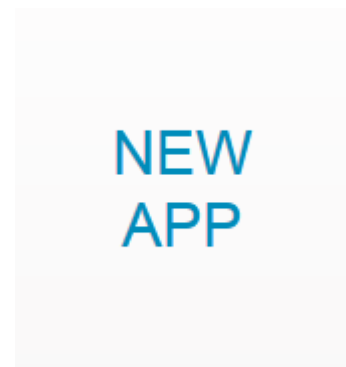
Actions in execution can be seen in the menu *Actions* → *Instances*. This view [Figure 19] shows all the running actions, giving us the possibility to kill them if needed. We can see the action name, identity and all the devices participating.

Action name	Identity	Participants		
CoffeeBreak	id1416498104577	Jose@Bq	Jose@Droxio	✖ kill

Figure 19. Action instances in execution.

Finally, the *Apps* menu [Figure 21] shows a grid with all the registered applications in the platform. These applications can be executed to provide a functionality to the connected devices.

We can run and stop applications directly from this view, and if we press the *edit* button we will see the usual editor where we can edit the application's code. The first button in the grid [NEW APP] allows to create a new application.



[push](#) [remove](#)
followMe App

```

1 var httprequest = require( '../tools.js' ).httprequest;
2 var pubsub      = require( '../tools.js' ).pubsub();
3 var tools      = require( '../tools.js' );
4 var Fiber      = require( 'fibers' );
5
6 var deviceHashTags = { };
7 var contactDetails = { };
8 var sendedMap = { };
9
10 function executeActionDevice( actionName, parameters )
11 {
12   var params = {
13     actionName: actionName,
14     parameters: parameters
15   };
16
17   httprequest( {
18     uri: '/api/1/actioninstance',
19     method: "POST",
20     form: params
21   }, function( error, response, body ) {} );
22 }
23
24 module.exports = {
25
26   settings: { },
27
28   logic: function()
29   {
30     // Get the "online" status. This will be executed each time a device is connected.

```

Write your app description here and hit save.

[start](#) [apps](#) [stop](#)

Figure 20. Example of the editor showing part of the followMe App's code, one of my proof of concept applications.

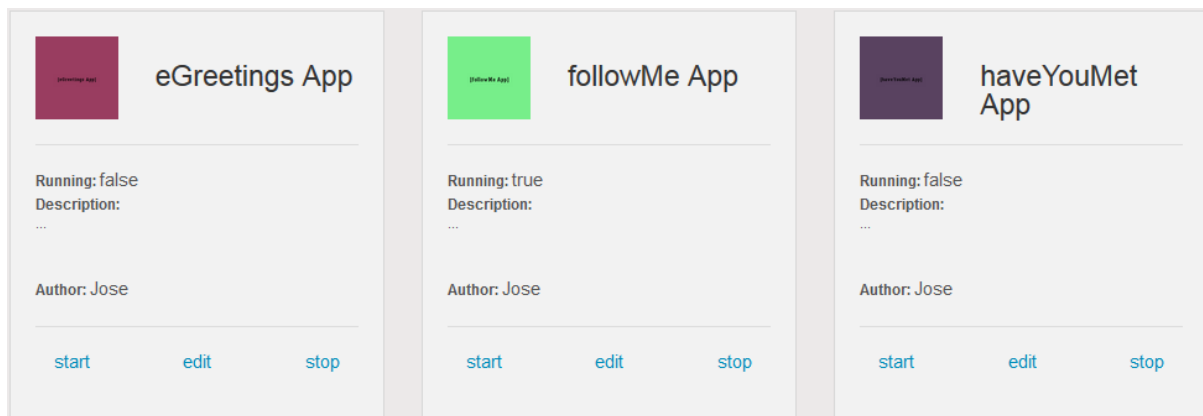


Figure 21. Grid view showing three applications. The *followMe App* is currently running. We can start and stop applications easily from this view.

REST API

The **Orchestrator Core** allows managing resources remotely through its REST API. The API supports posting apps, defining or triggering actions, changing device states, etc. The developers can also update the context data of a device with a single method call inside the capability implementation.

Actions

To create a new action we can send a POST message to the specific API. For example using *curl* would be something similar to this:

```
curl -i -X POST http://localhost:9000/api/1/action/MyAction -H "Content-Type: application/xml" --data-binary @D:/actions/my_action.js
```

This command will upload the action D:/actions/my_action.js to the server with the name MyAction.

To delete an existing action we can send a DELETE message. For example using *curl* to delete an action called myAction it would be like this:

```
curl -X DELETE http://localhost:9000/api/1/action/MyAction
```

Instances

We can also trigger an action as we do using the Web interface. For example, to execute the action myAction in the device called me@myDevice:

```
curl -X POST http://localhost:9000/api/1/actioninstance -d '{"actionName": "myAction", "parameters": [ "device:me@myDevice" ] }' -H "Content-Type: application/json"
```

Note: *The only parameters we have to change are the underlined.*

The REST API will send us a response with the status of the request. In the last request we can receive several messages:

- *“Cannot find module XXXX”*, indicates that the specified action is not present in the platform. Check if the name is correct.
- *“Not all devices found”*, which indicates that the specified devices are not registered in the platform.
- *“Device with id: me@myDevice not connected!”*, when the device is not connected to the platform. Maybe the connection was lost.
- *An action Id*, like for example: id1416531581346. If you receive this response everything went well and your action is currently in execution. This id is the action Id, which can be used to see the status or to kill it.

We can also kill an action using the action Id using a DELETE request:

```
curl -X DELETE http://localhost:9000/api/1/actioninstance/id1416531581346
```

Capabilities

In the same way we can create or modify capabilities, specifying the file to upload with the code of the capability's interface:

```
curl -i -X POST http://localhost:9000/api/1/capability/MyCapability -H "Content-Type: application/xml" --data-binary @D:/caps/my_capability.js
```

Devices

And finally, we can register new devices in the platform. This can be useful to avoid the user having to enter in the website and register the device manually. For example, we can check if it's the first time we run the application in an Android decide to register it automatically into the server.

```
curl -X POST http://localhost:9000/api/1/device -d '{ "identity": "me@myDevice", "bluetoothMAC": "AA:BB:CC:DD:EE:FF", "name": "me", "type": "android", "capabilities": [ "TalkingDevice", "Apocalymbics", "PlayerDevice", "UrlScreen", "TestDevice" ] }' -H "Content-Type: application/json"
```

In this example we are registering the device with identity me@myDevice, which has the Bluetooth address AA:BB:CC:DD:EE:FF. The device belongs to the user me, so it will be registered to that account. It is an Android device and has the selected 5 capabilities enabled.

Using this REST API we can implement our custom client in any platform capable to send HTTP requests, so the potential functionality of the platform can be extended to almost any platform. OrchestratorJS is available for Android, iOS, .NET Gadgeteer and Python. Also, web-based client can be embedded to web sites, acting as *virtual devices*. This makes possible the integration with Internet of Things and Internet of People.

RAW Context Data

Using the URL `http://localhost:9000/api/1/devices` we can see all the information stored in the platform about all the registered devices [Figure 22]. The output is in JSON format and can be useful for debugging purposes but also to parse the information and read it from a device, an application or an action.

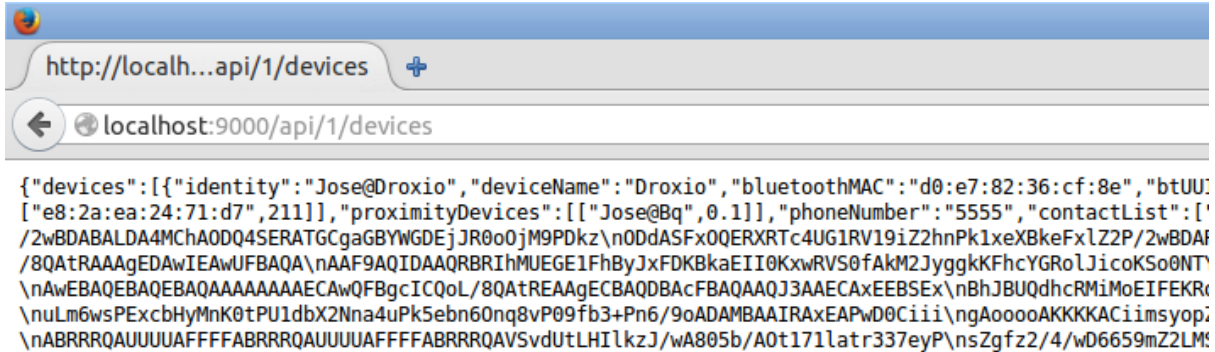
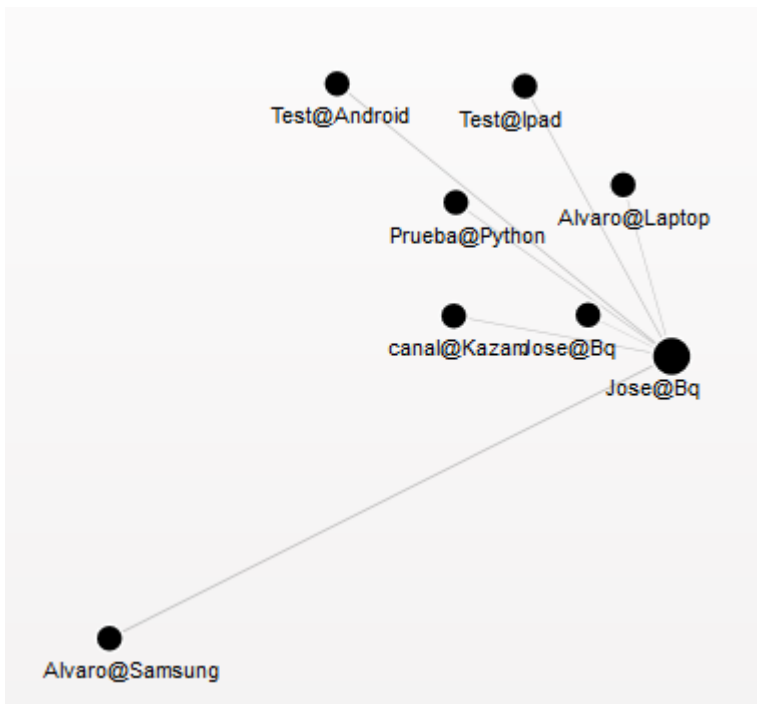


Figure 22. Example of all the context data stored in the platform.

This file includes information like the identity, Bluetooth address, username and last connection to the platform, and all the enabled capabilities. Also all the context data is shown, including the proximity of nearby devices. This can be parsed by the server or application if is necessary to read specific data. I use this file to get some information in of my test applications, we will see how it is done.

Proximity Graph

The proximity graph shows all the nearby devices in a comfortable graph view.



Devices are represented as nodes and the edges represent the relative distance of two devices.

The biggest node is the current device, and this graph shows only the devices in its proximity.

This view [Figure 23] can be accessed from the *Device Monitor* view, which we saw before. It can be useful to know an approximation of how devices can interact in a proximity based application.

Figure 23. Proximity Graph of the device called *Jose@Bq*.

Getting things work

As we have seen, OrchetratorJS has *Users*, *Devices*, *Capabilities*, *Actions* and *Applications* but, how they work together?

Everything is connected in the platform, users are connected to devices, which have capabilities enabled by the user. [Figure 24] shows the main behaviour of the Platform:

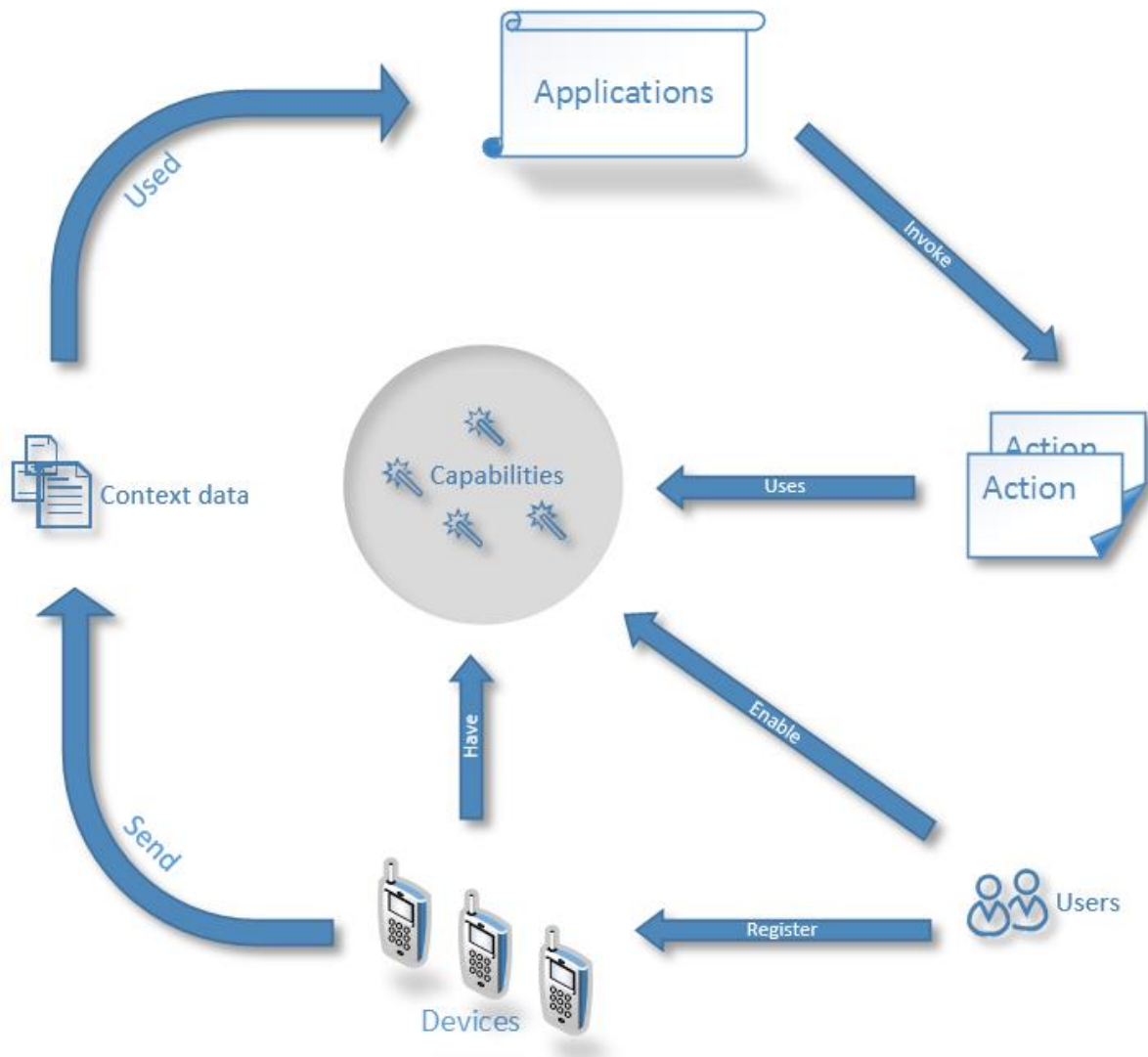


Figure 24. Basic behaviour of OrchestratorJS Platform.

- **Users** can register devices in the platform and enable a set of capabilities in each device.
- **Devices** have a set of capabilities implemented in their native language. The set of implemented capabilities can be different in each device, and not all of them have to be enabled by the user. Only enabled capabilities can be used, but enabled capabilities must be implemented in the device¹.

¹ This is one functionality I have implemented in the platform.

- **Context Data** is the information sent by the device to the platform. This information can be geolocation data, Bluetooth proximity, or data from the user's sociological profile. The context data can be sent automatically by the device, manually when the user performs an action, or triggered by an action executed in the server.
- **Capabilities** are the interfaces that allows the server to execute code in the devices. They are only an interface in the platform, but this interface must be fulfilled by the implementation in the device. Capabilities can be enabled in the client-side and in server-side, and only the capabilities enabled in both sides will be executed. This allows a special two-steps protection to the user, because both server and client will reject any unauthorized request.
- **Actions** are the basic piece to program in server-side. An action can be a simple action like sending a notification to the device, or a complex action involving some devices that interact between them. Actions can execute code on the server machine, to execute code on the device they have to use capabilities. Here is where enabling and disabling capabilities is useful to the users.
- **Applications** are a set of actions executed in a specific order. An application can be very complex and involve a big set of actions (which will involve a set of capabilities). Applications can also read the context data sent by the devices, and use that information to call actions. The context data can be read using **observers**, which will trigger a set of instructions when a specific type of context data arrives. Observers are very useful to avoid polling the context data to know when it changes¹.

One of the most important characteristics is the possibility to create observers, which will listen for a specific context data type. The application can trigger actions when a defined context data arrives to the application. This avoid the need of constantly poll the context data to see if new data has arrived.

```
// Example: pubsub observer for monitoring device online state
pubsub.on( 'online', function( contextValue, deviceIdentity ) {
  if ( contextValue !== true )
    return;
}
```

Figure 25. Example of an observer listening the context data called *online*. This context data is automatically sent by the client when it connects to the platform.

¹ The original OrchestratorJS Platform didn't inform when the same context data arrives. I have changed that behaviour because sometimes I need to know when the data arrives to the platform, regardless if it is the same or not.

If we need to read information from a website we can use the class *request*. Invoking actions is done with the *httprequest* class and calling the REST API of the platform.

```
// Example how an action can be triggered
var params = {
  actionName: 'MyAction',
  parameters: [ 'device:' + settings.someDeviceId ]
};

httprequest( {
  uri: '/api/1/actioninstance',
  method: "POST",
  form: params
}, function( error, response, body ) {} );
} );
```

Figure 26. Calling an action from an application. Invoking an action is achieved sending the corresponding POST message to the REST API.

We can also create and manage *Fibers*, which will allow us to create periodic actions, poll websites, and check if any user has request information, etc. A Fiber is basically as a Thread in any conventional programming language.

In the same way, we can use any function and library in NodeJS. The possibilities are endless: access a database, create sockets, connect to a web service, etc.

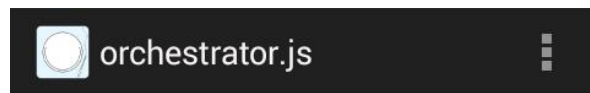
```
// Example: periodic loop e.g. for polling a website
Fiber( function() {
  // runs every ten seconds until the app gets stopped
  while ( true ) {
    console.log( 'tick' );
    tools.sleep( 10 );
  }
} ).run();
```

Figure 27. A basic *Fiber* printing a message each 10 seconds while the application is running.

Android Client

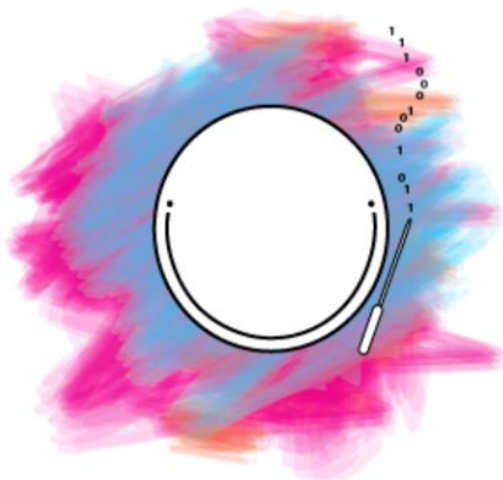
Now we understand how the server works, let's see how the Android client connects to the server, manages capabilities and executes the actions sent by the platform.

The main class in Android is the *OrchestratorJSActivity.java*. This class contains the basic functionality of the client: connection to the server, management of capabilities, invocation of actions, settings and configurations. All the classes must be inside the main package *com.ojs*.



Connect

Disconnect



The main class will execute when we open the application, it has a minimized view with two small buttons, to connect and disconnect. The first time we execute the application it probably won't connect to our server because it needs to be configured. In chapter 4 we will see how to install and configure everything.

When the device connects to the platform a small message appears, if the connection is lost it will be also noticed with a small message at the bottom of the screen. The device will try to load all the enabled capabilities.

Capabilities are differentiated through packages, each package means a different capability.

All the capabilities must be in a package called *com.ojs.capabilities.myCapability* and they must implement the public method *initCapability(Context applicationContext)*.

```
1 package com.ojs.capabilities.hashTagCapability;
2
3
4+ import com.ojs.OrchestratorJsActivity;
9
10
11 public class HashTagCapability {
12
13     private static Context applicationContext_;
14
15- public void initCapability( Context applicationContext )
16     {
17         HashTagCapability.applicationContext_ = applicationContext;
18     }
--
```

Figure 28. *initCapability* method from the *hashTagCapability*.

Note: The name of the package and the capability class are very important. The client will try to locate the capability in a package with the same name as the capability in the server, but with the first letter in lowercase. Also the name of the class that implements the capability must have exactly the same name as in the server's capability interface.

As we can see in [Figure 28], the capability name is *HashTagCapability*, which is the name in the server's capability interface. The name of the class implementing that capability must be the same, and the name the package is *hashTagCapability*, with the same letter in lowercase and inside the package *com.ojs.capabilities*.

The capability must also implements the method *initCapability* as in the figure. In one of these steps fail, the capability will be automatically disabled from the client¹.

Once this is accomplished, the capability will be able to load in the device and it is ready to execute the actions sent by the server. The capability should implement all the functions defined by the server's capability interface. If one of the functions is not implemented, the device will send an exception to the server when an action tries to trigger the method.

Usually a capability will create a new Activity in the device, this is not too much different as creating a normal activity in a usual Android application. Methods in capabilities can have arguments to receive information from the server, in this example I have the method *showNotification*, which receives a *title* and a *message*. Both arguments are sent to the new activity through an *Intent*.

```
public void showNotification( String title, String message ) throws Exception
{
    Log.d(OrchestratorJsActivity.TAG, "showing notification!");

    Intent i = new Intent(NotificationCapability.applicationContext_, NotificationCapabilityActivity.class);
    i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

    i.putExtra("notificationTitle", title);
    i.putExtra("notificationMessage", message);

    NotificationCapability.applicationContext_.startActivity(i);
}
```

Figure 29. Implementation code in Java for the method *showNotification* of *NotificationCapability*.

Then, we can read the Intent in the created activity and use the data properly.

```
Intent args = getIntent();

String title = args.getStringExtra("notificationTitle");
String message = args.getStringExtra("notificationMessage");
```

¹ This is one of the features I have implemented.

```
1
2 module.exports = {
3
4   showNotification: function (title, message) {},
5   showNotificationTimeout: function (title, message, seconds) {},
6   showNotificationJSON: function (title, message, jsondata) {},
7   showToast: function (message, seconds) {},
8
9 };
10
```

Figure 30. NotificationCapability interface deployed in the server. The capability installed in the client should implement all of these functions.

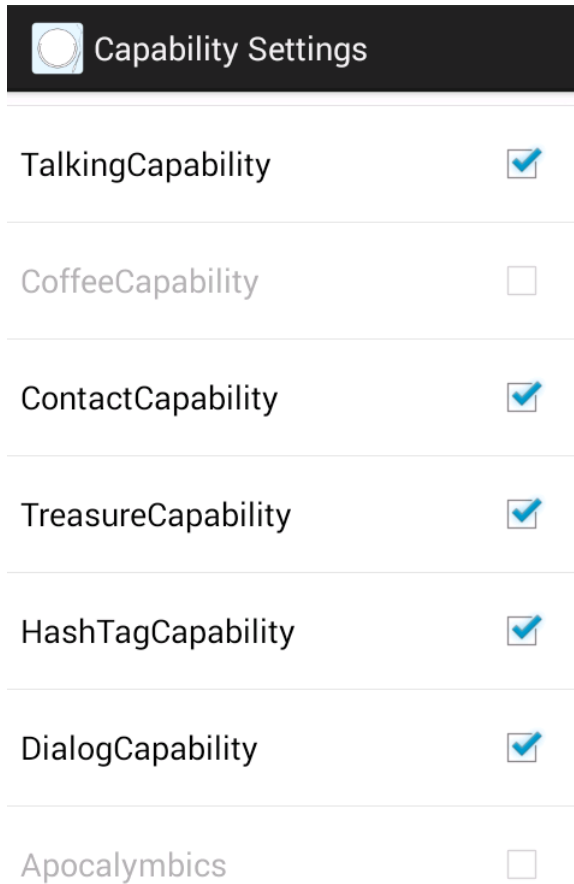
Changes and additional features

To accomplish the target of this project and be able to develop the five test applications I have to make some changes and develop new features in the platform. I will explain here what these changes are and how they work:

- Triggering observers when the arriving data is the same: In the original platform when context data arrives to the platform, it didn't trigger the observer if the content was exactly the same as the platform already had. I have changed this behaviour because sometimes I need to trigger the observer. For example, in some applications the server asks the user to introduce data, like the phone number. When the user introduces this data the application can start, however, if the user introduced the same phone number as the existing one in the platform (what is very probable), the application didn't trigger the action.
- Changed the RSSI Bluetooth values: I have created 5 different ranges to determine the distance of a device based in the RSSI Bluetooth value. I changed this values because it would be better if we have a better approximation of the distance. Also the original values had negative values (and my devices send a value between 0 and 255), and sometimes the proximity graph didn't draw correctly because values were too much high.

```
if( rssi > 195 )
  return 0.1;
else if( rssi > 140 )
  return 0.25;
else if( rssi > 90 )
  return 0.5;
else if( rssi > 40 )
  return 0.75;
else
  return 1.0;
```
- Added support for Bluetooth proximity in the Android client: originally when I started my project the Android client had no Bluetooth support. I have implemented it in my version. Later, a Bluetooth service was integrated in the main platform. However, both services can coexist in the application. I had implemented the Bluetooth support as a capability, what the user can enable or disable from the device. Server applications call this capability when an update of the Bluetooth proximity is needed.

- Automatically disable unimplemented capabilities in the Android client: if a new capability interface is deployed in the server, probably our installed Android application has no support for it. I have changed the behaviour of how capabilities are activated, so the user won't be able to enable capabilities not implemented in the client. Uninstalled capabilities can't be selected in the menu [Figure 31].



- Handle disconnection from the server to finish capabilities: in my implementation I use some capabilities as a service, this services can, for example, constantly update Bluetooth proximity or GPS position. I have added a trigger when the connection with the server is lost, to allow finish some operations that have no much sense to be in execution once the client is disconnected.
- Improved debug messaged and clean-up: sometimes the console was difficult to read because some debug messages appear really fast. I have done small improvements in this aspect to be able to read the debug console in an easier way. I have also added some error messages to the Android client.

Figure 31. Not installed applications can't be enabled.

- Bug fixes: some small fixes and improvements in the platform and the client, to avoid crashes and random disconnections of the users.

Goals

The main goal of OJS is to support Social Devices concept, this means that the middleware is targeted to facilitate and increase interactions between people and devices when they meet face-to-face. The objective is to coordinate devices wherever they are in the proximity of each other.

Service compositions may be different as another platform, the main separation is that the devices can independently socialize, starting to resemble humans, and getting similar characteristics.

Apart the main goal, there is a set of four goals the system should meet [8] (Bronsted, Hansen, & Ingstrup, Service composition issues in pervasive computing, 2010):

Context Awareness

The system should be able to respond context changes by providing information or executing a command. OJS is specially designed to compose the services in an ad hoc manner as the interactions (actions) are very momentary by their nature. Furthermore, OJS can report and store heterogeneous and sparse context data. As we have seen, the platform supports the implementation of observers, which can be used for reporting and receiving notifications to react these changes in the context data. The developers can define their own logic to do that, and decide how to inform the user.

Managing Contingencies

The second goal is to manage contingencies. OJS actions are JavaScript modules that contain the interaction definition (*body*), but 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 the action body, or in device-side code. OJS automatically relays exceptions from device to the server.

- The developer can use the error method to trace what happened, and react accordingly, for example informing the user. If the error method is not 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. For example, we can define as a precondition that a certain device must be connected, or the context data has a special value. So the action won't be executed until these preconditions are not satisfied.
- 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. The default handler does not kill the action.

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. One of the main concepts of the *Action-Oriented Programming Language* are capabilities which are used for describing what a device can do. This makes it straightforward for the app developer to pick appropriate devices for the interactions and set their roles.

The contextual information stored in registry, as well as the information in social media and other party services, can be accessed within app 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 taking advantage of many of the everyday physical objects to OJS, which also supports the goal as it helps to build custom devices to some specific purposes.

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 application he wants to run and what interactions his phone can take part to, by enabling or disabling device capabilities.

If the user never wants the phone to talk, he can easily disable that capability. If the device is taking part in an undesired interaction, he can easily stop the application.

Example: The Coffee Break Machine

As we introduced before, there is an example of Coffee Machine developed using the Social Devices concept, and integrated in the Orchestrator.js Platform [25]. Here we will see a small abstract of how this prototype was developed. I think this is a very interesting prototype because it illustrates how the platform can be integrated to apply the *Internet of Things* and the *Internet of People* models.

In this implementation, Bob is a worker that takes the bus every morning to his office. His mobile phone detects when he is in the bus and vibrates to ask if he wants to have a coffee when he arrives to the office.

The phone sends data from sensors to *Device Registry* of the OrchestratorJS Platform, for example with GPS location or nearby WIFI networks, to know the user is leaving home and way to the office.

```

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

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

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

20                 httprequest( {
                    uri: '/api/1/action/MakeCoffee',
                    method: 'POST',
                    form: { [ settings.coffeeMachineId,
                             deviceIdentity ] }

25                 } );
            } );

    pubsub.on( 'coffeeReady',
              // observer code omitted for brevity (20 loc)
30             } );
        }
    };

```

Figure 32. Partial code of the Coffee Machine Prototype.

The user receives a notification asking if he wants to have the coffee, he can always decline having a coffee, or just ignore the notification and it will vanish after a short period. The contextual data can be stored in any format, and then be processed by the app. Maybe it's necessary to translate the GPS coordinates to address, or calculate the distance and speed to know when Bob will arrive the office.

The Device Registry allows storing contextual data in any format and publishing it through *pub/sub* API. Coffee Break app has two observers that get notified when user's *location* changes, and when *coffee is ready* [Figure 33]. In lines 13 to 18 the application checks if the user hasn't yet been asked about having coffee.

When all the conditions are met, the application calls the REST API using the *httprequest* library to trigger the *MakeCoffee* action. The second observer [line 28] checks when the *coffeeReady* context data arrives from the Coffee Machine, which indicates that the coffee is ready. The application will then trigger the *InviteForCoffee* action to invite Bob and all his colleagues at the office to have a coffee together in the coffee room.

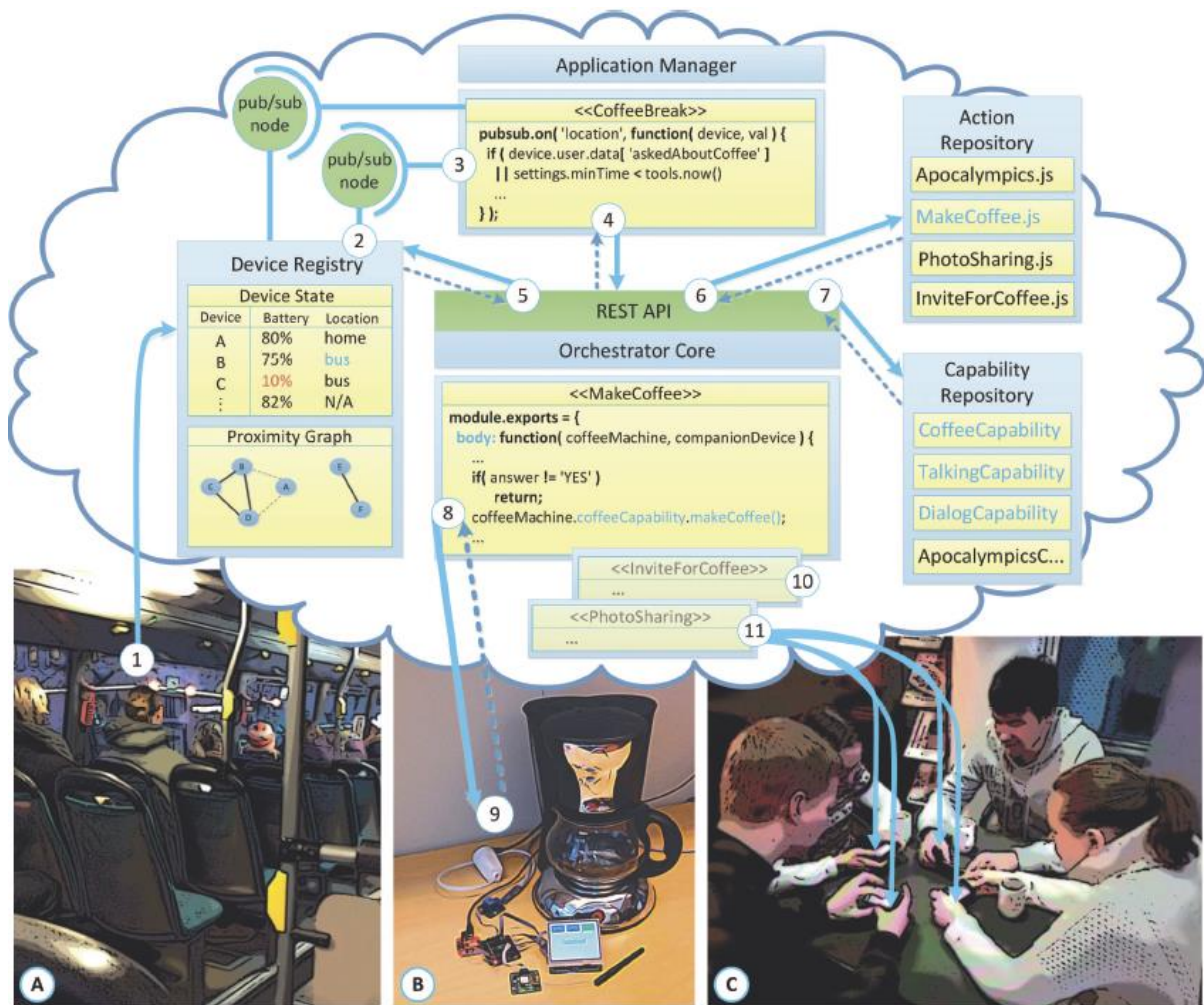


Figure 33. OrchestratorJS and all the components used in the Coffee Machine implementation.

The application was developed using the Web IDE and it is a small example of what can be done with the platform. This prototype was developed using the platform Framework for .NET Gadgeeter. This framework allows to make the technology available for different kinds of hardware devices, which can help to integrate with IoT.

Chapter 3

Related technologies

Android Programming

Android is a mobile operating system based on the Linux kernel and developed by Google. It's nowadays the most popular operating system for mobile devices and it's designed primarily for touchscreen devices such as smartphones or tablets. Android has currently more than 1 million Android Apps published, and there are more than 1 billion active Android users per month.

The source code is released by Google under an Open Source license, what makes it a popular choice for technology companies which require a low-cost and customizable operating system for the devices. Android, with its open source nature, has encouraged a large community of developers and enthusiasts, who collaborates in the development, making new applications or adding new features to the operating system.

Architecture

Android consists of a kernel of Linux with architectural changes implemented by Google. There are some middleware libraries and APIs written in C on top of the Linux kernel, and an application framework to execute application software compatible with Java. The Java code executed is executed by a special Java Virtual Machine (called *Dalvik Virtual Machine*) which translates Java bytecode.

Linux Kernel

The *Linux Kernel* is positioned at the bottom of the Android architecture and provides the abstraction level between the device hardware and the Android libraries. Android uses only the Linux kernel, which provides multitasking, process and energy management, network capabilities, memory services, audio and display.

Libraries and Android Runtime

Each application running on an Android device creates its own instance of the *Dalvik Virtual Machine*. This way has a number of advantages:

- Applications are fully sandboxed, so they cannot interfere with the operating system or other applications nor access the device hardware.
- It can be used in any hardware.

The core libraries are Java Runtime Libraries developed specifically for the Android system and they provide support for tasks such as string handling, networking and file manipulation. These libraries are called *Java Interoperability Libraries*.

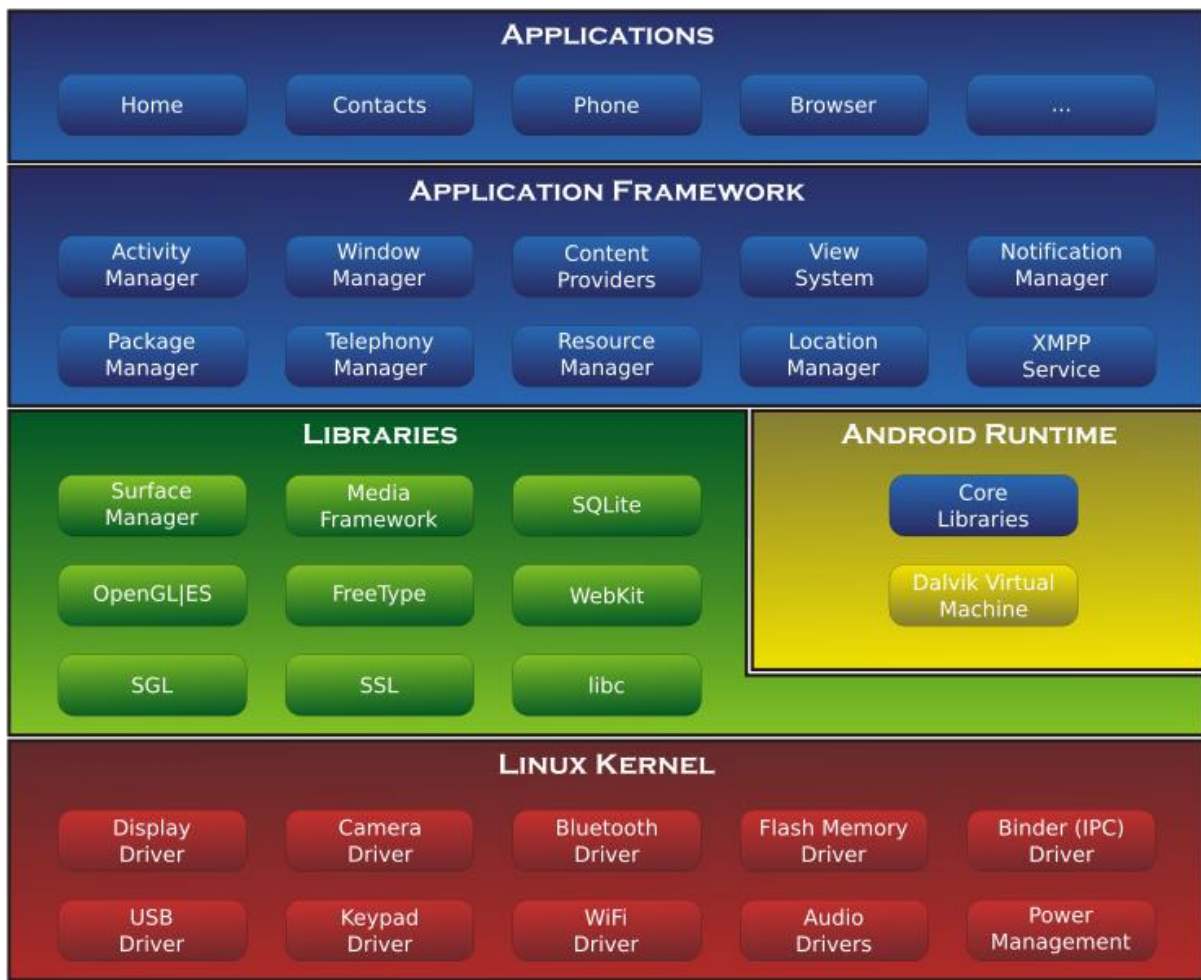


Figure 34. Android's architecture diagram.

The Android libraries are a set of Java-based libraries that are specific to Android devices. These libraries includes the support for graphics, access to databases, OpenGL, operating system services, web-browsing and access to hardware capabilities like the accelerometer and light sensor.

There are also many C/C++ libraries, some of them are called from Java when we try to access to some capabilities. For example the OpenGL Library is written in C++, and it is called from Java Libraries when we access to it. Functions for 2D and 3D, the Secure Sockets Layer (SSL), video and audio are some examples of native libraries written in C++.

Application Framework

The Application Framework is a set of services to prepare the needed environment in which Android applications can be run and managed. Android applications are constructed under the concept of components being reusable, interchangeable and replaceable.

These are the key services of the Android Framework [3]:

- **Activity Manager:** Controls all aspects of the application lifecycle and activity stack.
- **Content Providers:** Allows applications to publish and share data with other applications.
- **Resource Manager:** Provides access to non-code embedded resources such as strings, colour settings and user interface layouts.
- **Notifications Manager:** Allows applications to display alerts and notifications to the user.
- **View System:** An extensible set of views used to create application user interfaces.
- **Package Manager:** The system by which applications are able to find out information about other applications currently installed on the device.
- **Telephony Manager:** Provides information to the application about the telephony services available on the device such as status and subscriber information.
- **Location Manager:** Provides access to the location services allowing an application to receive updates about location changes.

Applications

The applications are located at the top of the Android software stack. Applications can be the native applications preinstalled with the particular Android implementation, like the web browser and email applications, and the third party applications installed by the user.

Android Virtual Devices

Android Virtual Devices (AVDs) are essentially emulators that allow Android applications to be executed without a physical Android device. The virtual device can be configured to emulate any kind of device and with particular hardware features (screen size, memory, camera, etc.). AVDs are very useful for Android developers, because it facilitates the testing and debugging process.

However, the Android Virtual Devices has some hardware limitations [5]. For example, it is not possible to emulate any Bluetooth device. This limitation provokes that the Social Devices Platform and OrchestratorJS Platform cannot be used in this kind of emulated devices.

Android Virtual Devices can be used to debug some aspects of the platform, but it won't be possible to test all the capabilities without at least two real android devices.

Android Applications

Android applications are written in Java, however it takes the concept of re-usable components to a higher level. The applications are created through one or more components known as Activities that consists in a single module of functionality which usually is related with a single user interface screen.

Activities are intended to be fully reusable and interchangeable with different applications. They are created as subclasses of the Android Activity class and must be implemented to be entirely independent of other activities.

Communication between Activities is not possible. There is a mechanism called Intents, which can be used to launch another activity and send data to it. Broadcast Intents are a type of Intent that can be used to send a message to all the applications currently listening. It is a system wide Intent and can be used to indicate changes in the device, such as the status of the battery, if the screen is turned on or off, or the device is charging. Applications can create Broadcast Receivers, which makes them able to respond to Broadcast Intents.

There are processes that can be run in the background without user interface, they are called Android Services and can be started and managed from Activities. These services are ideal for situations where an application needs to continue performing tasks but is not necessary a visible user interface. However services can notify the user using notification messages (*Toasts*) and they can use Intents to run other Activities.

Content Providers are another mechanism for sharing data between applications. They allow to provide access to its data to other applications. The access to the data is provided via a *Universal Resource Identifier* (URI) defined by the Content Provider. This Content Providers may be located using a *Content Resolver*.

The *Application Manifest* comprises all the elements that conforms the application. It consists in a XML file that contains the activities, services, broadcast receivers, data providers and permissions of the application.

Applications can have *Application Resources*, which are additional files that can be used to create the user interface, such as images, strings, fonts, videos, sounds, etc. This resources are typically saved in the /res subdirectory of the application project.

```
nal int advanced_pref_default_orhestrator_host=0x7f06000d;
nal int advanced_pref_default_orhestrator_port=0x7f06000e;
nal int advanced_pref_default_proximity_host=0x7f060014;
nal int advanced_pref_default_proximity_port=0x7f060016;
nal int advanced_pref_header_orhestrator=0x7f06000d;
nal int advanced_pref_header_proximity=0x7f060012;
```

Figure 35. Example of R class with some references.

Finally, the *Application Context* is a context that can be used in the application code to gain access to the application resources in runtime. A class named R is created, which contains references to all the application resources. Also there are a lot of resources that can be accessed to gather information and make changes to the application's environment at runtime.

Android Processes

Within an Android system, the current state of a process is defined by the highest ranking active component in the application. The application can be changed from one state to another by the operating system in some situations. There are five states:



Figure 36. Possible states of a component or application.

- **Foreground Process:** They have the highest level of priority and only one process can be executing in this state, usually the current opened application in which the user is doing something. The system will never terminate this process in case of low memory.
- **Visible Process:** This is the state of a program that is currently running and visible on the screen, but there is another activity or dialog in the foreground where the user is interacting.
- **Service Process:** Processes that contain a started Service and currently running.
- **Background Process:** Processes that contain one or more activities not currently visible to the user and without a Service Process running. These processes have a high risk of termination in case of low memory. The background processes will be terminated in chronological order, so old unused processes are killed first.
- **Empty Process:** These are processes that no longer contain any active applications and are waiting to host a new launched application.

In case of dependencies inter process, Android will take them in consideration, so a Service that is serving to another process with higher priority will get its priority. As a basic rule, a process can never be ranked lower than another process that it is currently serving.

Activity Lifecycle

Each application that is running on an Android device has an *Activity Stack* maintained by the system. When the application is launched, the first activity executed is placed onto the stack. The next activities launched will be placed on the top of the stack, and the previous activity is pushed down. The activity at the top of the stack is the active or running activity.

When the active activity exists, it is popped out of the stack, and the activity located immediately beneath it becomes the current active activity, at the top of the stack. An activity can also be closed by the user when he presses the back button on the screen. When this happens, the current activity is closed and removed from the stack, so the next activity will be the active.

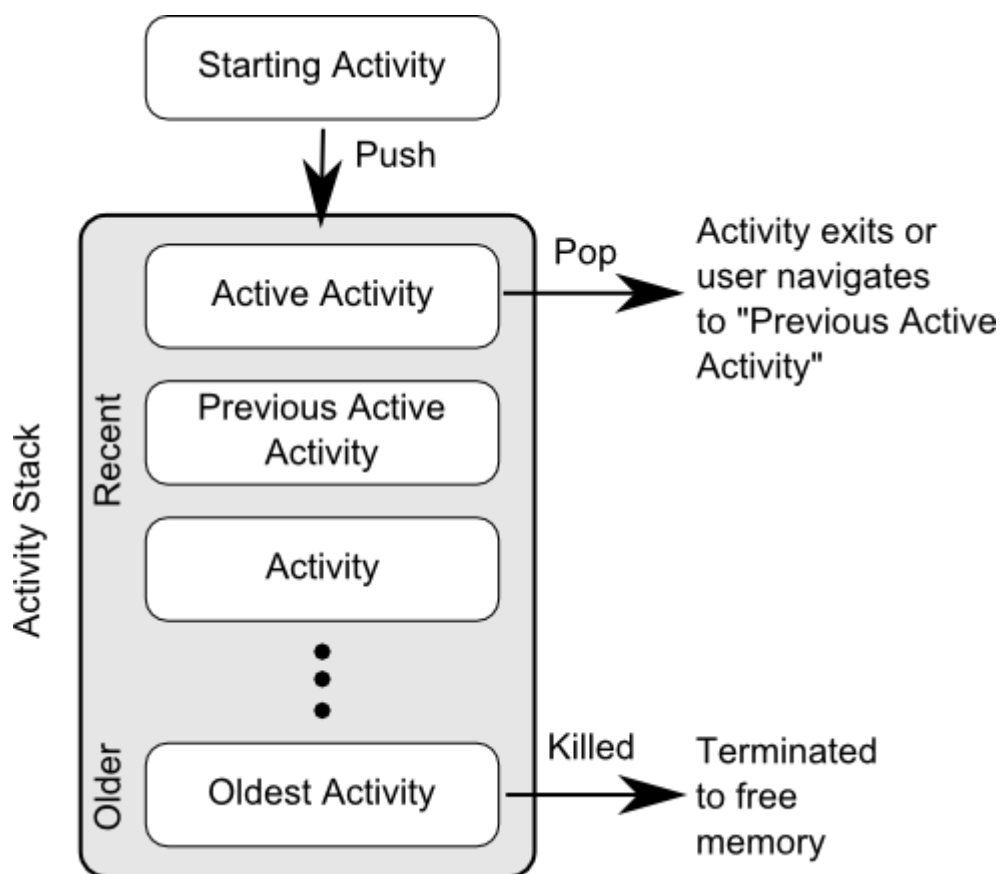


Figure 37. A visual representation of the Android Stack.

As mentioned, the stack is working like a Last-In-First-Out (LIFO) stack of activities, with the running activity always at the top of the stack.

In case of the device memory is too low, the system will kill activities, starting with those at the bottom of the stack.

Activity States

An activity can be in one of the different states during the course of its execution within an application:

- **Active / Running:** The activity is at the top of the Activity Stack and is the foreground task visible on the device screen, with the focus and interacting with the user. This would be the last activity to be terminated in case the device is low of resources.
- **Paused:** The activity is visible to the user, but does not currently have focus, for example because there is a dialog or another small activity on top asking for an action from the user. These paused activities are hold in memory, and can be quickly restored when gains the focus again.
- **Stopped:** The activity is not visible to the user and it is retained in memory like in paused state. However activities with this state have a higher risk of be terminated in low memory situations.
- **Killed:** The activity has been terminated by the runtime system to free up memory. The activity is no longer present in the Activity Stack and has to be restarted.

Usually, when the runtime system is going to change the state of an activity, it sends a notification before the change. This allows the activity to react accordingly. For example, we can save all the work before being killed.

MongoDB

MongoDB [31] is a cross-platform Documented-Oriented Database¹. It is classified as a NoSQL database and the data is saved in a structure of JSON-like documents with dynamic schemas, making the integration of data in certain types of applications easier and faster.

It was developed by the software company *10gen* (now MongoDB Inc.) in October 2007 as a component of a planned Platform-as-a-Service product. In 2009 it was shifted to an open source development model.

MongoDB focuses on flexibility, power, speed and ease of use [32]:

- **Flexibility:** MongoDB stores data in JSON documents, which is serialized to BSON (Binary JSON). JSON provides a rich data model that seamlessly maps to native programming language types, and the dynamic schema makes it easier to evolve your data model than with a system with enforced schemas such as a Relational Database.

¹ A Document-Oriented Database is a kind of NoSQL database designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data [13].

- **Power:** MongoDB provides a lot of the features of a traditional Relational Database Management System such as secondary indexes, dynamic queries, sorting, rich updates, upserts (update if exists, insert if it doesn't), and easy aggregation. This gives the wide functionality from a RDBMS, with the flexibility and scaling capability that the non-relational model allows.
- **Speed and scaling:** By keeping related data together in documents, queries can be much faster than in a relational database where related data is separated into multiple tables and then needs to be joined later. MongoDB also makes it easy to scale out your database. Auto-sharding¹ allows to scale your cluster linearly by adding more machines. It is possible to increase capacity without any downtime, which is very important on the web when load can increase suddenly and bringing down the website for extended maintenance can cost your business large amounts of revenue.
- **Ease of use:** MongoDB works hard to be very easy to install, configure, maintain, and use. To this end, MongoDB provides few configuration options, and instead tries to automatically do the "right thing" whenever possible. This means that MongoDB works right out of the box, and you can dive right into developing your application, instead of spending a lot of time fine-tuning obscure database configurations.

Some important features that MongoDB supports are:

- **Replication:** MongoDB provides high availability with replica sets. A replica set consists of two or more copies of the data. Each replica set member may act in the role of primary or secondary replica at any time. The primary replica performs all writes and reads by default. Secondary replicas maintain a copy of the data on the primary using built-in replication. When a primary replica fails, the replica set automatically conducts an election process to determine which secondary should become the primary. Secondaries can also perform read operations, but the data is eventually consistent by default.
- **Aggregation:** MapReduce can be used for batch processing of data and aggregation operations. The aggregation framework enables users to obtain the kind of results for which the SQL GROUP BY clause is used.
- **Server-side JavaScript execution:** JavaScript can be used in queries, aggregation functions (such as MapReduce), and sent directly to the database to be executed.
- **Capped collections:** MongoDB supports fixed-size collections called capped collections. This type of collection maintains insertion order and, once the specified size has been reached, behaves like a circular queue.

¹ [http://en.wikipedia.org/wiki/Shard_\(database_architecture\)](http://en.wikipedia.org/wiki/Shard_(database_architecture))

How it works

MongoDB can be managed in Linux from a command-line terminal. To connect to a running MongoDB server we simply execute the command:

```
mongo [--host localhost] [--port 21017]
```

Parameters *host* and *port* are optional, if they are not specified it will automatically connect to a database listening on port 21017 on the localhost interface.

To show a list of the databases existing in the server we can execute the command:

```
show dbs
```

We can switch to a database using the command *use* and the name of the database. Note that if the database doesn't exist it will create it automatically. That's why MongoDB is completely flexible DBMS:

```
use mydb
```

```
db
```

The last command *db* will show the current selected database. Now you are using the new created database called *mydb*. This new database is empty at this moment, and it won't be shown in the list of databases returned by the *show dbs* until any data is stored on it.

Let's insert some data in JSON (called documents). Creating a new document is done in the same way as we would do in JavaScript:

```
> doc1 = { name : "Jose", año : 2014 }
{ "name" : "Jose", "año" : 2014 }
> doc2 = { ciudad : "Málaga", año : 2014 }
{ "ciudad" : "Málaga", "año" : 2014 }
> █
```

Now we can insert this new documents into the database. We will insert them in a collection called *myTest*:

```
> db.myTest.insert(doc1)
> db.myTest.insert(doc2)
> show collections
myTest
system.indexes
> █
```

The *show collections* command will show all the collections existing in the database.

```
> db.myTest.find()
{ "_id" : ObjectId("5471e3ef37367ddf22f81e0d"), "name" : "Jose", "año" : 2014 }
{ "_id" : ObjectId("5471e3f937367ddf22f81e0e"), "ciudad" : "Málaga", "año" : 2014 }
4 }
> █
```

The *find* command will output all the documents existing in the specified collection.

We can specify arguments with the value of the keys, so it will return only the documents matching the specified search:

```
> db.myTest.find( { name : "Jose" } )  
{ "_id" : ObjectId("5471e3ef37367ddf22f81e0d"), "name" : "Jose", "año" : 2014 }  
>
```

Updating and deleting data is not very different or complicated. MongoDB provides also mechanisms to iterate through returned data from queries, called *Cursors*.

An important example of where MongoDB is very useful is combined with *Internet of Things* [33]. Internet of Things needs to store information from huge amounts of sensors and objects, such amount of data is too much for a relational database.

Bosch has built its Internet of Things suite on MongoDB, bringing the power of big data to a new range of Industrial Internet applications including manufacturing, automotive, retail, energy and many others.

Internet of Things is a big challenge for databases, each new generation of 'things' comes with new sensors, which create new data and new functionality requirements. Relational databases make it hard to incorporate new data and iterate on your data model. MongoDB can manage data of any structure, no matter how often it changes. You can ship new functionality without redesigning the database, allowing build apps that could never build before.

Also, relational databases are not designed to handle data from 40 billion devices, while MongoDB is built to scale out on commodity hardware.

Finally, MongoDB can analyse data of any structure directly within the database and in real time. In relational databases you probably need to process data and prepare for output in the correct format.

Node.js

Node.js [29] is a way of running JavaScript on the server, it allows to create Web applications using JavaScript entirely. Node.js is an interface to the V8 JavaScript runtime, the fast JavaScript interpreter implemented by Google that runs in the Chrome browser.

Conventional servers, like Apache, spawns a new thread for each connection to the server, but this doesn't scale well and it's very easy to quickly lose performance or the site goes down. With Node.js this is not the case, you will need to handle the core server architecture yourself. This can be overwhelming when you start out with Node.js, but there are modules to make this easier, and result will be a high performance web application.

JavaScript is an event-based language, so anything that happens on the server triggers a non-blocking event. Each new connection fires an event, received data from a form fires an event, requested data from the database fires an event. This means a

Node.js website will never lock up and can support tens of thousands of concurrent users [9].

In Node.js you don't have a main code that runs constantly in the background, you don't have to check if something has finished to continue, you will never wait for the response of a database request. In Node.js you specify functions that should be run of completion of something else, and that functions will execute when they are triggered by the specified event.

```
var http = require("http");
http.createServer(function (request, response) {
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  response.write('Hello World!');
  response.end();
}).listen(8080);
console.log("Server running!");
```

Figure 38. A basic *Hello World* program in Node.js.

[Figure 38] shows a Hello World program that simply prints the *Hello World!* message when a new connection is created in the port 8080. This basic example can help to understand how asynchronous class work.

Node.js is based in an event driven model, what is event-driven programming? I liked this description from a Princeton University Page [15] (Event-driven programming, 2014):

“In computer programming, event-driven programming or event-based programming is a programming paradigm in which the flow of the program is determined by events—i.e., sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads.

Event-driven programming can also be defined as an application architecture technique in which the application has a main loop which is clearly divided down to two sections: the first is event selection (or event detection), and the second is event handling. In embedded systems the same may be achieved using interrupts instead of a constantly running main loop; in that case the former portion of the architecture resides completely in hardware.

Event-driven programs can be written in any language, although the task is easier in languages that provide high-level abstractions, such as closures. Some integrated development environments provide code generation assistants that automate the most repetitive tasks required for event handling.”

To do a login system, we simply call the database query when the user sends the form data, Node.js sends the query to the database, but it won't wait for the response. When the response from the database arrives, we will have a different function triggered when the data arrives, which will send the result back to the user. This allows the server to execute instructions (for another user, for example) instead of wasting CPU cycles waiting for an IO response [52].

Event-driven programming is more complex than usual programming, however Node.js platform provides abstractions and modules that make it easier to create event-driven applications.

Node.js is single threaded, it can use only 1 CPU at any given time, but it provides functionality to have multiple processes listen on the same port. Each individual process can use a single CPU so you can scale out a multi-CPU server by using multiple processes. You can also take advantage of scaling across multiple host by using a reverse proxy, load balancing web server [20].

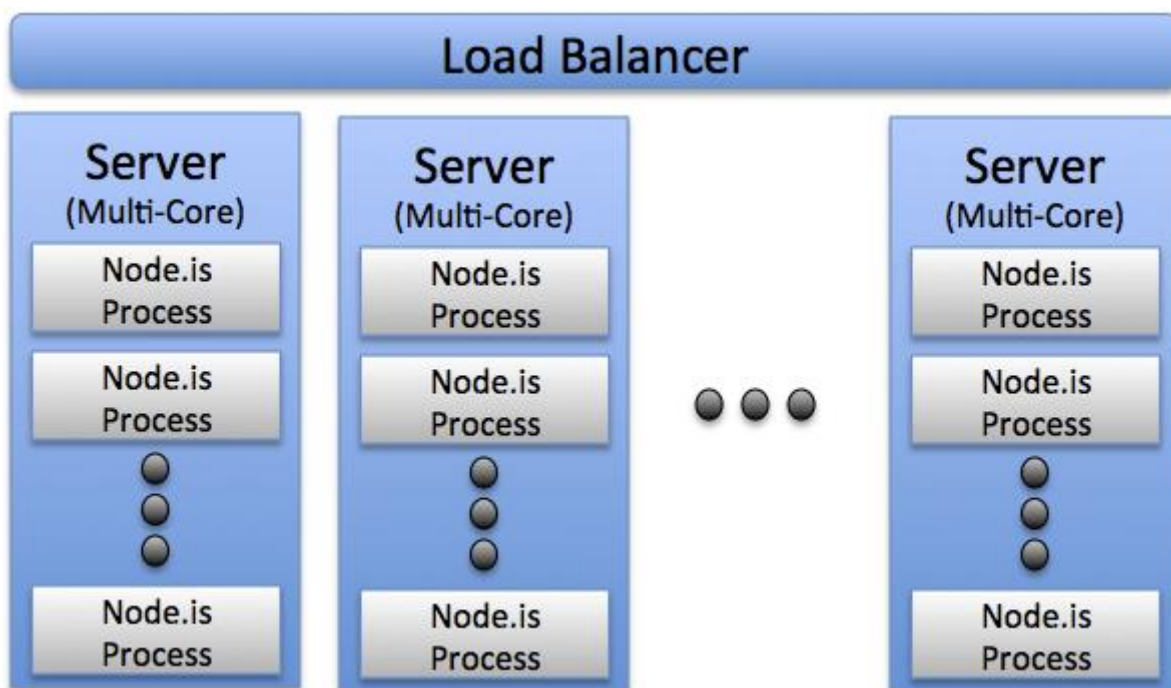


Figure 39. How a Node.js application scales on a per processor basis and across servers.

Node.js can also take a huge advantage of JSON, *JavaScript Object Notation*, which facilitates data transmission between the server and the application without the need of XML. JSON is lightweight, human-readable and can be parsed directly in JavaScript.

Node.js is also a good choice to develop RESTful services very efficiently and can handle thousands of requests concurrently, where PHP would just collapse.

The major downside to Node.js is that it is still in the early stages of development, meaning that some features are likely to change as development progresses. In fact, the documentation includes a stability index, which shows how risky use of each feature is currently [Figure 40].

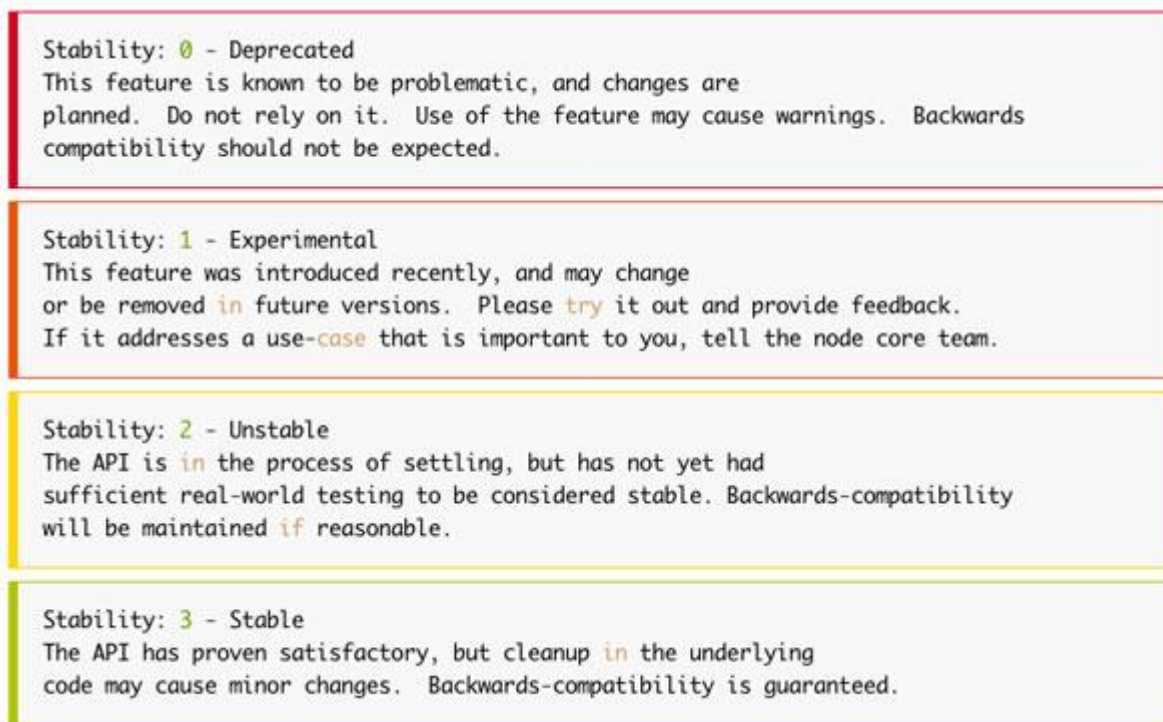


Figure 40. Stability index.

In conclusion, Node.js scales much better than other platforms: consumes less memory and is faster. The development can be complicated to people not familiarized with the event-driven programming model and with JavaScript, however is a good alternative to develop small and middle web applications with thousands of connected users as could be, for example, a RESTful service.

Integration with MongoDB

Node.js can be easily integrated with MongoDB using the package *MongoJS*, which lets you access MongoDB using an API that is extremely similar to MongoDB's JavaScript shell.

The package can be installed from NPM simply using the command:

```
npm install mongojs
```

Let's see some examples. First step would be the connection from Node.js to MongoDB:

```
var databaseUrl = "mydb";
var collections = ["users", "messages"];
var db = require("mongojs").connect(databaseUrl, collections);
```

This example will create a new connection called *db* connected to the database *mydb* using the default hostname and port (localhost and 21017). We can specify different values as:

```
var databaseUrl = "username:password@example.com/mydb";
```

The parameter *collections* can be used to specify the set of collections out application uses. This parameter is optional, but I recommend to set it because it will allow us to emulate a MongoDB JavaScript client like API within our Node.js application.

Now, to find a document the syntax is very similar to the MongoDB console:

```
db.users.find({sex: "female"}, function(err, users) {
  if ( err || !users) console.log("No female users found");
  else users.forEach( function(femaleUser) {
    console.log(femaleUser);
  } );
});
```

This will return all the female users. This query is not like usual queries in other different programming languages. We are not waiting until the result of the query, we are sending the query to the DB and specifying the function that will be executed when the database returns the result.

Saving a new document into the MongoDB database is also very similar:

```
db.users.save({email: "user@email.com", password: "testMongoDB", sex:
"male"}, function(err, saved) {
  if ( err || !saved ) console.log("User not saved");
  else console.log("User saved");
});
```

Finally, let's see how to update a document. In this case we are changing the password of the user *user@email.com*.

```
db.users.update({email: "user@email.com"}, {$set: {password:
"testUpdate"}}, function(err, updated) {
  if ( err || !updated ) console.log("User not updated");
  else console.log("User updated");
});
```

As you have probably realized, all the functions has a parameter *err*, which will allow us to determine if an error occurred, and a parameter with the result of the operation. In the first query, the parameter *users* will include the list of results.

Chapter 4

Getting started with Social Devices

In this chapter we will see how to install and configure the OrchestratorJS Platform and all the needed software to get the platform prepared to use and develop. This guide explains how to install the platform and all the required software in Ubuntu. I have used a Virtual Machine to install everything, but it is valid for any version and computer.

Note: *It's very important to keep the same version of some of the dependencies, especially the npm modules, as the newer versions have changed some names and functions and they are not backward compatible.*

Dependencies

My test server runs on Ubuntu 14.04 LTS, however should be no problem in run another Linux Distribution or version. However this tutorial and all the dependencies included in the CD-ROM are specific for this version, and could not be valid in any other version or distribution.

- Ubuntu 14.04 LTS.
- NodeJS v0.10.25.
- Npm 1.3.10.
- MongoDB Shell version 2.4.9.
- OrchestratorJS Platform from my Github repository¹.
- Android Client for OrchestratorJS, which can be downloaded from my repository².

To test the applications we would need at least two Android devices, with Android 4.0 or higher, and Bluetooth capabilities.

Note: *Due the Platform is still under development, some applications could stop working if specifications change in the Github repository. If you want to test these applications properly I encourage you use my repository instead the official.*

¹ <https://github.com/Jacb667/OrchestratorJS.git>

² <https://github.com/Jacb667/OrchestratorJSAndroid.git>

Installation

Install Node.js

Probably you can install it from the repository of your Linux Distribution, but if it's not possible here you have one repository you might use. To add the repository to Ubuntu 14.04 you can run a terminal and write these commands:

```
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
```

Once the repository is successfully added, we can install Node.js:

```
sudo apt-get install python-software-properties python g++ make nodejs
```

If you have any problem to install Node.js or you want to install it in a different platform you can download and install from the Node.js website:

<http://nodejs.org/download/>

Install MongoDB

As in Node.js, probably you can install it automatically from your distribution's repository, if not, you can add this repository to Ubuntu and install it:

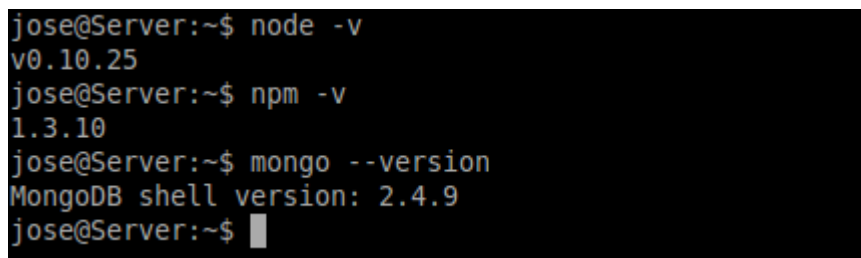
```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist
10gen' | sudo tee /etc/apt/sources.list.d/mongodb.list
sudo apt-get update
sudo apt-get install mongodb-10gen
```

If you have any problem you can download and install it from the MongoDB website:

<http://www.mongodb.org/downloads>

Check if everything is correct

We can check if all the dependencies was correctly installed executing these simple commands in the terminal:



```
jose@Server:~$ node -v
v0.10.25
jose@Server:~$ npm -v
1.3.10
jose@Server:~$ mongo --version
MongoDB shell version: 2.4.9
jose@Server:~$ █
```

Figure 41. Terminal screen in Ubuntu showing the version of all the installed dependencies.

Note: Usually `npm` will be installed with `Node.js`, if it was not installed automatically you will get an error executing the second command. In that case you can install `npm` manually in `Ubuntu` with the command: `sudo apt-get install npm`

Installing the OrchestratorJS Platform

First of all we need the sources of the Platform. We can download them from the repository or simply take the version included in the CD-ROM, which is fully functional due the repository might change in future. If you decide to take the version included in the CD-ROM you can continue to next step.

Downloading sources

If you have the Git Tool installed you can clone the repository from Github with the command:

```
git clone https://github.com/Jacb667/OrchestratorJS.git
```

Or simply you can download the ZIP file prepared by Github which contains the last versión in the repository:

```
https://github.com/Jacb667/OrchestratorJS/archive/master.zip
```

Extracting and installing

Once you have the sources you can extract them in any directory. Then open a terminal in your Operating System and navigate to the directory where you copied the sources. To install the application you simply execute this command:

```
npm install
```

This command will automatically download and install all the required dependencies.



```
npm http 200 https://registry.npmjs.org/winston/-/winston-0.7.3.tgz
npm http GET https://registry.npmjs.org/mongoose/3.6.18
npm http 200 https://registry.npmjs.org/socket.io-client/-/socket.io-client-1.0.2.tgz
npm http 200 https://registry.npmjs.org/node-fs/-/node-fs-0.1.7.tgz
npm http GET https://registry.npmjs.org/socket.io/-/socket.io-0.9.16.tgz
npm http 200 https://registry.npmjs.org/forever
npm http GET https://registry.npmjs.org/mongoose/-/mongoose-3.6.18.tgz
npm http 200 https://registry.npmjs.org/crypto/-/crypto-0.0.3.tgz
npm http GET https://registry.npmjs.org/forever/-/forever-0.11.1.tgz
npm http 200 https://registry.npmjs.org/fibers/1.0.1
npm http 200 https://registry.npmjs.org/passport/-/passport-0.2.0.tgz
npm http 200 https://registry.npmjs.org/passport-local/-/passport-local-1.0.0.tgz
Z
npm http GET https://registry.npmjs.org/fibers/-/fibers-1.0.1.tgz
npm http 200 https://registry.npmjs.org/node-uuid/-/node-uuid-1.4.1.tgz
npm http 200 https://registry.npmjs.org/request/-/request-2.36.0.tgz
npm http 200 https://registry.npmjs.org/socket.io/-/socket.io-0.9.16.tgz
```

Figure 42. Console output after executing the `npm install` command.

Then we should install the forever library manually, because sometimes it is not installed. To install it manually simply execute this command:

```
sudo npm -g install forever
```

After executing this command I recommend to make sure that we have the required version of all the libraries installed, because we could get a newer library and sometimes they don't offer backward compatibility. We can check all the recommended versions in the Annex 1.

Configuration file

There is a config file in the *orchestratorjs* folder, called *config.json*. Most of the configuration parameters won't be changed, but we here we can enable or disable a debug mode, the database name in MongoDB (if the database doesn't exist in our MongoDB server it will be automatically created), the host and port to be used. Also we can add new device types.

Running the server

Now everything is installed and configured, the last step is running the server and hopefully everything will work well. To execute the server we need to open a Terminal and go to the directory *orchestratorjs*. In the directory we can execute the following command:

```
forever orchestrator.js
```

The server will take some seconds to start, now we can access to it from a browser to check it is running. Remember to put the configured port (9000 by default):

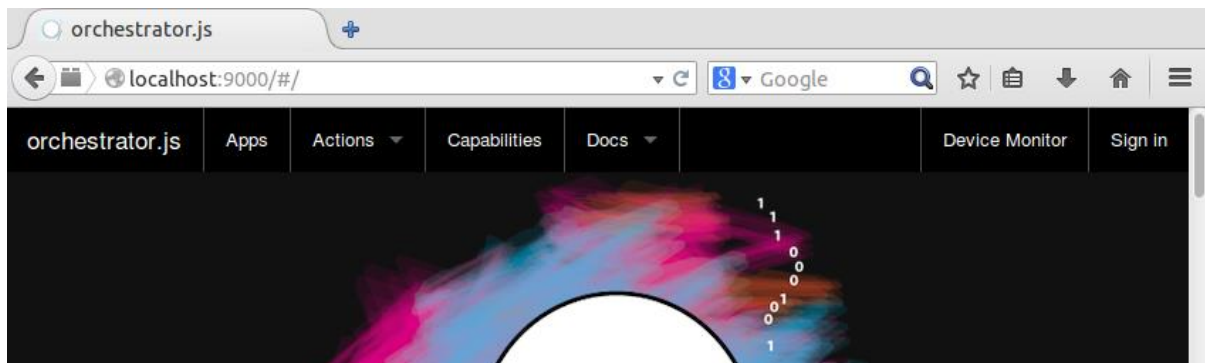


Figure 43. Welcome screen of OrchestratorJS.

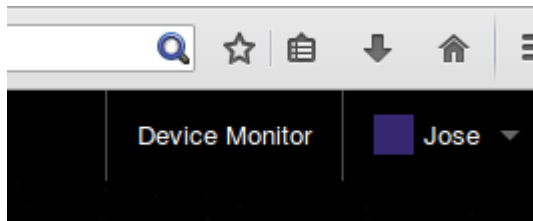
We should access the welcome screen of OrchestratorJS.

Getting started

Creating a new user

Now we have the OrchestratorJS server running and configured, let's start using it. The first step is register a new user in the platform. To do that we can go to the *Sign in* option in the main menu. A new screen will appear asking details to log in the application, to create a new user we click in the button at bottom.

Not yet registered? [Sign up](#)



The platform will ask for a username and password, after introducing them click the *Register* button. Your user is created successfully and now you can log in to the platform reintroducing the username and password. If everything went correct you should

see now your name in the right corner instead of the *Sign in* button.

Registering a device

Now we should register a device to be able to use it from the platform. To do this we can go to our user → *Add Device*. A new view like in [Figure 44] will appear. In this view we have to specify a Device name, the type and the Bluetooth MAC Address.

A screenshot of a form titled 'Add Device'. The form is divided into two main sections: 'Required Settings' and 'Optional Settings'.
Required Settings:

- Device Identity:** Jose@
- Username:** Jose (text input field)
- Device name:** nickname that is used within actions (text input field)
- Device type:** Radio buttons for android, gadgeteer, arduino, and ios.

Optional Settings:

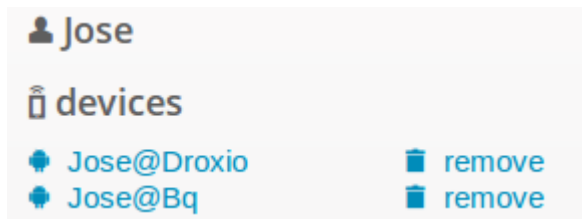
- Bluetooth MAC:** for detecting proximity (text input field)

Device Capabilities: A list of checkboxes for various capabilities:

- Apocalymbics
- Apocalymbics2
- BluetoothCapability
- CoffeeCapability
- ContactCapability
- DialogCapability
- GpsCapability
- HashTagCapability
- LocateCapability
- NotificationCapability
- PhoneNumberCapability
- PlayerCapability
- PlayerDevice
- SocialNetworkCapability
- TalkingCapability
- TestCapability
- TreasureCapability

Figure 44. View to add a new device.

The *Device Identity* will be automatically generated from our username and the device name. This identity will be asked later when we install the Android client in our device.

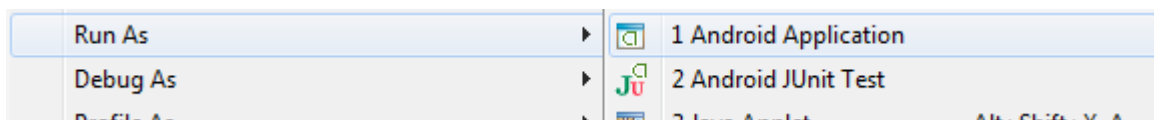


From this view we can enable or disable the desired capabilities directly from the server. We can do this now or we can return to this view in any moment from our user main screen.

The user screen can be accessed pressing in our name in the main menu (where was the *Sign in* button). In this view we will see all our registered devices, allowing us to modify their configuration or remove them.

Installing the Android client

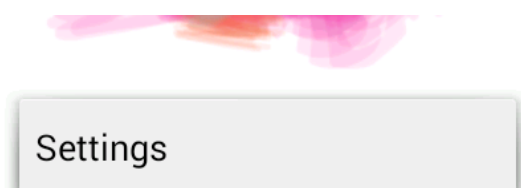
Everything is ready to install the android client. We can install the .APK file directly from the CD-ROM or import the project into the Eclipse AVD. Importing the project into eclipse will be necessary if you want to add your own changes to the client, like new capabilities.



To execute the project from Eclipse AVD it is only necessary to select the *Run As* → *Android Application*, like in any other android application. The application will be installed and executed in your device.

Note: *The Android Virtual Device cannot emulate Bluetooth capabilities, I recommend to test the platform on a real Android device.*

Once the application is running in our device, we won't be able to connect because we need to configure some parameters before. To open the application's settings view you have to press the button in your device to see the context menu, and then select the *Settings* option.



The first we should configure is the *Device Identity*, we should write here our device identity given by the website when we added the device. Usually this identity is formed in the way "*username@devicename*".

The device name must be written in the second field. In [Figure 45] the device name is set as "Emulator", and the device identity is "Jose@Emulator".

The second step, and probably the most important, is to configure the *Server Settings*. These advanced settings are the hostname and port of the OrchestratorJS server. If this is not well configured we won't be able to connect to our server neither see the available capabilities.

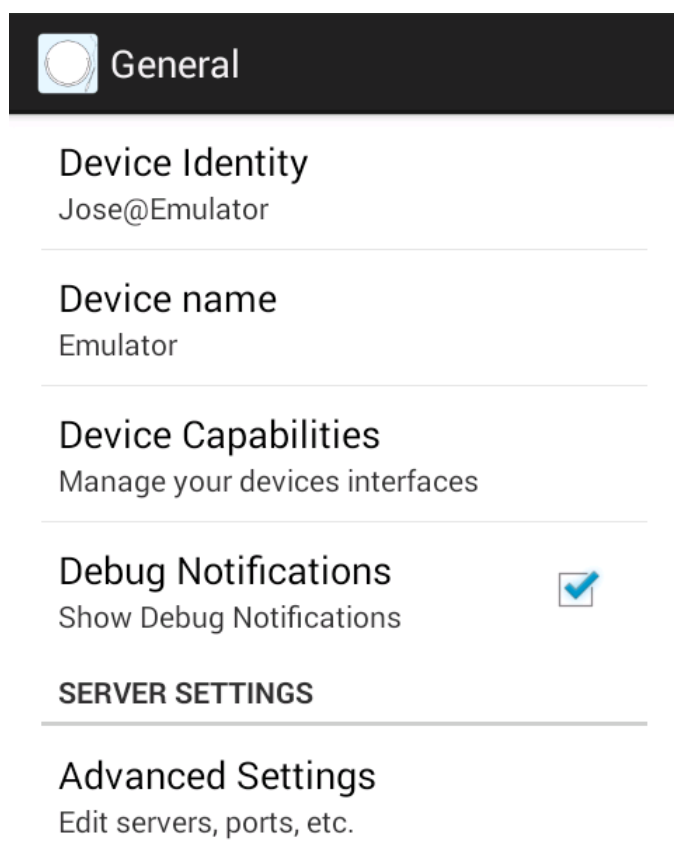
Below the *Orchestrator Settings* there are two input fields called *Proximity Settings*. These fields are not used in this moment and they can be ignored.

ORCHESTRATOR SETTINGS

hostname
192.168.1.180

port
9000

Note: If *Device Capabilities* option shows an empty screen the problem is the client is not able to connect to the server and download the list of capabilities. If the server is correctly set up the main reason could be that the hostname and port are not correct.



The *Device Capabilities* option will bring us to the view where we can enable or disable the desired capabilities. Take note that in this option we are disabling the capability on the device, not in the server.

It is possible to disable in the server-side and in the client-side. Only capabilities enabled in both sides will be executed. If the capability is disabled in server-side, the server will block immediately the actions using the capability, so the device won't receive any message.

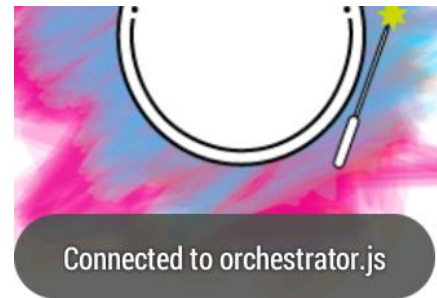
On the other hand, if we disable the capability on the device, but not on server-side, the server will send the actions, but the device won't execute them.

Figure 45. Settings screen of OrchestratorJS.

Debug Notifications is a useful option for developers, we can see small notifications showing any error occurred in the client. For example if the connection is lost or the delivery of context data fails for any reason.

Executing a first action

Everything is now configured and ready to user. In the Android client now we can close the settings menu and press the button *Connect* in the main screen. If everything is running and well configured we should see a message saying *Connected to Orchestrator.js*.



In the browser in the *Device Monitor* now the device has the background in a blue colour, which means the device is connected to the platform.

Identity	MAC	BLE UUID	Owner	Type	Capabilities
Jose@Emulator	AA:BB:CC:DD:EE:FF	88fa4cb1-2df3-4d5f-88e4-5e25bda35f43	Jose	android	

Figure 46. Device Monitor showing a connected device. The background is in blue.

Let's execute an action to see how it works. If they are not enabled, you should enable the capabilities *TalkingCapability* and *DialogCapability*, which are the capabilities used in this example.

push
remove

DialogTest

```

1 // the body
2 this.body = function (dl) {
3
4
5     var misc = require('./misc.js');
6
7     var s = 'say yes or no';
8     dl.talkingCapability.say(s);
9     dl.dialogCapability.showDialog(s, ['YES', 'NO'], 60);
10    while( !dl.dialogCapability.getDialogChoice() ) {
11        misc.sleep(1);
12    }
13
14    var choice = dl.dialogCapability.getDialogChoice();
15    console.log('CHOICE: '+choice);
16
17    dl.talkingCapability.say('the choice is '+choice,'david','0.8');
18
19
20    return choice;
21 };

```

Use previous arguments

BLE coordinator device identity

x "device:Jose@Emulator"

trigger

Figure 47. DialogTest action implementation, being executed in the device *Jose@Emulator*.

When these capabilities are enabled, we can go to the menu *Actions* → *Definitions* and select the action called *DialogTest*. This action [Figure 47] will send a basic dialog to the device, asking to answer *YES* or *NO*. The question will be also said loudly by the device, using the Text-To-Speech Capability called *TalkingCapability*.

The action will then sleep for one second continuously while the user answers the question, once the user has answered, the action leaves the loop and reads the response, which will be output to the server console. Then, it calls the *TalkingCapability* again, which will say the chosen option to the user.

To execute the action we can go down to the arguments field and select our device in the list. This parameter will be sent to the action as the *d1* parameter of the function. Then, pressing the button *trigger* will send the action to the device.



The device will say the question using the *TalkingCapability* and show the question and the options to choose.

To understand how capabilities work, you can go now to the settings view in your device and disable the *TalkingCapability*. If you execute the same action now, it will show the question and ask the user for an option, but the device won't use the Text-To-Speech feature.

Starting the development

In this section I will explain the most important technical details that every developer should know about the platform. I think this information can be a great starting point for any developer interested in create new applications for the OrchestratorJS Platform.

Note: *At the moment of this work there is no further information about how to develop applications for the platform, all of this information is the result of my investigation, and could vary in future versions.*

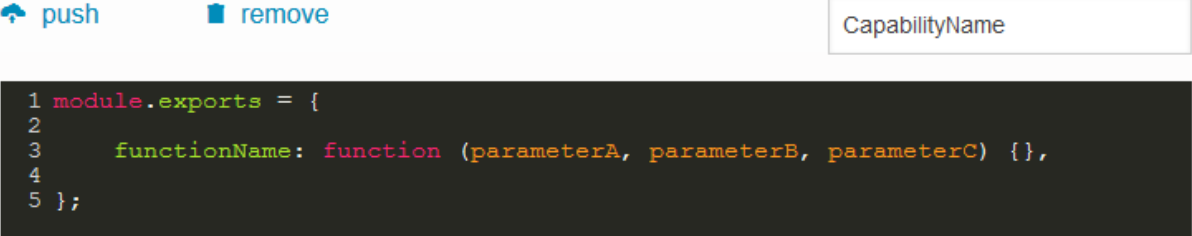
Capabilities

Capabilities are the main piece to develop when starting to implement a new functionality, as they are the only piece that is implemented in the client device, and thus, the only way to create new features.

We will use the Web IDE to create a new capability. To open the IDE, select the menu *Capabilities* in the main menu of the OrchestratorJS main page.

[Figure 48] shows the basic interface of a new capability called *CapabilityName* with a function *functionName*.

All the code inside the interface should be inside the JSON object *module.exports*. Inside the object we can add functions in the same way as we create JSON Objects, in the format *functionName : function (parameters...)*.





The screenshot shows a code editor with a dark background. At the top, there are two buttons: 'push' with a blue arrow icon and 'remove' with a blue trash icon. To the right, there is a text input field containing 'CapabilityName'. Below this, a code block contains the following JavaScript code:

```
1 module.exports = {
2
3   functionName: function (parameterA, parameterB, parameterC) {},
4
5 };
```

Figure 48. Basic code of a new functionality.

We can add as much functions as we want, all of these functions must be implemented in the client to avoid problems and runtime errors.

In this example I have created a function called *functionName* with 3 parameters: *parameterA*, *parameterB*, and *parameterC*. There is no need to specify the type of these parameters, the client will map them automatically through the name.

Let's see how this capability would be implemented in an Android client. First step will be create the right package to store the new  `com.ojs.capabilities.capabilityName` capability. This package must be inside the  `CapabilityName.java` package `com.ojs.capabilities` and the name of the package must be *capabilityName*, with the first letter in lowercase.

Inside the package we have to create the main class of the capability. The class has to be named exactly the same as the capability: *CapabilityName*.

```
public class CapabilityName {

    private static Context applicationContext_;

    public void initCapability( Context applicationContext )
    {
        CapabilityName.applicationContext_ = applicationContext;
    }

    public String functionName(String parameterA, Integer parameterB, JSONObject parameterC)
    {
        return "result";
    }
}
```

Figure 49. Implementation of the method *functionName* in *CapabilityName*.

Parameters of the method must be named as we defined in the interface, but they can be whatever type, including a `JSONObject`. Also we can specify a return type. The returned value will be taken by the server when executing the action, but the server won't wait for it.

We can specify anything inside the body of the function. For example creating a new Android Activity, calling a web service, asking an input from the user, etc.

The method *initCapability* is mandatory, we have to specify it in the main class of all the capabilities. This method will be executed when the client connects to the server and it allows to adjust any initialization settings the capability could need.

[Figure 50] shows how to send context data to the server. The context data must be a *JSONObject*, which can include any kind of data (*JSONArrays*, *Strings*, *Integers*, *Booleans*, *Doubles*, *Floats*, etc.). This data needs a name to differentiate it in the server-side.

```
try
{
    // Create a JSON Object to send the data
    JSONObject sendData = new JSONObject();

    JSONArray exampleData = new JSONArray();
    exampleData.put(10);
    exampleData.put("message");

    sendData.put("exampleData", exampleData);
    OrchestratorJsActivity.ojsContextData(sendData);
}
catch(JSONException ex)
{
    ex.printStackTrace();
}
```

Figure 50. Sending context data to the platform.

Actions

Let's create an action which uses the new capability. To create an action from the Web IDE we can go to the menu *Actions* → *Definitions*.

Actions have to be declared also inside the *module.exports* object, and they are really a new *JSON Object*. The body of the action must be always a function declared with the name *body*.

Additional functions can be declared, for example, an exception handler, which will be executed when the device throws an exception.

[Figure 51] shows a small example action which calls the method *functionName* from our capability. I included the library *misc.js*, which has a *sleep* method that will be useful as the return value from our capability method won't block the action waiting for the result. We need to poll the method until the value arrives.

I have implemented an *exceptionHandler* in this example, this method is optional and by default it will print the exception message in console and stop the action. We can specify here any behaviour to treat the exception.

```
1 module.exports = {
2
3   // exception handler
4   exceptionHandler: function(action, device, exception_value) {
5     console.log('error on client-side: ' + device.identity +
6               ', ' + exception_value);
7     action.finishAction();
8   },
9
10  // the body
11  body: function (device) {
12
13    var misc = require('./misc.js');
14
15    while( !device.capabilityName.functionName("parameterA",
16                                              10, { nombre : "Jose" }) )
17      {
18        misc.sleep(1);
19      }
20
21    var result = device.capabilityName.functionName("parameterA",
22                                                  10, { nombre : "Jose" });
23    console.log('RESULT: ' + result);
24
25  }
26 };
27
```

Figure 51. Action example using our new capability *CapabilityName*.

The *body* function has a parameter called *device*, this will be the device to execute the action. This parameter can be set in the arguments list when triggering the action.

Applications

Finally, we will implement a simple application which will call our new Action. Simply we have to go to the *Apps* menu and select the button *NEW APP* in the grid. This will open us the Web IDE with an example code of how to implement basic things.

An application will basically monitor context data received from the devices and trigger actions based on that data. We can also create threads (*Fibers*) to execute periodic tasks, like poll a website or service.

[Figure 53] shows the first part of a basic application. The first we should do is to include all the required modules from Node.js:

```
var httprequest = require( '../tools.js' ).httprequest;
var pubsub      = require( '../tools.js' ).pubsub();
var tools      = require( '../tools.js' );
var Fiber      = require( 'fibers' );
```

Then, the JSON Object *modules.exports* is declared, in the same way of actions and capabilities, we have to define the name of the object and the value. An application will have basically two objects:

- **Settings:** this object will include a list of parameters required for the application. This parameters will be requested to the user when running the application [Figure 52].
- **Logic:** this object contains a function with the body of the application.

The settings defined can have a default value specified.

Figure 52. When the user runs the application, it will be asked to introduce all the parameters defined in the *settings* object.

In the example in [Figure 53] I have defined an observer [Line 16] which is reading the *online* context data. This data is automatically sent to true when a new device is connected to the platform.

If the device is connected (if the *contextValue* is set to *true*), and the identity of the connected device is the identity the user has specified in the parameter *deviceId*, the program will continue, otherwise it will return without executing any instruction.

Then, the application will call the REST API using the *httprequest* module, executing the action *ActionName* on the device contained in the variable *deviceIdentity*.

🔼 push
🗑️ remove

NewApplication

```

6 module.exports = {
7
8 // Define here settings that the app needs from user
9 // ( asked from the user s/he starts the app )
10 settings: { deviceId: null, value: null },
11
12 // Define your action triggering logic here
13 logic: function() {
14
15 // Example: pubsub observer for monitoring device online state
16 pubsub.on( 'online', function( contextValue, deviceIdentity ) {
17
18     if ( contextValue != true && settings.deviceId != deviceIdentity)
19         return;
20
21 // Example how an action can be triggered
22 var params = {
23     actionName: 'ActionName',
24     parameters: [ 'device:' + deviceIdentity ]
25 };
26
27 httprequest( {
28     uri: '/api/1/actioninstance',
29     method: "POST",
30     form: params
31 }, function( error, response, body ) {} );
32
33 } );
34
35

```

Figure 53. Example application (part 1).

[Figure 54] shows the second part of the code. This part only creates a thread (*Fiber*) that simply prints the word *tick* in console every 10 seconds.

```
36 // Example: periodic loop e.g. for polling a website
37 Fiber( function() {
38
39     // runs every ten seconds until the app gets stopped
40     while ( true ) {
41
42         console.log( 'tick' );
43
44         tools.sleep( 10 );
45
46     }
47
48     } ).run();
49 }
50 };
```

Figure 54. Simple example of a Fiber of the application (part 2).

Note: Always remember to save your code before executing it. To save your code simply press the button 'Push'. If you change something in your code and you press the button 'Start' or 'Trigger' (for actions), it will execute the code saved in the server, not the code you are editing in your browser.

Chapter 5

Application hiThere!

In this chapter I will introduce the first study application. The application *hiThere!* is the first of five applications to prove the potential of the platform. Also I will explain how the entire application, its actions, and its capabilities are developed, to help to understand the implementation process.

The goal of this application is to read the list of contacts from the device, and automatically send a notification to the user if one of his contacts is in the surroundings. This is achieved using a Bluetooth detection of nearby devices.

Each user will be asked to introduce his mobile phone number, this phone number will be mapped to the user's identity in the platform and, consequently, to his Bluetooth MAC address, which will be used to detect the proximity.

[Figure 55] shows how the application at high level. Now I will explain its details in deep, as actions, capabilities and context data.

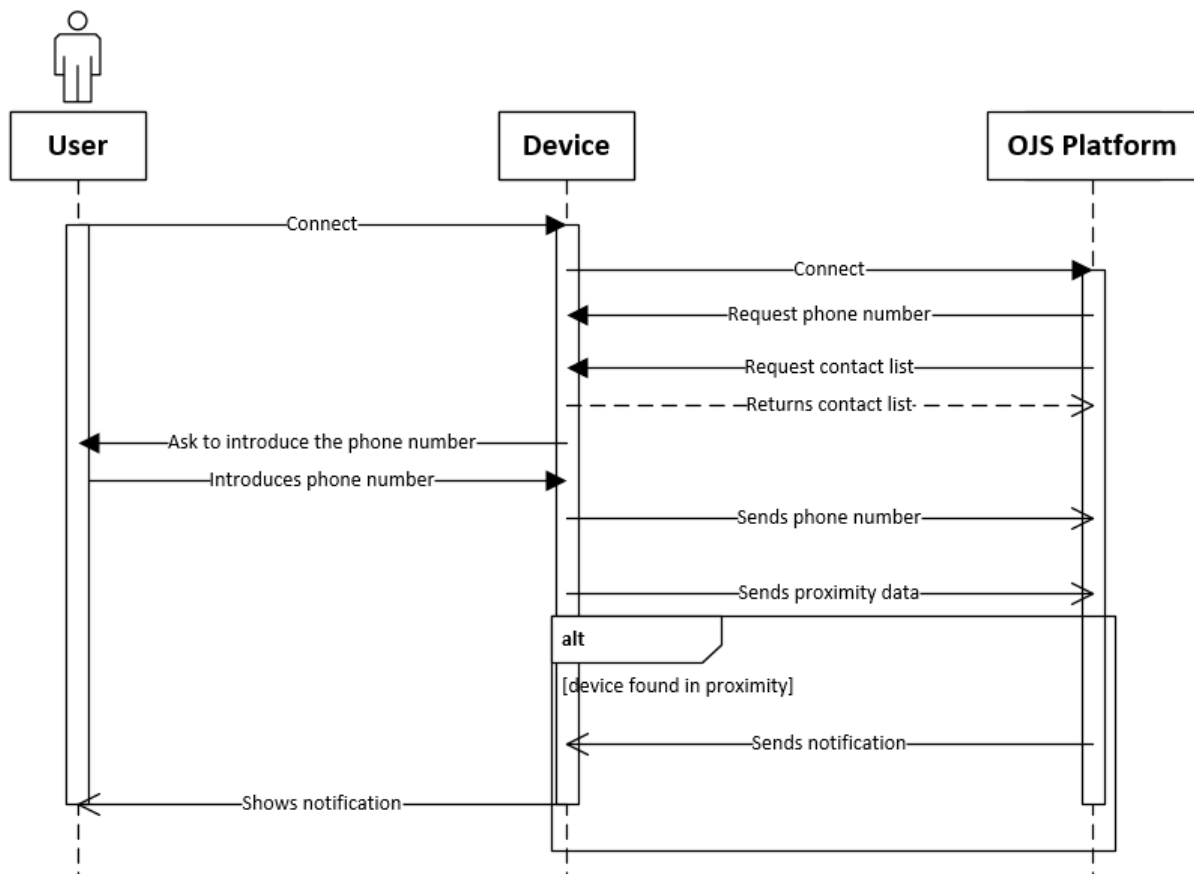


Figure 55. Activity diagram with the behaviour of the application *hiThere!*

Implementation

I will explain now the most important things about the implementation. This application will use the actions: *PhoneDialog*, *GetContactList*, *SendNotification* and *UpdateProximity*.

The *PhoneDialog* action will call the method *askPhoneNumber* from the *phoneNumberCapability*.

```
public void askPhoneNumber( String message, Integer timeout ) throws Exception
{
    Log.d(OrchestratorJsActivity.TAG, "asking Phone Number!");

    // Create a new Intent to start the activity
    Intent i = new Intent(PhoneNumberCapability.applicationContext_, PhoneNumberCapabilityActivity.class);
    i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);

    // Send the arguments from the platform
    i.putExtra("phoneMessage", message);
    i.putExtra("timeout", timeout*1000);

    // Start the capability
    PhoneNumberCapability.applicationContext_.startActivity(i);
}
```

In the client this method simply starts a new Android's Activity, which will show a dialog to the user to input the mobile phone number. The platform sends two parameters: a *message* to show to the user, and a timeout, which indicates the number of seconds to automatically close the Activity if the user doesn't introduce the phone number.

The Activity will send the phone number introduced by the user in an asynchronous message, as a context data.

The code in [Figure 56] shows the method to send the context data to the platform.

```
try
{
    JSONObject sendPhone = new JSONObject();
    sendPhone.put("phoneNumber", phoneEdit.getText().toString());
    OrchestratorJsActivity.ojsContextData(sendPhone);

    SharedPreferences settings = getPreferences(0);
    SharedPreferences.Editor editor = settings.edit();
    editor.putString("phoneNumber", phoneEdit.getText().toString());

    editor.commit();
}
catch (JSONException e)
{
    e.printStackTrace();
}
finish();
```

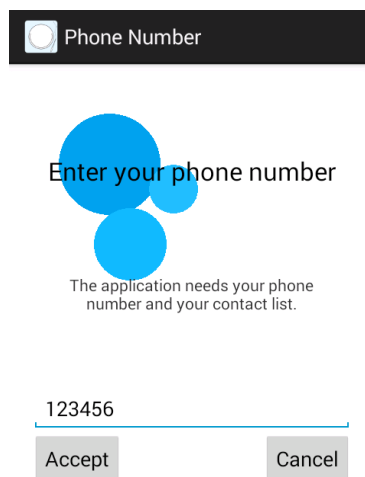


Figure 56. Fragment of code that sends the *phoneNumber* introduced by the user as context data to the OrchestratorJS Platform.

```
// Devices without a valid phone number cannot participate in this app
pubsub.on( 'phoneNumber', function( contextValue, deviceIdentity )
{
    console.log(deviceIdentity + ": " + contextValue);
    phoneMap[deviceIdentity] = contextValue;
} );
```

This is the code of the observer listening for the context data called *phoneNumber*, the application will receive the phone number introduced by the user and will store it in an object indexed with the device identity.

At the end of the application there is a Fiber that constantly executes updating the proximity of all connected devices. This will iterate through all the connected devices (devices which we have the phone number in the map) and request their Bluetooth proximity using the *UpdateProximity* action.

```
// Update Bluetooth visibility of all connected devices
Fiber( function()
{
    while ( true )
    {
        console.log( 'bluetooth tick' );

        // Iterate through all the devices and update the proximity.
        Object.keys(phoneMap).forEach(function(key)
        {
            console.log( "request proximity on " + key );

            var params = {
                actionName: 'UpdateProximity',
                parameters: [ 'device:' + key ]
            };

            httpRequest( {
                uri: '/api/1/actioninstance',
                method: "POST",
                form: params
            }, function( error, response, body ) {} );

        });

        tools.sleep( 30 );
    }
} ).run();
```


BluetoothCapability is as I added Bluetooth support to the OrchestratorJS Platform. One of the last versions of the platform has added Bluetooth support too, but it is implemented as a service that runs constantly on the device. My capability is compatible with this method, but my capability allows the developer to ask for a Bluetooth update only when it is needed.

This capability has some initialization code in the Android device, that code is executed when the user connects to the platform. The most important piece of code in this capability is the Bluetooth discovery:

```
// The BroadcastReceiver that listens for discovered devices and
// changes the title when discovery is finished
private final BroadcastReceiver mReceiver = new BroadcastReceiver()
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        String action = intent.getAction();

        // When discovery finds a device
        if (BluetoothDevice.ACTION_FOUND.equals(action))
        {
            // Get the BluetoothDevice object from the Intent
            BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            int RSSI = intent.getShortExtra(BluetoothDevice.EXTRA_RSSI, Short.MIN_VALUE);

            mDevicesMap.put(device.getAddress().toLowerCase(), RSSI);
        }
        // When discovery is finished, change the Activity title
        else if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED.equals(action))
        {
            if (connected && OrchestratorJsActivity.isConnected())
                sendDevices.run();
        }
    }
};
```

Figure 57. Piece of code that shows how the Bluetooth discovery is implemented.

This Receiver is executed for each discovered device in Android. What I do is storing all the discovered devices in the *mDevicesMap*, which stores the Bluetooth MAC address of the device and the RSSI, to be used as an approximation of the distance between both devices.

The capability has a timeout configured by the developer, after that timeout the client will send all the discovered devices. In the action *UpdateProximity* I configured this value as 10 seconds. That means that the device will send all the discovered devices after 10 seconds, but it will continue looking for devices and when it finishes will send all of them again.

The delivery is done in a similar way as the phone number, each device is a JSON Array with the Bluetooth MAC address and the RSSI. All the devices are stored in another JSON Array, and finally everything is wrapped into a JSON Object [Figure 58].

```

try
{
    // Create a JSON Object to send the data
    JSONObject sendData = new JSONObject();

    // Create a JSON Array to store all the devices
    JSONArray devices = new JSONArray();

    for (Entry<String, Integer> entry : mDevicesMap.entrySet())
    {
        // Create a JSON Array to add the device
        JSONArray device = new JSONArray();
        device.put(entry.getKey());
        device.put(entry.getValue());

        // Add the device to devices array
        devices.put(device);
    }
    System.out.println("SENDING BLUETOOTH DATA!");
    sendData.put("bt_devices", devices);
    System.out.println(sendData);
    OrchestratorJsActivity.ojsContextData(sendData);
}

```

Figure 58. Structure of the Bluetooth proximity message.

The OrchestratorJS Platform will now receive the object *bt_devices* internally (it is not parsed in my application). It will automatically translate the Bluetooth MAC addresses into Device Identities of the registered devices in the platform.

```

// Request data from proximity
pubsub.on( 'proximityDevices', function ( contextValue, deviceIdentity )
{
    for (var i = 0; i < contextValue.length;i++)
    {
        var device = contextValue[i][0];
        var phoneNum = phoneMap[deviceIdentity];

        // Check if the user has me in his contact list.
        if (contactList[device] != null)
        {
            for (var p = 0; p < contactList[device].length; p++)
            {
                // I have already sent a notification to this device.
                if (sendedMap[device] == deviceIdentity)
                    continue;

                if (phoneNum == contactList[device][p])
                {
                    sendedMap[device] = deviceIdentity;
                    // Send notification to the device!
                    // -- Rest of the code omitted for simplicity.
                }
            }
        }
    }
}

```

After parsing and translating the Bluetooth MAC addresses, it will broadcast the result in the context data *proximityDevices*. Now we can create an observer for this message in our application, which will read Device Identities instead of Bluetooth MAC addresses, what simplifies the process of searching nearby devices.

Now (see the simplified code in the previous page) I will check what of the proximity devices are in the contact list, the application will iterate through all the discovered devices checking if they are in the contact list and if I have already sent a notification. Please note that if the device is not connected to the application, its phone number will be null, so the notification will be never sent.

```
var request = require("request")
var url = "http://192.168.1.180:9000/api/1/devices"

request({
  url: url,
  json: true
}, function (error, response, body)
{
  if (!error && response.statusCode === 200)
  {
    for (var i = 0; i < body.devices.length; i++)
    {
      // This device
      if (body.devices[i]["identity"] == deviceIdentity)
      {
        phoneMap[deviceIdentity] =
          body.devices[i]["metadata"]["phoneNumber"];
      }
    }

    // Request the phone number to the device if we don't have it
    if (phoneMap[deviceIdentity] == undefined ||
        phoneMap[deviceIdentity] == null ||
        phoneMap[deviceIdentity] == "")
    {
      // Call the PhoneDialog Action (omitted for simplicity)
    }
  }
});

// Request the contact list to the device (GetContactList action)
```

The first part of this code will request the API with all the RAW JSON Data from the Platform (see [Figure 22] in [Page 48]). Then, all that data is parsed to check if we have already stored the user's phone number in the platform. If we have the phone number I

will read it instead of asking to the user to enter its number again. If the user changes his phone number he can delete it from the *Device Settings* in the Web Console, and the application will ask for the number again when he connects the device.

If the phone number was not found, the application will call the action *PhoneDialog* and the user will be able to enter it.

Finally, at the end of the code it requests the contact list to the device. This action will call the capability *ContactListCapability*, which will read the list of contacts from the device.

```
public void getContactList()
{
    ArrayList<String> phoneNumbers = new ArrayList<String>();
    ContentResolver cr = applicationContext_.getContentResolver();
    Cursor phones = cr.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
        null,null,null, null);
    while (phones.moveToNext())
    {
        /*String name = phones.getString(phones.getColumnIndex(
            ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));*/
        String phoneNumber = phones.getString(phones.getColumnIndex(
            ContactsContract.CommonDataKinds.Phone.NUMBER));
        phoneNumbers.add(phoneNumber);
    }

    phones.close(); // close cursor
}
```

This is how the list of contacts is read. They are stored in the *phoneNumbers* ArrayList, and then it is sent to the platform as a context data in a JSON Object.

Note: *In this application we use sensitive data from the user, as his phone number and contact list. Take note that hi can disable these capabilities in any moment, so the data won't be requested neither sent to the server.*

Conclusions

This application makes use of several capabilities to provide a new functionality to the user: the possibility to receive an alert when one of his contacts is in his surroundings.

Chapter 6

Application eGreetings

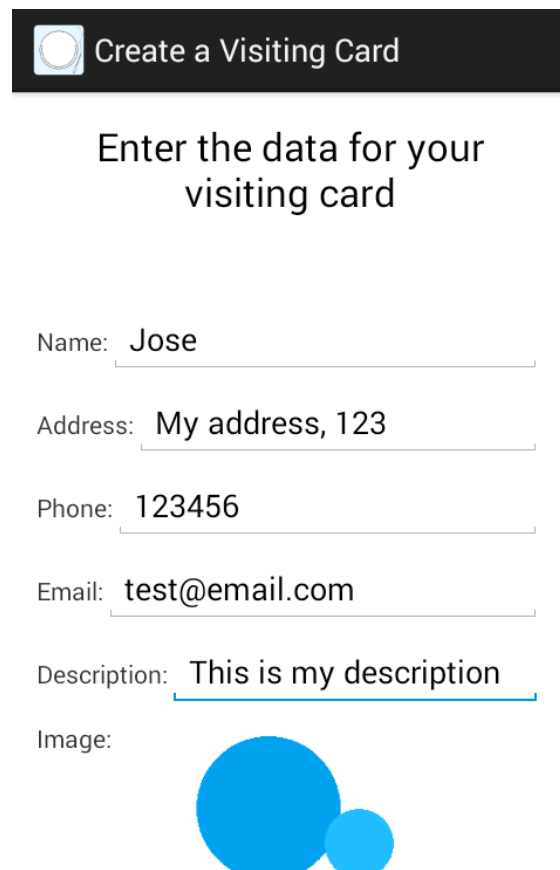
This is the second study application. The focus of this application is that when two people meet, their mobile device will automatically share their visiting cards each other. This interaction will be automatic and the user won't be required to do the interaction, he only needs to connect the device to the platform and set up the visiting card.

The behaviour of this application is simpler than the last one, however the difficult here is the devices have to share a bigger amount of data, which can include images.

Implementation

This application makes use of the actions: *RequestVisitingCard*, *ShowVisitingCard* and *UpdateProximity*.

- *RequestVisitingCard*: This is an action similar to *PhoneDialog* in the last one application. This action will call the method *requestVisitingCard* from the *visitingCardCapability*. The method simply creates a new activity where the user can introduce the data to create the visiting card.
- *ShowVisitingCard*: This action is which will send the visiting card of a user to another users in the proximity. This action has seven arguments, the first of them is the device identity to send the visiting card, and the rest of arguments are all the data of the visiting card. The last of these arguments corresponds to the image, which will be sent as a String in Base64 format [Figure 59].
- *UpdateProximity*: This is the same method as the first application and its behaviour is exactly the same.



Create a Visiting Card

Enter the data for your visiting card


Name:

Address:

Phone:

Email:

Description:

Image: 

ShowVisitingCard

```
1 module.exports = {
2
3   body: function ( dev, name, address, phone, email, description, image )
4   {
5
6     dev.visitingCardCapability.showVisitingCard(name,
7                                               address,
8                                               phone,
9                                               email,
10                                              description,
11                                              image);
12   }
13
14 };
```

Figure 59. ShowVisitingCard action implementation.

To simplify the code, I have created a new function called *executeActionDevice*, which will be present in all the applications. This method encapsulates the code to call the REST API to execute an action:

```
function executeActionDevice( actionName, parameters )
{
  var params = {
    actionName: actionName,
    parameters: parameters
  };

  httprequest( {
    uri: '/api/1/actioninstance',
    method: "POST",
    form: params
  }, function( error, response, body ) {} );
}
```

The first part of the application implements an observer as in the *hiThere!* application. This observer is executed each time a device is connected or disconnected from the platform. If the device is connection (the *contextValue* is set to *true*), the application will call the action *RequestVisitingCard* to ask the user create a new card.

```
// This will be executed each time a device is connected.
pubsub.on( 'online', function( contextValue, deviceId ) {

  if ( contextValue == true )
  {
    console.log( 'Connected device: ' + deviceId );

    // Request the visiting card to the user
    executeActionDevice( 'RequestVisitingCard', [ 'device:' + deviceId ] );
  }
  else
  {
    // Delete the phone from the map
    delete connectedDevices[deviceId];
    delete sendedMap[deviceId];
  }
} );
```

Figure 60. Implementation of the first observer.

If the device is disconnecting from the platform, the application will delete all the stored data (the objects *sendedMap* and *connectedDevices*).

When the user creates a new visiting card, it is sent to the platform as context data called *visitingCard*. This data contains all the fields and the image in Base64 format:

```
private void sendVisitingCard()
{
    String name = tname.getText().toString();
    String address = taddress.getText().toString();
    String phone = tphone.getText().toString();
    String description = tdescription.getText().toString();
    String email = temail.getText().toString();

    byte[] b = {};
    Bitmap bm = BitmapFactory.decodeFile(selectedImagePath);
    if (bm == null)
        bm = ((BitmapDrawable)timage.getDrawable()).getBitmap();

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    bm.compress(Bitmap.CompressFormat.JPEG, 50, baos);
    b = baos.toByteArray();

    String image = Base64.encodeToString(b, Base64.DEFAULT);

    try
    {
        // Object to be sent
        JSONObject sendData = new JSONObject();

        // Visiting card
        JSONObject visitingCard = new JSONObject();

        visitingCard.put("name", name);
        visitingCard.put("address", address);
        visitingCard.put("phone", phone);
        visitingCard.put("email", email);
        visitingCard.put("description", description);
        visitingCard.put("image", image);

        sendData.put("visitingCard", visitingCard);
        OrchestratorJsActivity.ojsContextData(sendData);
    }
    catch (JSONException ex)
    {
        ex.printStackTrace();
        Toast.makeText(this, "Error!", Toast.LENGTH_SHORT).show();
    }

    Toast.makeText(this, "Visiting Card created successfully!",
        Toast.LENGTH_SHORT).show();
}
```

Then, the server will receive this JSON Object and store the visiting card in the *connectedDevices* map, with the device identity as key. This data is stored in the server to facilitate the share of the information between devices without having to request the

information from the user each time. However the data is removed from the server when the user disconnects.

Finally, the last new part in this code is the *proximityDevices*. This code works in a similar way to the code in the last application, however in this case the visiting card will be sent to every user (connected to the platform) in proximity. Also I have added a configurable timeout in settings, set by default to 10 minutes. This means that the visiting card will be sent maximum once each 10 minutes.

```
// Request data from proximity
pubsub.on( 'proximityDevices', function ( contextValue, deviceIdentity )
{
  if (deviceIdentity in connectedDevices)
  {
    // My visitingCard to send
    var visitingCard = connectedDevices[deviceIdentity];

    for (var i = 0; i < contextValue.length;i++)
    {
      var device = contextValue[i][0];

      if (device in connectedDevices)
      {
        var now = new Date().getTime();
        if (sendedMap[deviceIdentity] != null &&
            sendedMap[deviceIdentity][device] != null &&
            sendedMap[deviceIdentity][device] > now)
        {
          console.log("I have already sent to this device!!");
        }
        else
        {
          // Don't send to this device in at least 10 minutes
          sendedMap[deviceIdentity][device] = new Date().getTime() +
            settings.timeout;

          var parameters = [ 'device:' + device, visitingCard.name,
                              visitingCard.phone, visitingCard.address,
                              visitingCard.email, visitingCard.description,
                              visitingCard.image ]

          executeActionDevice( 'ShowVisitingCard', parameters );
        }
      }
    }
  }
}
```

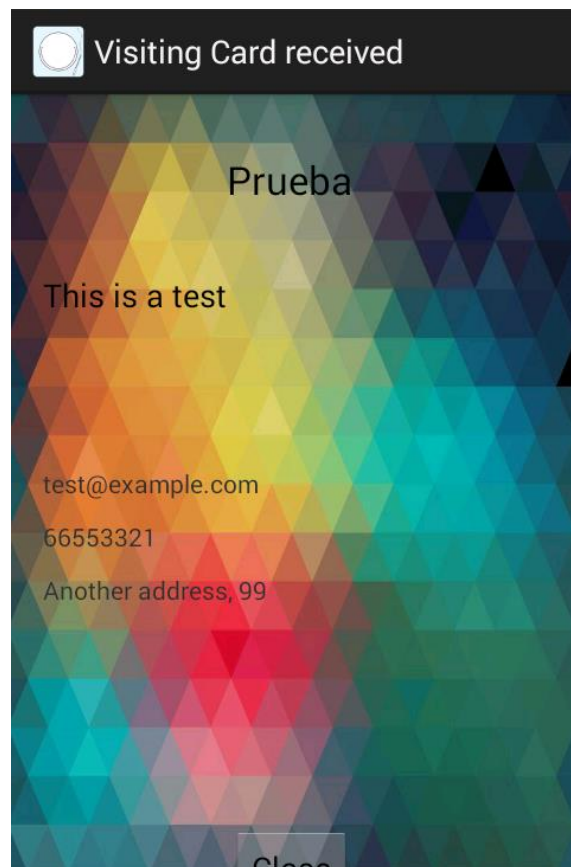
Note that the code has been simplified, I have removed logs and debug information not needed in this example.

Actions can receive only primitive data (*Strings, Integers, Doubles, Floats, and Booleans*) as parameters, they don't allow JSON Objects as parameters, which is the reason I have to send all the fields in the visiting card one by one instead of sending the entire object.

When all the requirements are met the action *ShowVisitingCard* is executed in the target device, showing my visiting card to the other device.

Each device will send the visiting card based on his own proximity. This means that when two devices meet, maybe they don't interchange their visiting cards in the same moment. Some devices can take more time to discover nearby devices.

[Figure 61] shows the code from *ShowVisitingCardActivity* which takes all the arguments, including the image, and shows the visiting card to the user.



```
String name = args.getStringExtra("name");
String phone = args.getStringExtra("phone");
String address = args.getStringExtra("address");
String description = args.getStringExtra("description");
String email = args.getStringExtra("email");
String image = args.getStringExtra("image");

tname.setText(name != null ? name : "");
tphone.setText(phone != null ? phone : "");
taddress.setText(address != null ? address : "");
tdescription.setText(description != null ? description : "");
temail.setText(email != null ? email : "");

if (image != null)
{
    try
    {
        byte[] decodedString = Base64.decode(image, Base64.DEFAULT);
        Bitmap decodedByte = BitmapFactory.decodeByteArray(decodedString, 0, decodedString.length);
        timage.setImageBitmap(decodedByte);
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}
```

Figure 61. Example of code to read all the arguments and show the visiting card to the user. The image is sent as a String codified in Base64. This string is decoded and showed in an ImageView.

Conclusions

This application can be very useful to businessmen who have a lot of meetings in a day. Their mobile devices can send the visiting card automatically to everybody in their surroundings. This can be a cheap way to do publicity.

For example, the application could run in a small device at the entrance of a shop, and it will send the visiting card to the people who pass in front of the shop. This system could be used to offer discount coupons to clients or to make small adverts.

Of course, some people can think this may be disturbing, however they can disable the capability in any moment and they will stop receiving visiting cards.

Chapter 7

Application followMe

FollowMe is the third application. This basic application asks the user to enter their preferences in a hashtag way. Then, the application will get all the user's hashtags and match them based on the devices in your proximity.

With this application you can easily find people with your interests in your surroundings. When the devices find one or more common hashtags, they ask the users if they want to share their data with the other user. If both users answer "yes", their social network data will be shared each other.

Implementation

When a new device connects to the application, it automatically sends two actions. The first, *RequesthashTags*, is a view to request the user to introduce his interests. The second action, *RequestSocialData*, will request the social network data of the user.

```
30 // Get the "online" status. This will be executed each time a device is connected.
31 pubsub.on( 'online', function( contextValue, deviceId ) {
32
33     console.log("online " + deviceId);
34
35     if ( contextValue == true )
36     {
37         console.log( 'Connected device: ' + deviceId );
38
39         // Ask the hashtags to the user
40         executeActionDevice( 'RequestHashTags' , ['device:' + deviceId] );
41
42         // Ask the social data to the device
43         executeActionDevice( 'RequestSocialData' , ['device:' + deviceId] );
44
45         sendMap[deviceId] = { };
46
47     }
48     else
49     {
50         // Delete all the data from the map
51         delete deviceHashTags[deviceId];
52         delete sendMap[deviceId];
53         delete contactDetails[deviceId];
54     }
55 } );
56
```

Figure 62. First part of the *followMe* code application with the login observer.

This data will be sent by the Android client, and receive by two different observers:

```
// Observer for hashTags
pubsub.on( 'hashTags', function( contextValue, deviceId )
{
    deviceHashTags[deviceId] = contextValue;
    console.log("hashTags " + deviceHashTags[deviceId]);
} );
```

```
// Observer for socialData
pubsub.on( 'socialData', function( contextValue, deviceIdentity )
{
    contactDetails[deviceIdentity] = contextValue;
    console.log("socialData " + contactDetails[deviceIdentity]);
} );
```

The user has no limit in the number of hashtags he can use. To add a new hashtag he has to simply press the button *New tag*. To remove a hashtag he can swipe it from the list.

Used hashtags are also saved in the Android client, so the user doesn't have to specify them when he disconnects and connects again. They will be there and the user can send them back to the server or edit them before delivering.

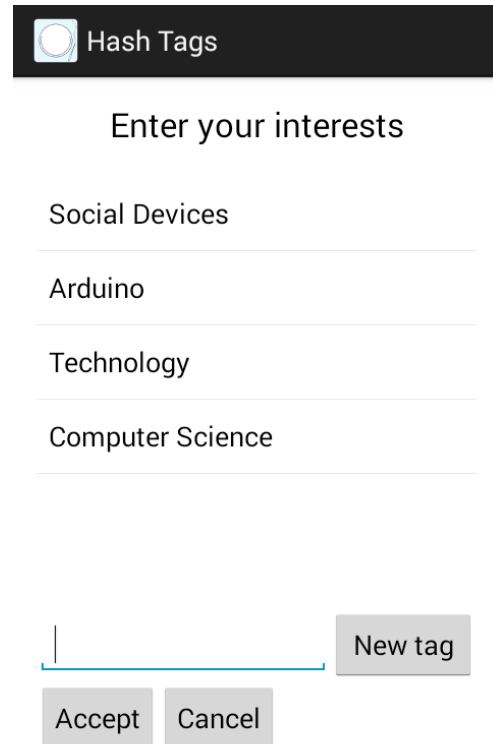
Finally, this application also uses the *UpdateProximity* action, which updates the proximity of all connected devices each 20 seconds.

This is the code to retrieve the saved social networks from the Android device:

```
public void requestDetails() throws Exception
{
    Log.d(OrchestratorJsActivity.TAG, "requesting Social Network data!");

    try
    {
        JSONObject sendData = new JSONObject();
        JSONArray socialData = new JSONArray();

        Account[] accounts = AccountManager.get(applicationContext).getAccounts();
        for (Account account : accounts)
        {
            if (account.type.equalsIgnoreCase("com.twitter.android.auth.login"))
            {
                JSONObject cuenta = new JSONObject();
                cuenta.put("twitter", account.name);
                socialData.put(cuenta);
            }
            else if (account.type.equalsIgnoreCase("com.facebook.auth.login"))
            {
                JSONObject cuenta = new JSONObject();
                cuenta.put("facebook", account.name);
                socialData.put(cuenta);
            }
        }
        sendData.put("socialData", socialData);
        OrchestratorJsActivity.ojsContextData(sendData);
    }
}
```



This code will read your social network data from *Twitter* and *Facebook*. Support for more accounts can be provided, and an option to let the user which of them he prefers to share or not.

[Figure 63] shows the main implementation of the *proximityDevices* observer. This is the main code of the application. This observer will iterate all the nearby devices reported by your device, comparing their hashtags with yours to find potential matches.

It has a cool down timer like the last application to avoid sending the notification several times to the same user.

```
75 // Request data from proximity
76 pubsub.on( 'proximityDevices', function ( contextValue, deviceIdentity ) {
77
78     console.log("Proximity: " + deviceIdentity + " - " + contextValue);
79     console.log("JSON: " + JSON.stringify(contextValue));
80
81     for(var i = 0; i < contextValue.length;i++)
82     {
83         var device = contextValue[i][0];
84
85         if(deviceHashTags[device] == null)
86         {
87             console.log( "Device " + device + " is not connected.");
88             continue;
89         }
90
91         var now = new Date().getTime();
92         if (sendedMap[deviceIdentity] != null &&
93             sendedMap[deviceIdentity][device] != null &&
94             sendedMap[deviceIdentity][device] > now)
95         {
96             console.log("I have already sent to this device in the last 10 minutes");
97         }
98         else
99         {
100            console.log("Device " + device + " is connected!!");
101            sendedMap[deviceIdentity][device] = new Date().getTime()+10*60*1000;
102            sendedMap[device][deviceIdentity] = new Date().getTime()+10*60*1000;
103
104            var commonHashTags = [ ];
105            for(var h = 0; h < deviceHashTags[deviceIdentity].length; h++)
106            {
107                var hashTag = deviceHashTags[deviceIdentity][h];
108                console.log("Checking if " + device + " has " + hashTag);
109                // Match hashtags with the other device
110                if (deviceHashTags[device].indexOf(hashTag) != -1)
111                {
112                    commonHashTags.push(hashTag);
113                }
114            }
115
116            if (commonHashTags.length > 0)
117            {
118                console.log("Found common hashtags: " + commonHashTags);
119
120                executeActionDevice( 'SendRequestInformationDialog',
121                    ['device:' + device,
122                    'device:' + deviceIdentity,
123                    commonHashTags,
124                    JSON.stringify(contactDetails[device]),
125                    JSON.stringify(contactDetails[deviceIdentity])
126                    ] );
127            }
128        }
129    }
130 }
131 );
```

Figure 63. Main implementation of the *proximityDevices* observer.

However, this application doesn't use your proximity data to send your information (like the eGreetings app). In this case, the first device who finds the potential hashtag match will start the action. The timer will be set in both devices, to avoid to call the action again when the second device's proximity sensor finds the current device.

Once the application has found all the common hashtags, it will call the action *SendRequestInformationDialog* with 5 parameters: the device identity I have found, my own device identity, the list of common hashtags, the JSON Object converted to String (Stringified) with the contact details of the first device, and the contact details of myself.

[Figure 64] shows the implementation code of the action. This action will send a dialog to both devices asking if they want to share their contact details with each other. This is done using the capability *DialogCapability*, one of the original capabilities from the OrchestratorJS Platform. This capability will create a small dialog with the options YES and NO, with a timeout of 60 seconds.

If both users answer YES to the sharing of contact details, both devices will share the information each other and the action will finish.

```
1 // the body
2 this.body = function (dev1, dev2, hashTags, contact1, contact2) {
3
4     var misc = require('./misc.js');
5
6     console.log(contact1);
7     console.log(contact2);
8
9     var message = 'A user was found in your surroundings with common Hash
10     hashTags + '. Do you want to contact him?';
11
12     dev1.dialogCapability.showDialog(message, ['YES', 'NO'], 60);
13     dev2.dialogCapability.showDialog(message, ['YES', 'NO'], 60);
14     while( !dev1.dialogCapability.getDialogChoice() ) {
15         misc.sleep(1);
16     }
17     while( !dev2.dialogCapability.getDialogChoice() ) {
18         misc.sleep(1);
19     }
20
21     var choice1 = dev1.dialogCapability.getDialogChoice();
22     var choice2 = dev2.dialogCapability.getDialogChoice();
23
24     console.log( dev1 + " CHOICE " + choice1);
25     console.log( dev2 + " CHOICE " + choice2);
26
27     if (choice1 == "YES" && choice2 == "YES")
28     {
29         dev1.notificationCapability.showNotificationJSON("Social information
30         "The user has accepted your request, contact him at: ",
31         contact2);
32         dev2.notificationCapability.showNotificationJSON("Social information
33         "The user has accepted your request, contact him at: ",
34         contact1);
35     }
36
37 };
```

Figure 64. Implementation of the action *SendRequestInformationDialog*.


As we have seen in previous examples, requesting data directly from a capability is not synchronized, what means that the action won't wait for the response. That means


that the action has to poll it constantly until it receives the response. This is achieved using the *sleep* method from the *misc.js* library, which will sleep the running action for 1 second.

Conclusions

This application can be useful to meet new people with same interests. Probably every day when you are going to the work, shopping, or home, you pass near people with your same interests.

This system can be very useful to help people in the street. If someone is looking for something he can configure an alert in the device, for example, he is looking for a room to rent in the neighborhood. Probably you are renting a room, so you can configure the advert in your device and when you are nearby in the street Social Devices will do the rest to join you.

 Notification



Social information

The user has accepted your request, contact him at:

Facebook

cuenta@facebook

Twitter

cuenta@twitter

Chapter 8

Application haveYouMet

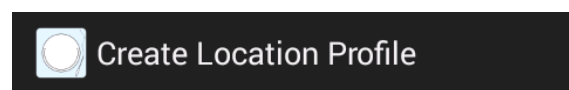
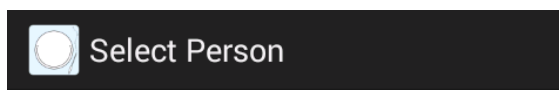
Imagine you are in a big concentration of people and you are looking for somebody, which could be a friend or a new person you don't know. Social Devices can help you to find that person using the *haveYouMet* application.

This application is focused in meet people when you are near them but you don't know where. It uses GPS technology combined with Bluetooth to provide indications of proximity about the person you are trying to locate.

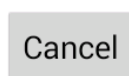
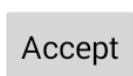
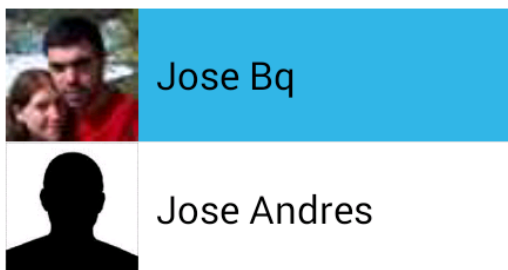
Implementation

This application requires the creation of a profile with the name and the photo of the user. This profile is requested when the user connects to the application [Figure 66].

When the user has sent the profile, he receives a list with all the users connected to the application. If he is looking for somebody he simply has to select that person in the list [**Error! Reference source not found.**], and the application will help him to locate



Enter the name of the person you are looking.



Enter your nick and avatar

Enter your profile data, so other people can locate you.

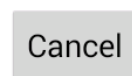
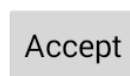
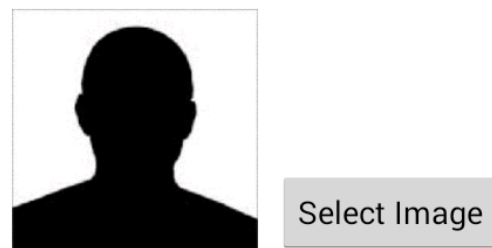


Figure 66. Creation of a new profile.

Figure 65. View to select a person to locate.

that person.

```
29 // Request the profile to the user
30 pubsub.on( 'online', function( contextValue, deviceIdentity ) {
31
32     if ( contextValue == true )
33     {
34         executeActionDevice( 'RequestLocateProfile',
35                             [ 'device:' + deviceIdentity ] );
36         locateDevice[deviceIdentity] = [ ];
37     }
38     else
39     {
40         delete connectedDevices[deviceIdentity];
41         delete locateDevice[deviceIdentity];
42     }
43
44     } );
```

Figure 67. Fragment of code that requests the profile to the connected device.

This application uses the actions:

- *RequestLocateProfile*: used to ask for the user profile to the device. It will show the Activity where the user can introduce his data [Figure 66].
- *RequestLocatePerson*: the device will show a view which contains information about all the connected people, where the user will be able to select a person from this list [**Error! Reference source not found.**].
- *RequestGpsPosition*: simply requests the GPS coordinates to the device, the response will be sent as asynchronous context data, which will be received by an observer in the application.
- *SetBluetoothTarget*: used to start locating a person. The device will receive information from the target device, as GPS coordinates and the Bluetooth MAC address of the target device.
- *SendNotification*: this is the simple notification message, it is used to notify the user if something was wrong trying to locate the device.
- *UpdateGPSLocation*: this action will send the current GPS coordinates of the device to all the devices trying to locate it.

The code executed when a new device connects to the application is shown in [Figure 67]. It simply requests the profile to the user. When the user disconnects all the information is removed from the application.

```
45 pubsub.on( 'locateProfile', function( contextValue, deviceIdentity ) {
46
47     connectedDevices[deviceIdentity] = contextValue;
48
49     executeActionDevice( 'RequestLocatePerson',
50                         [ 'device:' + deviceIdentity,
51                           JSON.stringify(connectedDevices) ] );
52
53     } );
```

Figure 68. Observer checking for the *locateProfile* context data.

The user's device will send the profile as context data called *locateProfile*. The observer in [Figure 68] will receive this data and store it. Then it will call the action *RequestLocatePerson*, sending a String with the content of the *connectedDevices* object, which contains all the information about the devices using the application.

The user can now select one of the connector users from the list, and the server will receive the response as context data. If the user doesn't want to locate a person at this moment he can simply close or cancel the Activity.

```
53     pubsub.on( 'locatePerson', function( contextValue, deviceIdentity ) {
54
55         console.log(deviceIdentity + " wants to locate " + contextValue);
56         var found = false;
57         for (var device in connectedDevices)
58         {
59             if (connectedDevices[device]["name"] == contextValue)
60             {
61                 locateDevice[device].push(deviceIdentity);
62                 found = true;
63
64                 executeActionDevice ( 'RequestGpsPosition', [ 'device:' + device ] );
65                 executeActionDevice ( 'SetBluetoothTarget', [ 'device:' + deviceIdentity,
66                                                             contextValue,
67                                                             connectedDevices[device]["bmac"],
68                                                             connectedDevices[device]["image" ] ] );
69
69                 break;
70             }
71         }
72
73         if (!found)
74         {
75             executeActionDevice( 'SendNotification', [ 'device:' + deviceIdentity,
76                                                         "Device not found",
77                                                         "There is no device with such name.",
78                                                         "30" ] );
79         }
80     }
81 }
```

Figure 69. Code of the observer that receives the person you want to locate.

This figure shows the code of the observer, it looks for the user in the connected devices, because probably the user got disconnected while we were selecting him. If the user is connected, the application requests an update of his GPS Position, and send the information to our device.

The information received includes the name and the image of the user, and the Bluetooth MAC address of his device. This MAC address will be user to locate the device in out proximity.

Our device will now open a new Activity with some information of the user and the location data. Each minute we will receive the GPS coordinates of our target, and our device will calculate the distance to our target using our GPS position. Our device will also try to locate the device by the Bluetooth MAC address. The discovering of devices will be done each minute, however when we are getting closer conform the GPS distance, the Bluetooth discovering will update faster to try to locate the device.

The *locateDevice* will store all the devices currently trying to locate me. The object is indexed by the identity device who is trying to be located, and the values are all the devices trying to locate me.

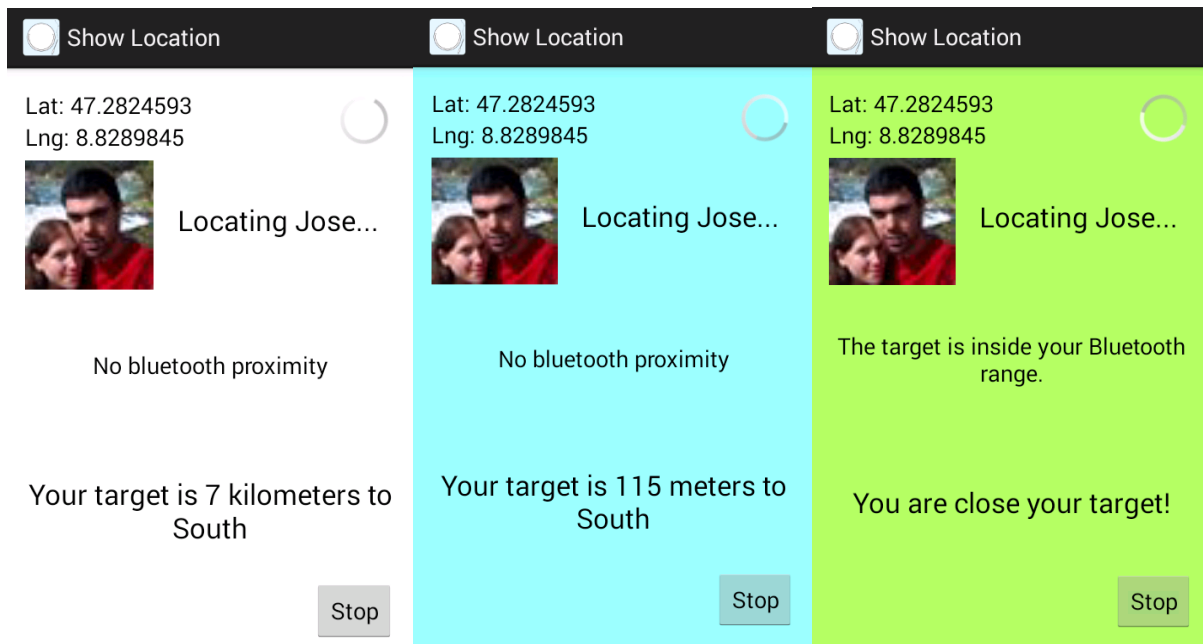


Figure 70. Show location view in three different stages.

[Figure 70] shows three different stages from the application in the client. The first shows the screen when the target is very far (more than 200 meters). The second, when the target is closer, at this moment the background will change to blue, and the Bluetooth proximity check will be faster to try to locate the target device. Finally, when the device has been found by Bluetooth the background colour changes according the Bluetooth signal strength. It has 4 different states, from orange to green.

The code of this activity is bigger, so I won't comment it here. However it is not difficult to understand. The activity used is *MapLocateCapabilityActivity.java*.

Finally, the last interesting part of this application. There is a *Fiber* at the end of the script that requests the GPS position of all the connected devices. That request will provoke the client send a context data called *gps_location*. [Figure 71] shows the code of the observer. This observer will take the *locateDevice* object and send my GPS position to all the devices trying to locate me.

```
pubsub.on( 'gps_location', function( contextValue, deviceIdentity ) {
    // Resend the GPS Position to every device trying to locate us
    for (var i = 0; i < locateDevice[deviceIdentity].length; i++)
    {
        console.log("Sending to " + locateDevice[deviceIdentity][i]);
        executeActionDevice( 'UpdateGPSLocation', [ 'device:' + locateDevice[deviceIdentity][i],
            connectedDevices[deviceIdentity]["name"],
            JSON.stringify(contextValue) ]);
    }
} );
```

Figure 71. Observer for the *gps_location* context data.

Conclusions

This application can be useful in meetings and conferences with a lot of people, also it can be used to locate a new person you have meet for Internet for example, and you don't know him.

The application can be extended with extra functionality like google maps to see the exact place of your target, however this could violate the privacy of the user.

GPS location is useless inside a building, however this application combines it with Bluetooth proximity to give a more accurate distance of your target.

Chapter 9

Application treasureHunt

The last application is a small treasure hunting game where the users can play and compete each other. This application also combines GPS location with Bluetooth proximity to know where you are and what do you have in your surroundings.

Each treasure can have some information like a name, a description or hint, the GPS location, a score, etc. There is also a scoreboard with the scores of the players.

Implementation

Treasures are static devices, they cannot move and they can be implemented in any device with Bluetooth capabilities. They won't need to execute the client neither Internet connection. For example, a treasure could be a small Arduino or similar device with Bluetooth capabilities.

They are set in the first lines of the application:

```
1 var httprequest = require( '../tools.js' ).httprequest;
2 var pubsub      = require( '../tools.js' ).pubsub();
3 var tools      = require( '../tools.js' );
4 var Fiber      = require( 'fibers' );
5
6 var treasureList = { 'Tesoro Casa':{'location':[47.2224593,8.8289845], 'bmac':'e8:2a:ea:24:71:d7'} };
7 var players = { };
8 var scores = { };
9 var foundTreasures = { };
10
```

This application makes use of the following capabilities:

- *RequestPlayerProfile*: This is very similar to the used in the last application, it requests data from the user, like the username and the avatar that will be shown to another players. This action will call the capability *locateCapability*.
- *RequestLocateTreasure*: Very similar to the *RequestLocatePerson* of the last application, but this will show the list of all nearby treasures, so we can pick one to start the hunting.
- *SendPlayerList*: as the name says, this will be the action responsible for sending the scoreboard to all the players. The scoreboard is sent when a new user connects to the application, and each time a user finds a treasure.
- *SetBluetoothTreasure*: similar to *SetBluetoothTarget*, this action will send special data like position and Bluetooth MAC address of the target treasure.
- *SendNotification*: basic notification to inform the user when something was wrong. For example, when he has already found a treasure.

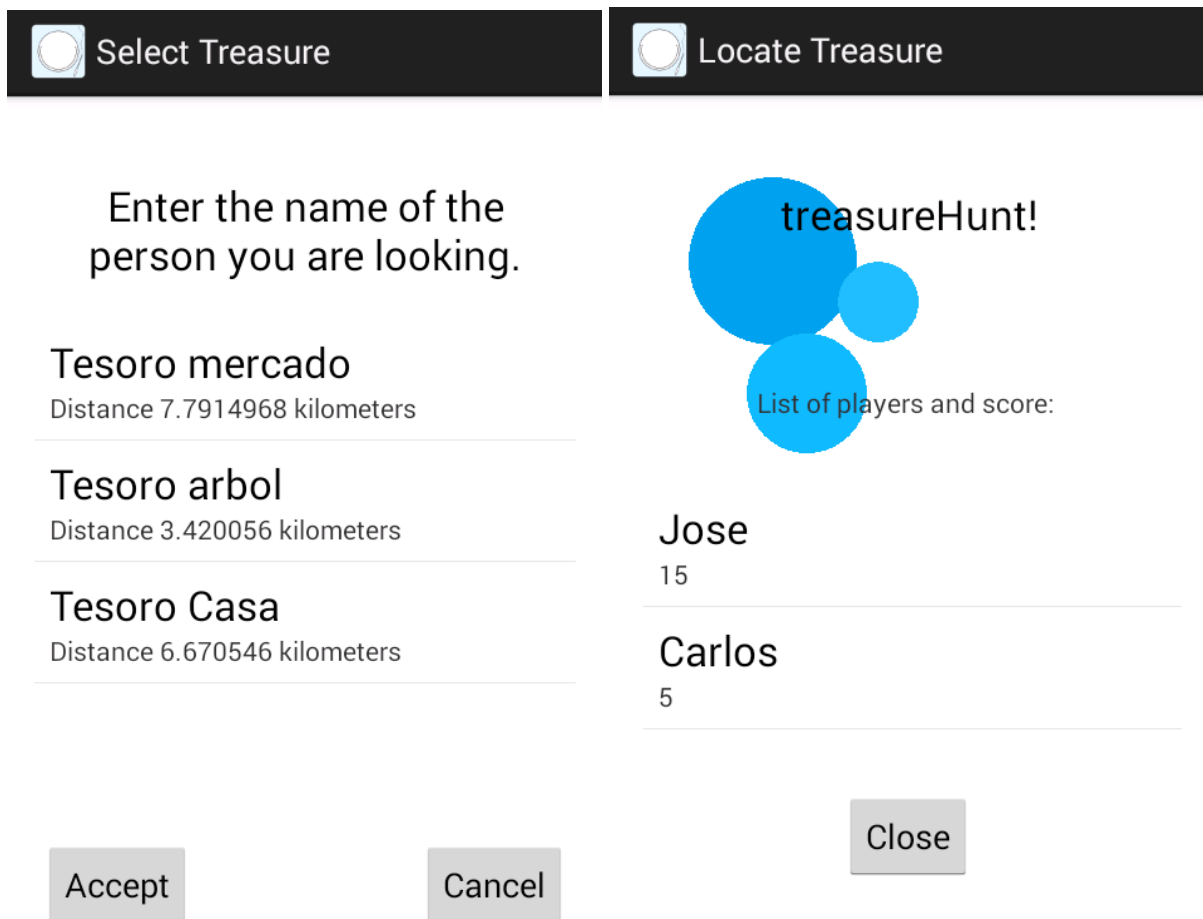


Figure 72. Example of views. The first shows all the nearby treasures, the user should select one of them to start the location. Each treasure shows the distance to the current position of the user. The second figure is an example of the scoreboard.

The code of the observer *locateTreasure* is similar to the *locatePerson* in the last application, however now we have to check if the user has already found the selected treasure:

```

87  pubsub.on( 'locateTreasure', function( contextValue, deviceIdentity ) {
88
89      console.log(deviceIdentity + " LOCATE TREASURE " + contextValue);
90
91      if (treasureList[contextValue] != null)
92      {
93          if (foundTreasures[deviceIdentity].indexOf(contextValue) == -1)
94          {
95              executeActionDevice( 'SetBluetoothTreasure', [ 'device:' + deviceIdentity,
96                                                            contextValue,
97                                                            treasureList[contextValue]["bmac"] ] );
98          }
99          else
100         {
101             executeActionDevice( 'RequestLocateTreasure', [ 'device:' + deviceIdentity, JSON.stringify(tr
102             executeActionDevice( 'SendNotification', [ 'device:' + deviceIdentity,
103                                                         "Already found",
104                                                         "You have already found that treasure, choose a ne
105                                                         "30" ] );
106         }
107     }
108     else
109     {
110         executeActionDevice( 'RequestLocateTreasure', [ 'device:' + deviceIdentity, JSON.stringify(tr
111         executeActionDevice( 'SendNotification', [ 'device:' + deviceIdentity,
112                                                         "Treasure not found",
113                                                         "The treasure cannot be found",
114                                                         "30" ] );

```

Finally, the *MapTreasureCapabilityActivity* will do the same as *MapLocateCapabilityActivity*, it will check periodically the location of the treasure, giving hints of the orientation and distance. When the treasure is inside the Bluetooth range, the screen will change its color according to the distance.

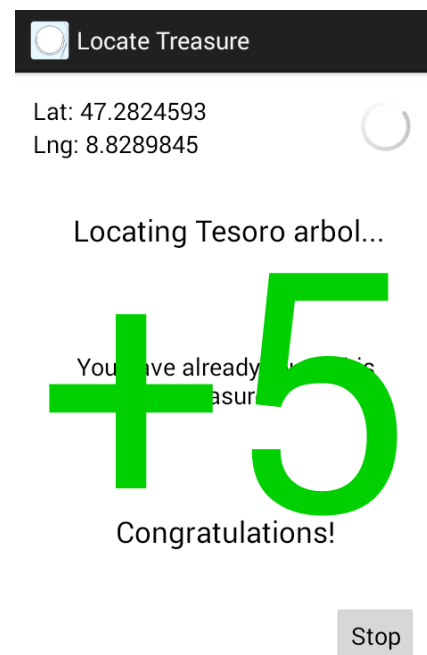
When you are closer enough the treasure (about 2-3 meters), you will get your score, and the possibility to select another treasure to hunt. The scoreboard will be sent to all the players to know you have found a new treasure.

```
pubsub.on( 'foundTreasure', function( contextValue, deviceId ) {  
  
    if (scores[deviceId] == null)  
    {  
        scores[deviceId] = 5;  
    }  
    else  
    {  
        if (foundTreasures[deviceId].indexOf(contextValue) == -1)  
        {  
            scores[deviceId] += 5;  
        }  
    }  
  
    foundTreasures[deviceId].push(contextValue);  
  
    executeActionDevice( 'RequestLocateTreasure', [ 'device:' + deviceId  
  
    for (var player in players)  
    {  
        executeActionDevice( 'SendPlayerList', [ 'device:' + deviceId,  
    }  
    } );
```

Figure 73. Code of the observer *foundTreasure*.

[Figure 73] shows the code of the *foundTreasure* observer. This observer will give you the corresponding points (5 in this case, but they can be configured). Then, it will ask you to select a new treasure to hunt, and send the scoreboard to all the connector players.

You will see a small animation [see figure at right] giving you the score. This treasure will be automatically disabled for you, so you won't get your points again.



Conclusions and future work

Social Devices is aimed to increase, facilitate and enrich social interactions between people in various kinds of co-located and face-to-face situations. In this work I have presented the latest platform implementing this concept: OrchestratorJS. This platform allows to create and manage all the devices, capabilities, actions and apps through the website without the need of external tools.

Furthermore we have seen how the concept can be integrated with such technologies like *Internet of Things* and *pervasive computing*, which are not the main target of the platform, but has a big potential with them.

To facilitate the development, the platform is based in the *Action-Oriented Programming Model*, which offers abstractions for developers that help them to structure pervasive applications. The development of applications is very flexible and modular because the main piece are *Actions*, a small piece of code which can create a small interaction between some devices. Actions are reusable and easily integrated in any application.

In this work I have presented several prototypes of thirds using the People-as-a-Service concept or related with it. To finish this work, I have developed five study cases which show the potential of the platform. These applications are only small examples, with an important didactical objective, because at the moment of this work there was no documentation about how to start the development in the platform.

In the future, the platform could have new features and capabilities, recently the Bluetooth support has been added, and probably GPS support will be a feature soon. Probably in the videogames field this kind of platforms can have an important future, there are two example videogames where users can start the game and invite people in their surroundings to start playing. With the different frameworks and the REST API, the technology can also be extended to many different types of devices.

Furthermore the new Bluetooth 4.0 Low Energy can be a perfect solution to avoid the current Bluetooth problems, which make the devices to waste huge amounts of energy when finding or communicating with nearby devices. Without Bluetooth Low Energy won't be possible the deployment of a technology like Social Devices.

Conclusiones y trabajo futuro

Social Devices está orientado a aumentar, facilitar y enriquecer las interacciones sociales entre las personas de diferentes formas cooperativas en situaciones cara a cara. En este trabajo presento la última plataforma que implementa este concepto: OrchestratorJS. Esta plataforma permite crear y gestionar todos los dispositivos, capacidades, acciones y aplicaciones desde un sitio web sin necesidad de herramientas externas.

Además hemos visto como el concepto puede ser integrado con tecnologías como *Internet de las Cosas* y *computación ubicua*, que no son el principal objetivo de la plataforma, pero tiene un gran potencial en ellas.

Para facilitar el desarrollo, la plataforma está basada en el *Modelo de Programación Orientado a Acciones*, que ofrece abstracciones que ayudan a estructurar aplicaciones ubicuas. El desarrollo de aplicaciones es muy flexible y modular ya que la pieza principal son las *Acciones*, un pequeño fragmento de código que capaz de crear una pequeña interacción entre varios dispositivos. Las acciones son reutilizables y fácilmente integrables en cualquier aplicación.

En este trabajo he presentado varios prototipos de terceros utilizando el concepto de *People-as-a-Service* o relacionados con él. Para terminar este trabajo, he desarrollado cinco casos de estudio que muestran el potencial de la plataforma. Estas aplicaciones son sólo pequeños ejemplos, con un importante objetivo didáctico, ya que en el momento de este trabajo no existe documentación sobre cómo empezar a desarrollar para esta plataforma.

En un futuro, la plataforma podría tener nuevas características y capacidades, recientemente se ha añadido soporte Bluetooth, y probablemente el soporte GPS será pronto una nueva funcionalidad. Probablemente en el campo de los videojuegos este tipo de plataformas puedan tener un importante futuro, existen dos videojuegos de ejemplo donde los usuarios pueden comenzar a jugar e invitar a personas a su alrededor para jugar juntos. Con los diferentes marcos de trabajo y la API REST, la tecnología puede también extenderse a muchos tipos diferentes de dispositivos.

Además la nueva versión de Bluetooth 4.0 de baja energía puede ser la solución perfecta para evitar los problemas del Bluetooth actual, que hace que los dispositivos gasten grandes cantidades de energía cuando buscan o se comunican con los dispositivos cercanos. Sin Bluetooth de baja energía no será posible el despliegue de una tecnología como Social Devices.

Annex 1: npm modules

Due to the amount of problems can appear because there are always new updates of the NodeJS libraries, I have done a dump of all my installed modules in npm. The idea is to keep exactly the same version of all the modules here appear, to avoid any problem.

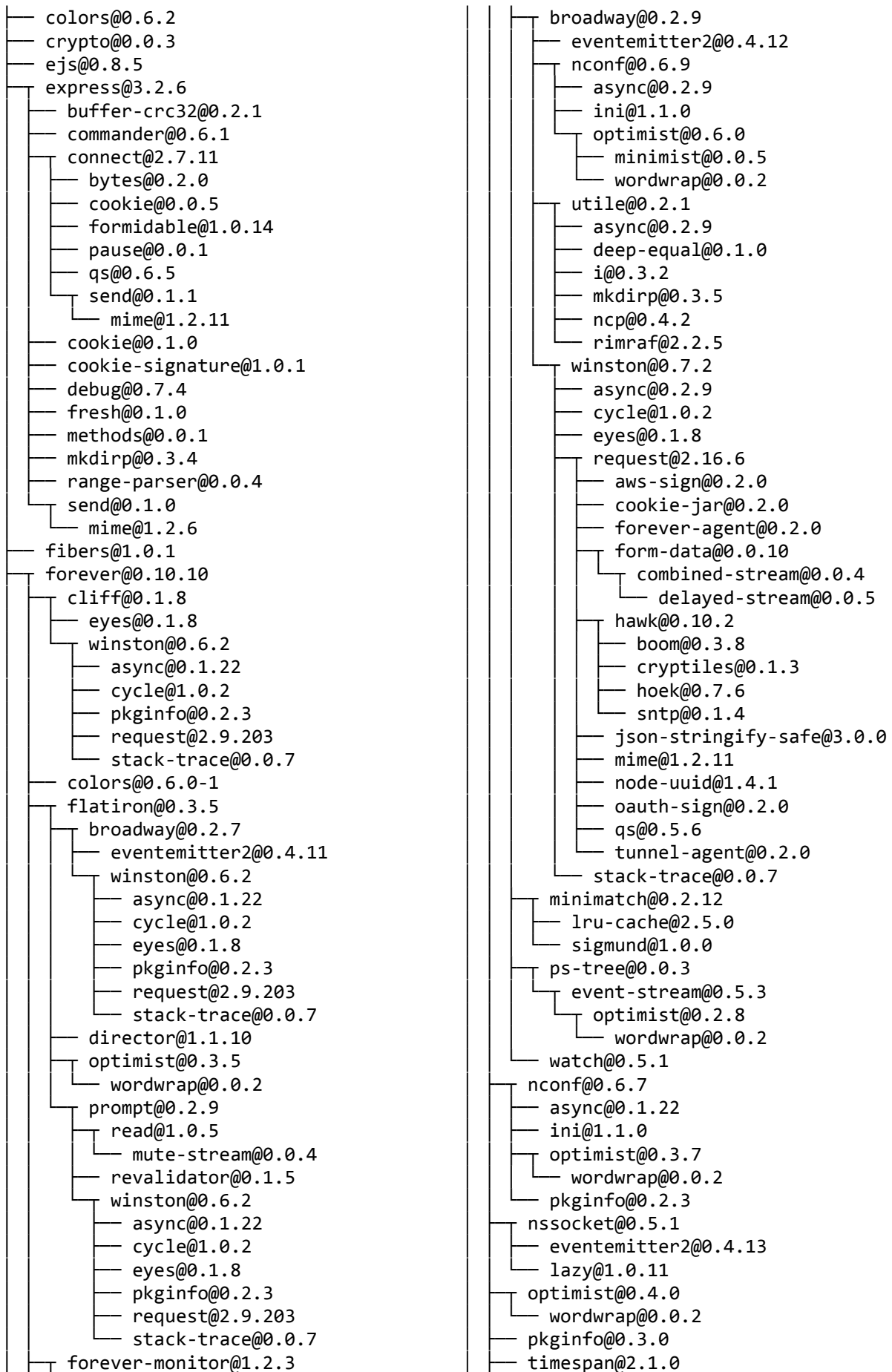
For example, the Sockets.IO library is one of the most problematic libraries, because from version 1.0 it doesn't offer backward compatibility and the OrchestratorJS Platform won't work as expected.

Here is the set of commands to install npm modules to the right version. At the moment these commands should be enough, but in the future more things could vary. However the way to install old versions would be the same:

```
npm install colors@0.6.2
npm install crypto@0.0.3
npm install ejs@0.8.5
npm install express@3.2.6
npm install fibers@1.0.1
npm install forever@0.10.10
npm install forever-monitor@1.2.3
npm install mongojs@0.7.17
npm install mongoose@3.6.18
npm install mu2express@0.0.1
npm install node-fs@0.1.7
npm install node-uuid@1.4.1
npm install passport@0.2.0
npm install passport-local@1.0.0
npm install request@2.34.0
npm install socket.io@0.9.16
npm install socket.io-client@0.9.16
npm install winston@0.7.2
```

To install the required version we only have to specify the name of the module and the desired version, separated with '@'.

In the next page is the entire dump of npm installed modules.



- utile@0.1.7
 - async@0.1.22
 - deep-equal@0.1.0
 - i@0.3.2
 - mkdirp@0.3.5
 - ncp@0.2.7
 - rimraf@1.0.9
- watch@0.7.0
- winston@0.7.1
 - async@0.2.9
 - cycle@1.0.2
 - eyes@0.1.8
- request@2.16.6
 - aws-sign@0.2.0
 - cookie-jar@0.2.0
 - forever-agent@0.2.0
 - form-data@0.0.10
 - combined-stream@0.0.4
 - delayed-stream@0.0.5
 - hawk@0.10.2
 - boom@0.3.8
 - cryptiles@0.1.3
 - hoek@0.7.6
 - sntp@0.1.4
 - json-stringify-safe@3.0.0
 - mime@1.2.11
 - node-uuid@1.4.1
 - oauth-sign@0.2.0
 - qs@0.5.6
 - tunnel-agent@0.2.0
- stack-trace@0.0.7
- forever-monitor@1.2.3
- broadway@0.2.9
 - cliff@0.1.8
 - eyes@0.1.8
 - winston@0.6.2
 - async@0.1.22
 - cycle@1.0.3
 - pkginfo@0.2.3
 - request@2.9.203
 - stack-trace@0.0.9
- eventemitter2@0.4.12
- nconf@0.6.9
 - async@0.2.9
 - ini@1.1.0
 - optimist@0.6.0
 - minimist@0.0.8
 - wordwrap@0.0.2
- utile@0.2.1
 - async@0.2.10
 - deep-equal@0.2.1
 - i@0.3.2
 - mkdirp@0.3.5
 - ncp@0.4.2
 - rimraf@2.2.6
- minimatch@0.2.14
 - lru-cache@2.5.0
 - sigmund@1.0.0
- pkginfo@0.3.0

- ps-tree@0.0.3
 - event-stream@0.5.3
 - optimist@0.2.8
 - wordwrap@0.0.2
- utile@0.1.7
 - async@0.1.22
 - deep-equal@0.2.1
 - i@0.3.2
 - mkdirp@0.3.5
 - ncp@0.2.7
 - rimraf@1.0.9
- watch@0.5.1
- mongojs@0.7.17
 - mongodb@1.3.19
 - bson@0.2.2
 - kerberos@0.0.3
 - readable-stream@1.0.17
 - thunky@0.1.0
- mongoose@3.6.18
 - hooks@0.2.1
 - mongodb@1.3.19
 - bson@0.2.2
 - kerberos@0.0.3
 - mpath@0.1.1
 - mpromise@0.2.1
 - sliced@0.0.4
 - ms@0.1.0
 - muri@0.3.1
 - regexp-clone@0.0.1
 - sliced@0.0.5
- mu2express@0.0.1
 - mu2@0.5.20
- node-fs@0.1.7
- node-uuid@1.4.1
- passport@0.2.0
 - passport-strategy@1.0.0
 - pause@0.0.1
- passport-local@1.0.0
 - passport-strategy@1.0.0
- request@2.34.0
 - aws-sign2@0.5.0
 - forever-agent@0.5.2
 - form-data@0.1.2
 - async@0.2.10
 - combined-stream@0.0.4
 - delayed-stream@0.0.5
 - hawk@1.0.0
 - boom@0.4.2
 - cryptiles@0.2.2
 - hoek@0.9.1
 - sntp@0.2.4
- http-signature@0.10.0
 - asn1@0.1.11
 - assert-plus@0.1.2
 - ctype@0.5.2
- json-stringify-safe@5.0.0
- mime@1.2.11
- node-uuid@1.4.1
- oauth-sign@0.3.0

```

|
|--- qs@0.6.6
|--- tough-cookie@0.12.1
|   |--- punycode@1.2.4
|   |--- tunnel-agent@0.3.0
|--- socket.io@0.9.16
|--- base64id@0.1.0
|--- policyfile@0.0.4
|--- redis@0.7.3
|--- socket.io-client@0.9.16
|   |--- active-x-obfuscator@0.0.1
|   |   |--- zeparser@0.0.5
|   |--- uglify-js@1.2.5
|   |--- ws@0.4.31
|   |   |--- commander@0.6.1
|   |   |--- nan@0.3.2
|   |   |--- options@0.0.5
|   |   |--- tinycolor@0.0.1
|   |--- xmlhttprequest@1.4.2
|--- socket.io-client@0.9.16
|   |--- active-x-obfuscator@0.0.1
|   |   |--- zeparser@0.0.5
|   |--- uglify-js@1.2.5
|   |--- ws@0.4.31
|   |   |--- commander@0.6.1
|   |   |--- nan@0.3.2
|   |   |--- options@0.0.5
|   |   |--- tinycolor@0.0.1
|   |--- xmlhttprequest@1.4.2
|--- winston@0.7.2
|--- async@0.2.9
|--- colors@0.6.2
|--- cycle@1.0.2
|--- eyes@0.1.8
|--- pkginfo@0.3.0
|--- request@2.16.6
|   |--- aws-sign@0.2.0
|   |--- cookie-jar@0.2.0
|   |--- forever-agent@0.2.0
|   |--- form-data@0.0.10
|   |   |--- combined-stream@0.0.4
|   |   |--- delayed-stream@0.0.5
|   |--- hawk@0.10.2
|   |   |--- boom@0.3.8
|   |   |--- cryptiles@0.1.3
|   |   |--- hoek@0.7.6
|   |   |--- sntp@0.1.4
|--- json-stringify-safe@3.0.0
|--- mime@1.2.11
|--- node-uuid@1.4.1
|--- oauth-sign@0.2.0
|--- qs@0.5.6
|--- tunnel-agent@0.2.0
|--- stack-trace@0.0.7

```

References

- [1] Aaltonen, T., Myllärniemi, V., Raatikainen, M., Mäkitalo, N., & Pääkko, J. (2013). An Action-Oriented Programming Model for Pervasive Computing in a Device Cloud. *20th Asia-Pacific Software Engineering Conference*, 467-475.
- [2] Abowd, G., Brumitt, B., & Shafer, S. (2001). Ubicomp 2001: Ubiquitous Computing. *International Conference Atlanta*. Georgia: Springer.
- [3] *An Overview of the Android Architecture*. (3 de July de 2014). Retrieved from Techotopia:
http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture
- [4] *Android (operating system)*. (21 de October de 2014). Retrieved from Wikipedia:
[http://en.wikipedia.org/wiki/Android_\(operating_system\)](http://en.wikipedia.org/wiki/Android_(operating_system))
- [5] *Android Virtual Devices*. (3 de July de 2014). Retrieved from Techotopia:
http://www.techotopia.com/index.php/Creating_an_Android_Virtual_Device_%28AVD%29
- [6] Atzori, L., Iera, A., & Morabito, G. (January de 2014). From "smart objects" to "social objects": The next evolutionary step in the internet of things. *Communications Magazine*, 52, 97-105.
- [7] Bronsted, J., & Hansen, K. (January de 2010). Service composition. *Pervasive Computing*, 9, 62-70.
- [8] Bronsted, J., Hansen, K., & Ingstrup, M. (2010). Service composition issues in pervasive computing. *Pervasive Computing*, 62-70.
- [9] Bruce, J. (18 de July de 2013). *What is Node.JS and Why Should I Care?*
Retrieved from Make use of: <http://www.makeuseof.com/tag/what-is-node-js-and-why-should-i-care-web-development/>
- [10] *Business Management Degree*. (2013). Retrieved from How Does Facebook Make Its Money?: <http://www.business-management-degree.net/facebook/>
- [11] CISCO. (2014). *Cisco iot connections counter*. Retrieved from <http://newsroom.cisco.com/feature-content?articleId=1208342>
- [12] Cui, Y., & Honkala, M. (2013). A Novel mobile Device User Interface with Integrated Social Networking Services. *Int'l J. Human-Computer Studies*, 71(9), 919-932.
- [13] *Document-oriented Database*. (2014). Retrieved from Wikipedia:
http://en.wikipedia.org/wiki/Document-oriented_database

- [14] Edwards, W., Newman, M., Seciivy, J., & Smith, T. (2005). Bringing network effects to pervasive spaces. *Pervasive Computing*, 15-17.
- [15] *Event-driven programming*. (2014). Retrieved from Princeton University: http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Event-driven_programming.html
- [16] Google Inc. (2010). Retrieved from https://investor.google.com/documents/20101231_google_10K.html
- [17] Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29, 1645-1660.
- [18] Guillén, J., Miranda, J., Berrocal, J., García-Alonso, J., Murillo, J. M., & Canal, C. (September de 2013). Architecting Infrastructures for Cloud-Enabled. *Advances in Service-Oriented and Cloud Computing*, 277-287.
- [19] Guillén, J., Miranda, J., Berrocal, J., García-Alonso, J., Murillo, J. M., & Canal, C. (17 de March de 2014). People as a Service: A Mobile-centric Model for Providing Collective Sociological Profiles. *Software, IEEE*, 31(2), 48-53.
- [20] Hirschauer, J. (September de 2013). *What is Node.js and why should I care? An operations guy searches for answers*. Retrieved from AppDynamics: <http://www.appdynamics.com/blog/nodejs/what-is-node-js-and-why-should-i-care-an-operations-guy-searches-for-answers/>
- [21] Hodges, S., Villar, N., Scott, J., & Schmidt, A. (2012). A New Era for Ubicomp Development. *Pervasive Computing*, 5-9.
- [22] Jansen, M. (2012). About using mobile devices as cloud service providers. *CLOSER'12*, (págs. 147-152).
- [23] Klein, A., Mannweiler, C., Schneider, J., & Schotten, H. D. (2010). Access schemes for mobile cloud computing. *First International Workshop on Mobile Cloud Computing at MDM'10*, 387-392.
- [24] Lassila, O., & Adler, M. (2003). Semantic gadgets: Ubiquitous computing meets the semantic web. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*.
- [25] Mäkitalo, N. (2014). Building and programming ubiquitous social devices. *Proceedings of the 12th ACM international symposium on Mobility management and wireless access* , 99-108.
- [26] Mäkitalo, N. (2014). *Collaborative Co-Located Interactions in a Mobile Cloud*. Retrieved from Social Devices: <http://socialdevices.github.io/>

- [27] Mäkitalo, N., Aaltonen, T., & Mikkonen, T. (2013). First Hand Developer Experiences of Social Devices. *Advances in Service-Oriented and Cloud Computing*, 233-243.
- [28] Mäkitalo, N., Pääkkö, J., Raatikainen, M., Myllärniemi, V., Aaltonen, T., Leppänen, T., . . . Mikkonen, T. (2012). Social devices: collaborative co-located interactions in a mobile cloud. *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, Article No. 10.
- [29] McLaughlin, B. (6 de July de 2011). *What is Node.js?* Retrieved from Radar: <http://radar.oreilly.com/2011/07/what-is-node.html>
- [30] Miranda, J., Mäkitalo, N., García-Alonso, J., Berrocal, J., Mikkonen T, Canal, C., & Murillo, J. M. (2014). From the Internet of Things to the Internet of People. *IEEE Internet Computing*, (Submitted).
- [31] *MongoDB*. (2014). Retrieved from Wikipedia: <http://en.wikipedia.org/wiki/MongoDB>
- [32] MongoDB. (2014). *MongoDB*. Retrieved from <http://www.mongodb.org/>
- [33] MongoDB, Inc. (2014). *Internet of Things*. Retrieved from MongoDB: <http://www.mongodb.com/use-cases/internet-of-things>
- [34] *Node.js*. (2014). Retrieved from Wikipedia: <http://en.wikipedia.org/wiki/Node.js>
- [35] NodeJS. (2014). *node.js*. Retrieved from <http://nodejs.org/>
- [36] Oulasvirta, A., Rattenbury, T., Lingyi, M., & Raita, E. (2012). Habits make smartphone use more pervasive. *Personal and Ubiquitous Computing*, 16(1), 105-114.
- [37] Paakko, J., Aalto Univ., A. F., Raatikainen, M., Myllärniemi, V., & Mannisto, T. (December de 2012). Applying Recommendation Systems for Composing Dynamic Services for Mobile Devices. *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, 40-51.
- [38] Papazoglou, M. (February de 2005). Extending the service-oriented architecture. *Business Integration Journal*, 7(1), 18-21.
- [39] Papazoglou, M., Traverso, P., Dustdar, S., & Leymann, F. (2007). Service-oriented computing: State of the art and research challenges. *Computer*, 38-45.
- [40] Raatikainen, M., Mikkonen, T., Myllärniemi, V., Mäkitalo, N., Männistö, T., & Savolainen, J. (2012). Mobile content as a service a blueprint for a vendor-neutral cloud of mobile devices. *IEEE Software*, 28-32.
- [41] Raento, M., & Oulastravira, A. (2009). Smartphones: An Emerging Tool for Social Scientists. *Sociological Methods Research*, 37(3), 426-454.

- [42] Salchan, M., & Negahban, A. (2013). Computers in Human Behavior. *Social Networking on Smartphones: When Mobile Phones Become Addictive*, 16, 2632-2639.
- [43] Satyanarayanan, N. (2001). Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 10-17.
- [44] Tampere University of Technology. (s.f.). *Co-Located User Interaction with Social Mobile Devices*. Retrieved from CoSMo: <http://www.cs.tut.fi/ihte/projects/CoSMo/>
- [45] *The Anatomy of an Android Application*. (23 de June de 2013). Retrieved from Techotopia:
http://www.techotopia.com/index.php/The_Anatomy_of_an_Android_Application
- [46] *Understanding Android Application and Activity Lifecycles*. (1 de July de 2013). Retrieved from Techotopia:
http://www.techotopia.com/index.php/Understanding_Android_Application_and_Activity_Lifecycles
- [47] University of Tampere. (2014). *Social Devices*. Retrieved from <http://social.cs.tut.fi>
- [48] Vazquez, J. I. (2007). A Reactive Behavioural Model for Context-Aware Semantic Devices. *PhD Thesis, University of Deusto*.
- [49] Vazquez, J. I., & López-de-Ipina, D. (2007). Principles and experiences on creating semantic devices. *Proceedings of the 2nd International Symposium on Ubiquitous Computing and Ambient Intelligence*.
- [50] Vazquez, J. I., & Lopez-de-Ipina, D. (2008). Social Devices: autonomous artifacts that communicate on the Internet. *Proceedings of the 1st international conference on The Internet of Things*, 308-324.
- [51] Xing, B., Seada, K., & Venkatasubramanian, N. (2009). Proximiter: Enabling mobile proximity-based content sharing on portable devices. *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications* (págs. 1-3). Washington DC: IEEE Computer Society.
- [52] Zaninotto, F. (2013). *Node.js for PHP Programmers #1: Event-driven programming... and Pasta*. Retrieved from DZone:
<http://architects.dzone.com/articles/nodejs-php-programmers-1-event>

