RICARDO FILIPE ALMEIDA CERQUEIRA DE CAMPOS BARBOSA

SENSING MIDDLEWARE: COLLECT AND SHARE

Orientador: Prof. Dr. Paulo Mendes
Co-Orientador: Bruno Batista

Universidade Lusófona de Humanidades e Tecnologias
Escola de Comunicação, Arquitectura, Artes e Tecnologias da Informação

Lisboa
2012

RICARDO FILIPE ALMEIDA CERQUEIRA DE CAMPOS BARBOSA

SENSING MIDDLEWARE: COLLECT AND SHARE

Tese apresentada na Universidade Lusófona de Humanidades e Tecnologias para a obtenção do grau de Mestre em Engenharia Informática

Orientador: Prof. Dr. Paulo Mendes
Co-Orientador: Bruno Batista

Universidade Lusófona de Humanidades e Tecnologias
Escola de Comunicação, Arquitectura, Artes e Tecnologias da Informação

Lisboa
2012

Dedicatória

À minha mulher Suzana, à minha filha Beatriz e à minha mãe Lurdes por todo o amor incondicional.

Ao meu querido António por teres sido um pai para mim e por olhares por nós onde quer que tu estejas.

Aos meus cunhados Ana e Nuno e aos meus sogros Belmira e Armando por toda a força.

Ao meu pai Filipe pelo carinho e apoio.

Acknowledgements

This dissertation would not have been possible without the existence of some extraordinary people that I am about to mention and express my gratitude.

First and foremost to my advisor, Prof. Dr. Paulo Mendes and to my co-advisor Bruno Batista, for their guidance, dedication and coherence throughout this journey. Also for the trust placed in me by always encouraging me into finding new ideas and solutions and above all for letting my work integrate their research project, CitySense.

A very special word also to one of the most impressive teachers I´ve ever had the privilege to learn from, Prof. Dr. José Rogado, whose quality teachings and ethics constantly guide my professional career and life.

To my classmates, especially Nelson Morais and Macaba Pedro for all that I´ve learned by working with you and for the good times we´ve spent together as colleagues and friends.

To Universidade Lusófona for all the working materials, facilities and excellent working environment that it provides to its students.

Finally, Mr. Anders Hejlsberg from Microsoft for having created such a beautiful and powerful programming language, C#.

Resumo

Barbosa, Ricardo. **SENSING MIDDLEWARE: COLLECT AND SHARE**. Lisboa . 2012, 67 páginas. Monografia, ULHT – Universidade Lusófona de Humanidades e Tecnologias.

Os avanços dos últimos anos nas áreas da micro-electrónica e telecomunicações ajudaram a materializar a visão de um modelo computacional ubíquo, onde através da incorporação de sensores e interfaces de comunicação em objectos do nosso quotidiano é possível em tempo real a troca de informação colaborativa e dinâmica entre os mesmos.

A acompanhar este fenómeno temos assistido ao aparecimento de sistemas que servem de interface a esta rede globalizada de sensores, permitindo não só a disponibilização imediata dos dados sensoriais recolhidos, bem como a interconectividade entre os diferentes dispositivos na rede.

Esta tese faz uma análise comparativa destas plataformas e sistemas, desde a forma como se organizam, suas características, arquitectura de sistemas, utilidade, forças e fraquezas e neste contexto especifica e implementa o projecto Maestroo, um sistema embebido concebido para ambientes sensoriais imersivos baseado na virtualização e partilha de sensores.

O Maestroo permite que dispositivos móveis ou fixos com diferentes sistemas operativos armazenem, disponibilizem e partilhem de uma mesma forma dados sensoriais, bem como o acoplamento de forma virtualizada de sensores que existem fisicamente noutros dispositivos. O projecto cria uma rede privada de três nós e permite a partilha de informação sensorial entre os mesmos. Esta rede foi criada para demonstrar as funcionalidades e capacidades deste software, não estando de qualquer forma restringida a esse número de nós.

Toda a arquitectura, concepção e implementação do Maestroo são descritos pormenorizadamente, bem como as suas aplicações práticas e trabalho futuro.

Palavras-chave
Sensores, Mobilidade, Partilha, Colecta, Colaborativo

Abstract

Barbosa, Ricardo. **SENSING MIDDLEWARE: COLLECT AND SHARE**. Lisboa . 2012, 67 páginas. Monografia, ULHT – Universidade Lusófona de Humanidades e Tecnologias.

The advances of recent years in the areas of microelectronics and telecommunications helped to materialize the vision of a ubiquitous computing model, where through the incorporation of sensors and communication interfaces in objects of everyday life it is possible in real-time, the exchange of collaborative and dynamic information between them.

Accompanying this phenomenon we have seen the emergence of systems that act as interfaces for this global sensor network, allowing not only the immediate availability of sensory data collected as well as interconnectivity between the different devices on the network.

This MSc Thesis makes a comparative analysis of these platforms and systems, from the way they are organized, features, system architecture, utility, strengths and weaknesses and in this context specifies and implements the project Maestroo, an embedded system designed for immersive sensing environments, based on virtualization and sharing of sensors.

The Maestroo software allows mobile or fixed devices with different operating systems to store, make available and share in the same manner sensor data as well as the coupling of virtualized sensors that physically exist in other devices. The project creates a private network of three nodes and allows the sharing of sensory information between them. This network was created to demonstrate all software features and potentialities and it is not limited in any sense to that number of nodes.

All the architecture, design and implementation of Maestroo are described in detail, as well as their practical applications and future work.

Keywords
Sensors, Mobility, Sharing, Collecting, Collaborative

Illustration Index

Table Index

Abbreviations

PCE – Pervasive computer environments

USP – Ubiquitous sensing platforms

Glossary

6LOWPAN: Short for IPv6 over Low Power Wireless Personal Area Networks developed by the Internet area of IETF that advocates that the Internet Protocol can and should be applied to small devices.

ATOM: Atom Syndication Format is an XML based application-level protocol used to publish and edit web feeds.

C++: General purpose intermediate language developed in 1970 by Bjarne Strousstup that added object oriented features to the C programming language.

C#: Object oriented language that is a ECMA standard (ECMA-334) and a ISO standard (ISO/IEC 23270:2006) developed by Microsoft that is shipped as part of .NET Framework.

CSV: Short for comma-separated values.

GZIP: GNU zip data compression library.

HTML5: Latest Markup language for presenting content on the World Wide Web.

GeoRSS: Standard for encoding location as part of a RSS web feed.

Java: Object oriented language, initially developed by Sun Microsystems and now held by Oracle.

JSON: Short for JavaScript Object Notation and although it derives semantically from JavaScript (array literals), it´s language independent and is designed for human-readable data interchange.

JME: Java-based technology that enables the development of software for embedded systems.

Personal sensing systems networks: Dynamic networks whose nodes are composed of heterogeneous multi-sensor personal devices that collect and share sensory data.

Monodroid: Software development platform for android devices that permits the use of C# coding instead of native Java code.

Monotouch: Software development platform for iphone devices that permits the use of C# coding instead of native Objective C code.

.NET Compact Framework: Subset of the .NET Framework that is targeted for mobile device software development.

.NET Framework: Extensive Library developed by Microsoft for software development which includes also an application virtual machine called CLR.

OAuth: Open standard for authorization that allows an applications from a particular domain to access user data stored on other application stored on a different provider without having to hand out their credentials in a user/password style but as a OAuth token.

Objective C: Object oriented language developed and held by Apple.

Protocol Buffers: Google´s structured data serialization protocol.

RDBMS: Short for relational database management system, that is a database management system which is bases on the relational model.

REST: Short for Representational State Transfer that focus on accessing named resources (data) by means of a single consistent interface.

RFID: Short for Radio Frequency Identification an automated identification system through radio signals.

SDK: Software Development Kit, libraries, documentation and utilities for programmers to build new applications.

SOA: Service Oriented Architecture for software development that in the form of services that define business functionalities or components that can be reused.

SOAP: Short for Simple Object Access Protocol for exchange of XML structured information in the implementation of Web Services.

SPA: Short for Single Page Applications, which are web applications composed of a single page that has all necessary its dependencies loaded within a single page load.

SQL: Short for Structured Query Language a declarative search language for relational database management.

SQLite: Open source RDBMS, packaged as a small C programming library that implements most of the SQL standards that is used for local / client storage.

Tcl: Short for Tool Command Language, a scripting language for embedded systems.

UID: User identifier.

URL: Short for uniform resource locator is a character string that constitutes a reference to an Internet resource.

XML: Short for eXtensible Markup Language that is a subtype of SGML produced by W3C in the XML 1.0 Specification and defines a set of rules to encode documents in human and machine readable formats.

XML-RPC: Remote procedure call protocol which uses XML for encoding its calls and HTTP as a transport mechanism.

ZIGBEE: Low-cost and low power wireless mesh network standard based on the IEEE 802 standard for personal area networks.

Index

1. Introduction

The pervasive computing vision of ubiquitous sensing enabled devices and more recently their widespread into everyday personal objects, the so called Internet of Things, has helped to put a spotlight on sensor – based systems and infrastructures.

Also, the generalization of embedded sensors on mobile phones offered a flexible and out of the box way for any person equipped with such a device to collect and share real time multi-sensor information, not only from the physical world, but also from every day's quotidian activities and events, with increasing fidelity and a much higher granularity level than ever before.

Compared to other energy-constrained sensors of more traditional sensor networks, smart phones can support more intensive computations, provide data storage, and offer long-range communication channels and although power management that affects many unattended sensor networks still remains a major challenge for smart phones, one can look at this constraint in a more relativized and benign way, since a phone can be easily recharged in its user's car, office, or home [1].

As this trend emerges mobile phones can be easily seen as collaborating devices that provide sensing coverage across different regions, dynamic points for collecting and sharing data produced by individual sensors or public sensor networks or sensor aware environment enablers through new mobile sensing applications.

Also with the recent phenomena of open source single-board microcontrollers like the Arduino, NetDuino or others where the sensors can be attached in a stackable almost plug-n-play manner, it becomes very easy and inexpensive to build a set of custom made sensor and actuator gadgets.

To realize this opportunity, regular sensor networks and the computational models we use to extract and process this sensor data must address a new collection of challenges. First of the new generation of sensor networks, have mobile devices as nodes which results in hundreds or thousands of individual and concurrent nodes that are likely unmanageable as an ensemble.

On second hand, unlike dedicated and application specific sensor networks where the majority of devices are homogeneous with known configurations, in the shared mobile phone sensor network, devices may be highly heterogeneous, not only in their hardware resources and bandwidth availability but also in terms of users willingness to share their privacy, provide assistance, share battery energy, sacrifice device performance and capabilities, local workload, volatile configurations, unknown delays, etc [1] .

1.1. Motivation

The aforementioned network heterogeneity implicitly enforces us to believe that mobile sensing applications cannot be programmed using traditional distributed computing models which rely on static and stable configurations.

It becomes necessary the existing of middleware that leverages contributions of individual sensor data from a diverse set of devices to intelligently scale computational modeling and classification on the phone.

A common software / hardware complexity abstraction layer able to extract data from different sensors in different devices will allow the processing of similarities in the behavior pattern of people as well as the proximity of people in time and space.

Research on existing projects for sensor networks have shown that they all follow a common denominator, middleware is in the majority of cases, as it will be seen in the related work section, presented as a remote platform or SOA platform. As depicted on Figure 1 sensing devices join these sensing networks through a mid-tier web services layer, building a set of interconnected sensing nodes.

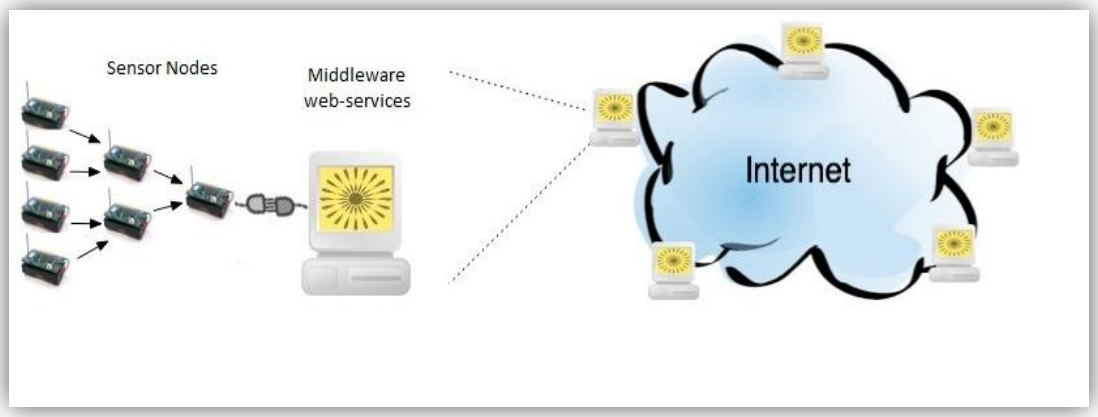


Figure 1: Classic sensor networks middleware architecture

This MSc Thesis approach is quite different because the middleware is device embedded and it´s located at the API of each device to abstract the underlying sensor hardware complexity, data and communication interface as it will be seen in more detail in the architecture specification section ahead.

1.2. Milestones

The first milestone of this dissertation consists in acquiring more knowledge about 4 of the most expressive new generation sensor networks, called ubiquitous sensing platforms (USP) that exist today, and try to understand their unification layer (middleware), if present, that acts as a bridge between the sharing of data and communications between their different mobile nodes.

It was not until recently, by the hand of the MetroSense [2] project, that mobile sensing applications started to be available for the general public in the so called marketplaces. This means that prior to this, USP where being 'fed' by enthusiasts and researchers that built standalone, isolated ad-hoc applications which made unification possible only at communication level (XML, SOAP, JSON, REST).

This first aspect is very important as it is crucial to this report to find these applications in order to draw a parallel between existing mobile sensing middleware architectures and our own that is proposed on this report.

The second milestone of this MSc Thesis was to develop Maestroo, a multi-platform, distributed, device embedded middleware that is expected to run on every device that can process C# language.

This dissertation initial goal was to run Maestroo on Windows phone devices (.NET Compact Framework), Android devices (.NET Framework supported by Monodroid) and Windows workstations (.NET Framework 4.0), so the major challenges faced was code reuse, because this different .NET Framework subsets sometimes forced different implementations for the same interfaces and desired functionalities.

1.3 Objectives

There are two main objectives for this MSc Thesis. First, getting to know in depth existing USP (the ones with most expression and users), their approaches in terms of sensing abstraction layers and second, with the knowledge obtained, understand the strengths and weaknesses of each approach, in order to develop our own middleware that can work on any of this sensor sharing platforms.

This last objective is obtained by building a small private sensor network of three nodes where all devices are running Maestroo and this is achieved by:

- Creating three development environments (virtual machines), one for Windows Phone, other for Android and finaly for Windows workstations;
- Architecting the software interfaces and functionalities;
- Parallel coding and build of main Maestro library on the three environments;
- Single environment interface implementation and unit testing;
- Setting up the private network (addresses);
- Deploying and testing on devices and emulators;

- Buffering sensor data directly to server;
- Exchanging sensors;
- Build a HTML5 Single Page Application (SPA) to interact with the sensed data;

1.4 Structure of the dissertation

This dissertation is organized as follows. Chapter 2 provides background concepts that are absolutely crucial as a framework for analyzing and understanding existing mobile sensor based systems.

Firstly it introduces the concept of Pervasive Computing and how it is at hand with today´s technology, secondly reveals the concept of Sensing Ubiquity and points out the three most common practices under that matter and finalizes with a relatively new concepts called People-Centric Sensing and it´s different usages.

Chapter 3 provides an overview of the 4 most relevant USP available at present date and exposes how sensing abstractions are handled in each one. On Chapter 4 there is an analysis of the different categories for sensing middleware and see which approach is more suitable and complete for Maestroo.

Chapter 5 presents a custom developed sensing middleware, Maestroo , its architecture and development stages, compares it to the existing middleware systems described before and provides an extensive insight of its functional environment and its challenges.

Chapter 6 conclusions are presented, future work and perspectives revealed.

2. Background concepts

This chapter introduces background research on base topics for the dissertation. It presents the concept of pervasive computing, how it is achievable today and what is its main key points and challenges these characteristics bring for applications built to run on these environments. It then approaches the concept of Sensing Ubiquity, showing the most common practices that materialize it and the part it plays in pervasive computing environments

Finally concludes describing a modern tendency in these sensing environments called Cooperative Sensing, it´s general characteristics how it degenerates into People-Centric Sensing and it´s different sensing scale paradigms: Participatory and Opportunistic Sensing.

2.1 Pervasive computing

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it." [3]

Mark Weiser a chief scientist at Xerox PARC, back in 1991, brought up this concept of upcoming technological environments where information processing would seamlessly integrate into everyday object and activities.

Such integration is at hand today, but it was only possible due to the technological advancements in microelectronics, computers getting smaller, more powerful chips and cheaper transistors and also in networking technologies, the efforts for porting the IP stack on embedded devices, the appearance of IPv6 over low power personal area networks (6LoWPAN) , ZigBee (IEEE 802.15.4), etc.

Computer artifacts in ubiquitous environments can range from a simple wrist watch, to wireless mobile device, a TV, a desktop computer or even a simple lamp.

For a better understanding one can imagine an 'intelligent' fridge that automatically fills up a shopping supermarket list as food runs out or a microwave capable of downloading recipes and automatically adjust power and set roasting time.

The term ubiquitous computing has led to a series of subsequent synonyms, *nomadic computing* [4], *pervasive computing* [5], *ambient intelligence* [6], i-centric communication model [7] and all with the same underlying concept of one user, many computers.

2.1.1 Particularities and challenges of pervasive computer environments

The following table summarizes the particularities of pervasive computer environments (PCE) and afterwards there will be an identification of the challenges that each one of them implies.

KEY POINTS	DETAILS
Resource Constraintability	Computing power, energy, low-bandwidth and size.
Node Mobility	Mobile devices as nodes, needs constant re-adaptations.
User-centric	Users are the origin and the targets of sensed data and inferences.
Heterogeneity	Devices, network infrastructures, communication protocols and sensing applications and integration middleware.
Volatile environment	Nodes and resources constantly appearing and disappearing.

Table 1: Particularities of pervasive computer environments

Each and every characteristic of PCE generates a set of incoming challenges. Following the table order above, resource constraints bring one of the major limitations on these types of environments, node uptime presence on the network.

PCE deal with small sized mobile devices, which are also small in computing power, battery sustainability and communication bandwidth and although recently we´ve seen the appearance of dual-core and quad-core phones, until some rather effective battery management and charging techniques become available, e.g. inductive charging or wireless electricity, the constant, computing vs. energy trade-off will always exist. So sensing applications for these environments must counter Moore´s law tendency and trying not to get wantonly computational demanding.

Node mobility means unpredictability, functional environment changes, resource availability degrading, etc. This enforces a constant re-adaptation and re-configuration as well as the ability to anticipate changes for applications running on PCE.

User-centred means that people and their activities are the focus of PCE, so it´s the user's behaviours in the surrounding environment that need constant monitoring. Applications that do these sensing activities should have mechanisms themselves for acquiring this relevant information, like sensor threshold filters or buffering time intervals, in order not only to save battery life, but to avoid network flooding with irrelevant data.

Heterogeneity is just something that PCE will always have to face, so that's why it´s so important the existence of middleware that unifies and leverages contributions of individual sensor data from diverse devices to intelligently scale computational modelling and classification on mobile devices. A flexible middleware able to extract data from different sensors in different devices will allow the processing of similarities in the behaviour pattern of people as well as the proximity of people in time and space.

PCE have their components constantly appearing and disappearing in the network. Device properties, profile and resources are highly volatile, so naming conventions should be adopted instead of IP addresses for a better scalability and use when possible intelligent service discovery techniques. Also this implies that nodes are always encountering entities that never contacted before so it´s very important that these applications can establish trustworthy and secure communication channels as well as guarantee of personal data privacy by limiting its access.

This environment it´s characteristics and challenges, serves as the basis where this MSc Thesis will fit, but even though the terms pervasive and ubiquitous computing are commonly used in an undistinguishable manner, this dissertation adopts the first one, because it seems to mirror a more practical approach for allowing this anytime, anywhere human-machine interaction.

2.2 Sensing ubiquity

Sensing Ubiquity or Ubiquitous Sensing can be defined as the faculty of applications / systems to constantly access data generated by sensors embedded in objects or in the physical world.

Applications that consume and manage this sensor data, are simply called sensing applications. These can range from context-aware applications, to vehicle information and communication systems, to social networking applications, etc.

Basically there are three common practices in ubiquitous sensing on mobile devices, as depicted in Figure. 2, and resumed in this manner:

- Local sensing: ad-hoc sensing modules are deployed to the device, which monitor internal sensor´s activity as well as providing interaction with any type of external sensors embedded in the environment, making raw data available to be consumed by external applications.
- Infrastructure – based sensing: external dedicated sensing servers provide sensor information, discovery, data processing and data storage as a service for applications to subscribe.
- Infrastructure – less sensing: spontaneously formed networks that consist of individual and heterogeneous sensing devices as sensor nodes, gathering, processing and making sensor data available, based on mutual cooperation and interests.

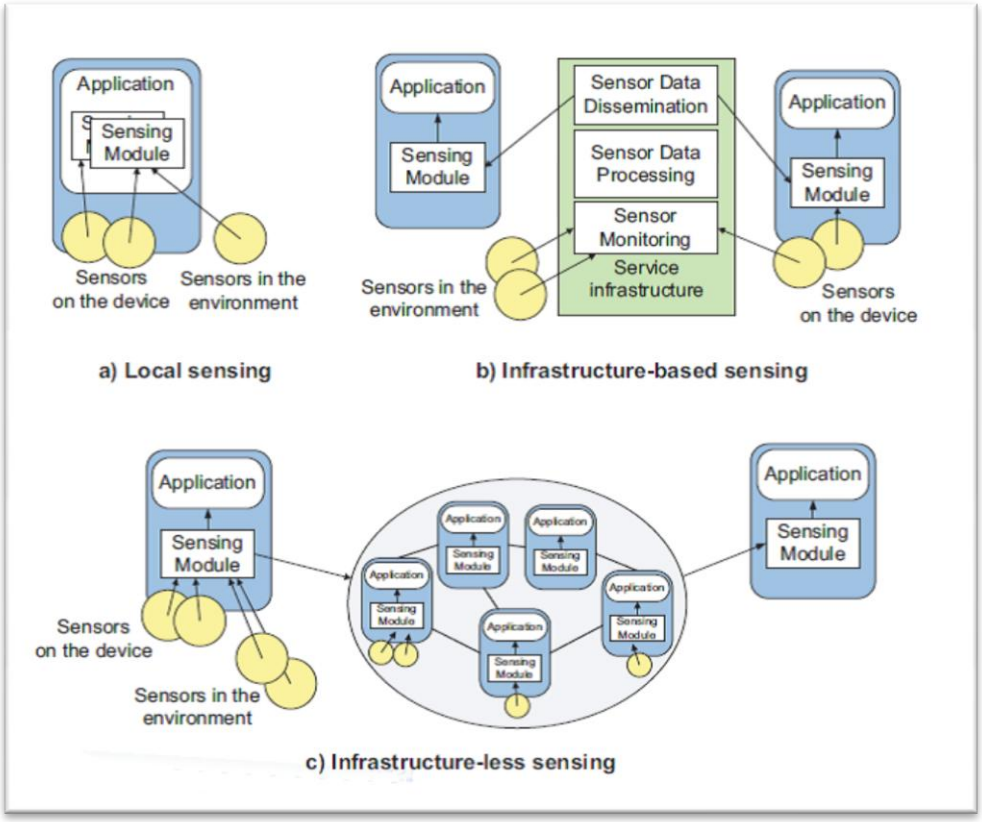


Figure 2: Ubiquitous sensing architecture strategies

2.3 Cooperative Sensing Systems

As general characteristics of Cooperative Sensing Systems we can resume:

- Large scale, long-lived, mostly mobile;
- Application-agnostic;
- Very limited multi-hop wireless;
- Not so energy-constrained;
- Mobility as a driving factor;
- People-Centric;
- Security, trust and privacy are important factors.

2.3.1 People-Centric Sensing

When in a ubiquitous sensing system, individuals rather than trees, buildings or machines, become the focal point of sensing, it´s common to say that such a system is people-centered.

This was a concept that was first introduced in [8] which helped to change the target scope on existing sensor networks that where made of custom devices who operated to solve specialized scientific or industrial issues, to large-scale, heterogeneous networks, addressed to solve urban-scale issues.

According to [8] this concept was already at hand today at it wasn´t necessary to build custom specialized hardware as sensor based devices were appearing everywhere, on mobile phones, PDA´s, MP3 players, small gadgets, etc. Also because communication resources like enterprise and public access wireless networks are well deployed today and it is relatively easy to build small almost instant networks based on these.

This kind of sensing has three main focuses, as observed in Figure.3, but all of them converge to a same objective, to identify behavioral patterns so that we can better understand ourselves and our daily lives, interactions and routines.

Figure 3: The main focuses of people-centric sensing systems

Personal sensing is focused on personal monitoring and archiving, public sensing where data is shared with everyone and finally social sensing were information is shared among common interest groups.

These aspects are not quite well addressed in the existing architectural models for people-centric sensing applications as they don´t treat sensor information as a combined whole, instead each sensor information is treated independently, resulting in poor inferences on raw data.

On the other hand, existing combining approaches are way computational expensive and leave no margin for the phone to do other tasks, limiting its performance and creating people reluctance in adopting them.

This sensing paradigm and its stated challenges are the main objective that this MSc Thesis will address, how to build a device embedded, interoperable, computational model (middleware), that can abstract hardware (sensor and communication interfaces) from
different devices, to intelligently scale and infer, in a resource efficient manner, people-centric sensor data?

A flexible middleware able to extract data from different sensors in different devices will allow the processing of similarities in the behavior pattern of people as well as the proximity of people in time and space to develop, new sensing and learning techniques that exploit the structure present in community sourced data to enable scale, learning algorithms
that adapt and personalize applications, techniques that allow groups of collocated mobile phones to achieve better inference performance and robustness and open sensing and inference software and APIs for the implementation of duty cycled and computational light sensing and inference.

2.3.2 Sensing paradigms: participatory vs. opportunistic

Under cooperative sensing environments two schools of thought emerged. One defends that all sensing interactions at application level must mandatorily need user interaction and approval, which means that all sensing activities and network data pushing is manual, allowing a more controlled data flow and privacy.

On the other hand another current emerged that advocates that all sensing must be opportunistic with no user interaction besides the initial parameterization, allowing much more immediate, precise and contextual sensed data, although a greater weight over the device resources, especially memory and battery, suited ideally for large scale deployments.

More recently in [9] research concluded that an effective people centred cooperative system should not be rooted on a single one of these extremes but sit in the middle, making them in fact complementary and our work helped to prove that concepts because some development SDK´s, for instance, Windows Phone just blocks access to automated features like camera access, microphone access and phone call access.

Figures 4 and 5 depict examples of these sensing paradigms applied in real applications, CenceMe [10] and UrbanSense [11].

Figure 4: Opportunistic Sensing

Figure 5: Participatory Sensing

2.4 Summary

PCE where made possible by Ubiquitous sensing which as it´s turn is essential to the deployment of Cooperative sensing applications. Sensing ubiquity defines the possibility for applications to access at any particular moment or at any particular place, data produced by sensors on other devices in the surrounding environment and by being cooperative and people-centred they allow the sharing of data between them with the purpose of understanding people and their relations with the environment.

This chapter presented all these background concepts that are truly necessary for a complete understanding of our problem space and all the sensing middleware approaches that chapter three presents.

3. Ubiquitous sensing platforms

This chapter tries to identify some real-world ubiquitous sensing platforms and how sensing abstractions are handled at each one, acting as a bridge to the middleware approaches that will be object of study on the next chapter.

3.1 People-centric sensing platforms

People-centric, as seen on the previous chapter involves obtaining sensing data that can help us understand and identify behavioral patterns in our everyday lives, thus on these types of platforms the inference capability is located on everyday personal objects, like e.g. mobile phones, tablets, PDAs, laptops or other, that can be carried by people to enable constant sensing operations.

3.1.1 The MetroSense project

The MetroSense [2] project is an extensive research project at Dartmouth University, which is focused on the already mentioned people-centric sensing paradigm. It´s a closed platform in terms that it hasn´t a public server for sharing sensor data to the world, it´s only for research projects on university campus.

This project proposes a three-tier architecture consisting of sensors at the first tier, sensor access points that collect and store data at the second tier, and servers at the third tier.

Sensors can be static and mobile. Sensor access points enable a secure gathering of sensor data by creating sensing tasks on available sensors for a given application. The mobility nature of the sensor nodes, sensor tasking, data sensing, and data collection occur opportunistically. Servers provide support for the collection of sensed data, data storage, and for running small applications related to a certain MetroSense[2] domain.

3.1.2 Funf: Open sensing framework

Funf [15] is an open source sensing and data processing framework developed by MIT [16] Media Labs and is now a fully commercial product of the spinoff company Behavio [17].

Funf [15] is built on top of the concept of Probes. These are software modules that act as controllers and data collectors for each sensor´s data. Called low-level probes when referring to raw data, and high-level probes when there´s inference on the data involved, e.g. the activity monitor probe that already incorporates motion logic over the accelerometer sensor, allowing for instance a person to monitor its physical activity and to allow others on the network to know if a person is walking, standing, running, lying down, etc.

This means that the data can be consumed immediately by custodians, but also Funf [15] enables sensor data persistence and sharing by allowing data to be saved to an internal database, so that it can be exported to any backend server, to a Drop-Box account, via Gmail, Bluetooth, etc.

3.1.3 Sensing abstraction analysis

The MetroSense [2] project comprises the following mobile applications for the iPhone and Nokia plaftorms:

- BikeNet [12];
- SkiScape [13];
- SoundSense [14];
- CenceMe [15].

Each and every one of these applications has a specific purpose, BikeNet [12] is a mobile sensing system for cyclists for collecting data that quantifies various aspects of the cycling experience, SkiScape [13] a monitoring systems for skiers, SoundSense [14] is a framework for modeling sound events on mobile phones in order to make sophisticated inferences about human activity, and CenceMe [15] is a personal sensing system that enables social network pushes of the user status (sitting, walking, running, happy, sad, at work, office) based on the inferences from device´s raw sensor data.

Figure 6 shows the general architectural model that is used in these applications here depicted for Nokia devices.

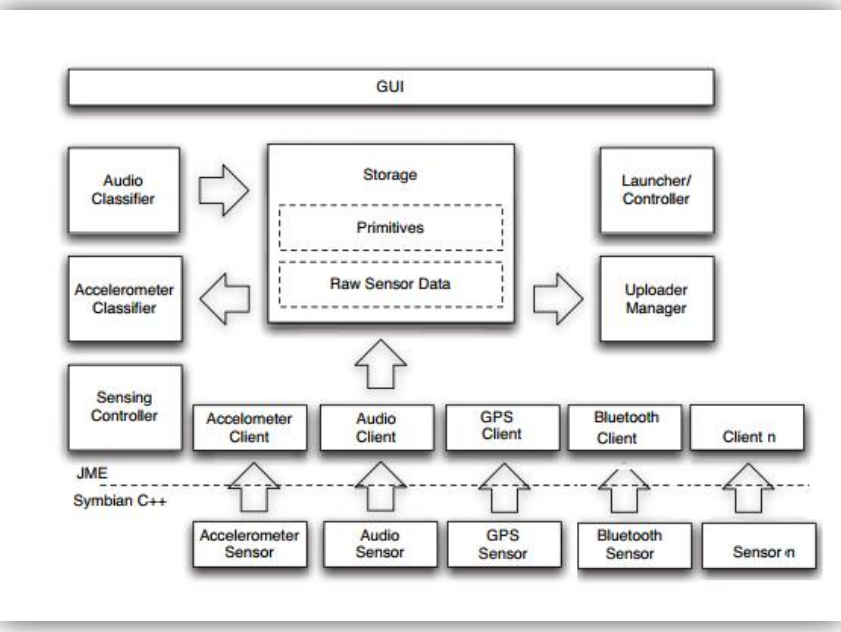


Figure 6: Architectural model of MetroSense [2] project applications.

As it can be noticed from the above figure there are clearly two abstraction layers, the first one comprises all sensor abstraction modules and it is a Symbian C++ module and the second one a JME layer that does the abstractions for clients, controllers, storage, data inference and communications. These abstraction layers fit into the embedded middleware model that is described on chapter 4.

The sensor modules on the first layer act as daemons that produce raw sensor data that is handled to the second layer via socket to its corresponding client. This data is sent as a byte stream which will be stored in local storage to be later processed by the phone corresponding classifier modules (inference on the data is processed locally). As this classification is performed raw stored data is discarded.

The sensor´s orchestration abstraction is done via a Sensing Controller. This module starts stops and monitors all sensor clients to guarantee the proper system operation.

After the inference over the sensed data the connection to backend servers is managed via the Upload Manager. This module abstracts communication interfaces, Wi-Fi, GSM and Bluetooth and pushes the inference over the data (primitives) for classification. Data exchange between the device and backend servers is initiated by the device at regular timed intervals and is done via the Apache XML-RPC library on the server. This local inference feature is something that could be easily seen as an advantage over the solution that this MSc Thesis presents, but clearly it is not.

When these inferences are processed locally there are three main disadvantages, first resource drainage on the device is extremely high, consuming not only more battery power from the phone, but also processor power which can result in having a non-responsive device even for basic operations like answering or making calls. This can easily lead the phone custodian to uninstall the solution. Second these resource constrained devices can never reach the performance or scalability that dedicated backend clusters or cloud platforms can offer, which results in less precise inferences over the sensed data. And finally a local inference limits deploying the middleware only to high-end more powerful devices, and that is certainly one of the reasons why the MetroSense [2] applications are only available for the Nokia N model phones, and the iPhone.

Table 2 resumes these layers and abstraction modules.

LAYERS	**MODULES**
Symbian C++	Sensor hardware abstraction
JME	Sensor clients Controllers Data storage Classifiers Communications

Table 2: MetroSense [2] applications abstractions resume

Concerning the Funf [15] project it comprises the following applications for the Android platform:

- Funf Journal [19].
- Funf in a Box [20];

The first one Funf Journal [19] is an application that comes with 30 built-in sensors (probes) and allows the collection and the sharing of personal sensed raw data via TCP/IP, Bluetooth, SD card and Dropbox [42] accounts. The second one Funf in a Box [20] is a custom version of the first one where the user can choose via a web interface which sensors the application will have at startup before downloading it.

Figure 7 depicts the architectural model for Funf [15] applications.

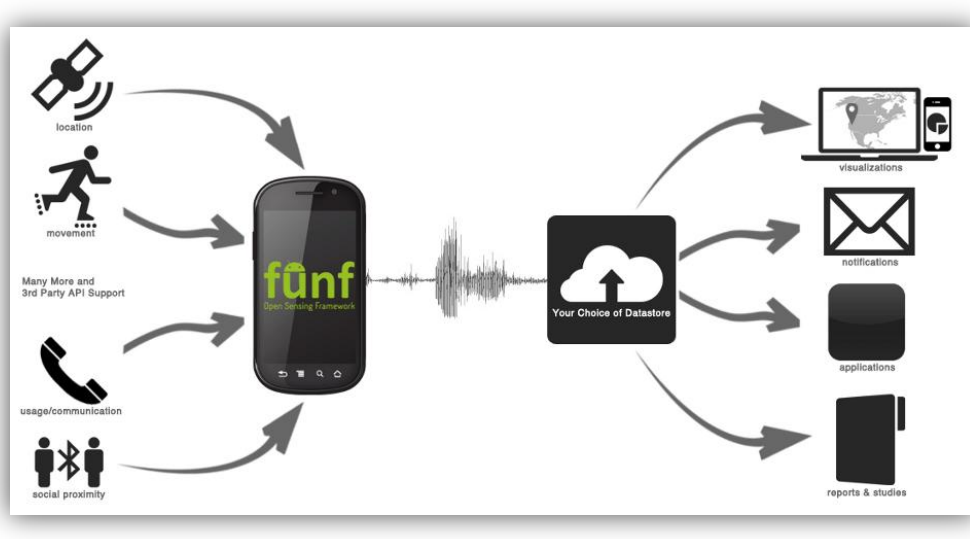


Figure.7: Funf [15] applications architectural model

From Figure 7 flows, it is possible to identify the three areas of abstraction that Funf [15] applications comprise:

- Sensors (probes);
- Storage;
- Communications.

The first layer contains all the logic for sensor hardware abstractions, which are called probes. These abstractions are applicable to the 'real' sensors functional operations, like GPS, accelerometer, gyroscope, etc, but also to other device a 'sensor', like call logs, SMS, Contacts lists, watched videos, etc. Table 3 does an overview of all the probe categories and their usage.

CATEGORY	PROBE	USAGE
Positioning (Use surrounding wireless signals to gather information about a device absolute location and relative location to other devices)	Location	Records the most accurate location available for the device, given device limitations and respecting battery life.
	Bluetooth	Detects Bluetooth devices within range.
	Wi-Fi	Records available Wi-Fi access points.
	Cell	Records ids for the current cell tower the device is connected to.
Social (Record information about relationships and communication between people)	Contact	Detailed information about the contacts available to the device. Sensitive information is one way hashed to ensure user privacy.
	Call log	Records the calls that are made by the device. Sensitive information is normalized and hashed consistently and can be compared to contacts on this device, or with other devices.
	SMS	Messages sent and received by this device using SMS. Sensitive data is hashed for user privacy.
Motion (Capture a devices motion to understand how a user is moving)	Accelerometer	Measures the acceleration applied to the device.
	Gravity	A three dimensional vector indicating the direction and magnitude of gravity
	Linear Acceleration	Records a three dimensional vector indicating acceleration along each device axis, not including gravity.
	Gyroscope	Measures angular speed around each axis.
	Orientation	Measures orientation of device.
	Rotation	The rotation vector represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle θ around an axis <x, y, z>.
	Activity	Records how active the person is. Uses the AccelerometerProbe data to calculate how many intervals the

		variance of a device's acceleration is above a certain threshold.
Environment (Discover details of the ambient surroundings of the phone)	Light	Detects the ambient light level.
	Proximity	How far the front of the device is from an object.
	Magnetic Field	Detect the local magnetic field.
	Pressure	Records the pressure on the touch screen of the devices.
	Temperature	Used to record temperature.
Device (Device Information)	Android Info	Information about the version of Android the device is running.
	Battery	Information about the type and current state of the battery in the device.
	Hardware Info	Details about the specific hardware the device is running, including component identifiers.
	Time Offset	Checks NTP servers to determine the devices current offset from the time servers.
	Telephony	Records telephony hardware, software, and account information.
Device Interaction (Information about user behavior on their device)	Running Applications	The current running stack of applications.
	Applications	What applications are installed on the device? Also specifies applications that were uninstalled.
	Screen	Records when the screen turns off and on.
	Videos	Information about video files on the device.
	Audio Files	Information about audio files on the device.
	Images	Information about image files on the device.

Table 3: Funf [15] Probes

The second layer encapsulates all data-persistence and management operations that function as file agnostic services and database specific services. The raw sensor data can be persisted to internal disk storage, SD cards and internal SQLite encrypted database files. This abstraction is done via a Database Service module recording service.

The third layer is responsible for all the communications of the application with backend services for data sharing. This abstraction is done via a Remote Archive module that acts as an interface for sending files to a remote server and an Upload Service module that enables file uploading to DropBox [42] accounts or to a URL. These abstraction layers also fit into the embedded middleware model that is described on chapter 4.

Funf [15] application don´t do inferences over the sensed data, the raw data is just exported for third-party application to consume as in Maestroo.

Table 4, below, resumes these layers and abstraction modules.

LAYERS	**MODULES**
Probes	Sensor hardware abstraction Device specific functionalities abstractions
Database Service	Data-persistence Data-management File encryption
Remote Archive Upload Service	Communications Http file uploads

Table 4: Funf [15] applications abstractions resume

As already mentioned, this abstraction analysis on Metrosense [2] and Funf [15] applications revealed that they both share an embedded middleware approach that will be object of study on chapter 4.

What is interesting to retain is that an architectural pattern can be drawn when thinking about creating an embedded abstraction model, sensors, storage and communications are three common layers of abstraction.

3.2 Web sensing platforms

A Web sensing platform differs from a people-centric one because the scope of the platform is not restricted to personal sensing operations.

Generally speaking, these platforms can comprise almost any type of sensor and sensor data and are normally open platforms that expose their services via an http URL.

3.2.1 SenseWeb

Microsoft Research [21] SenseWeb [22] is a global-scale ubiquitous sensing platform (USP) that provides a web-based platform and tools for collecting, sharing, processing and querying sensor data produced by network peers.

It´s goal is to provide open-source architecture for anyone to register sensors or sensor data repositories and also 3rd party sensor application development that uses these shared sensing resources. Unlike the last two projects where there was a development of sensing applications, here the middleware is web-based and its intent is to become a global sensor data indexing engine who acts as service for deployed sensors and sensing applications worldwide.

3.2.2 Cosm

Cosm [24], formerly known as Patchube is probably the greatest and best known sensor sharing platform and community in the world, now acquired by the LogMeIn [25] Company.

Using Cosm´s API users can create new sensing applications and devices, use the platform´s backend infrastructures for data management and storage and share their sensed data to the world.

The main goal of this USP is to connect devices and applications for secure data storage and exchange. Cosm [24] has also a commercial side to it, a provisioning service for companies to pair their client´s devices with custom applications to control or use the sensed data.

3.2.3 Sensing abstraction analysis

On these types of platforms there are really no applications defined, they just act as enablers for data-sharing to occur and on some of them small widgets like graphical analyzers or maps are used to do small inferences on the data, therefore abstractions must be explained at the platform architecture levels.

SenseWeb [22] is built upon a layered modular architecture as shown in Figure 8. The different colors around each module mean for the blue ones that they are already provided by the solution and the white ones show that they can be 3rd party extensions.

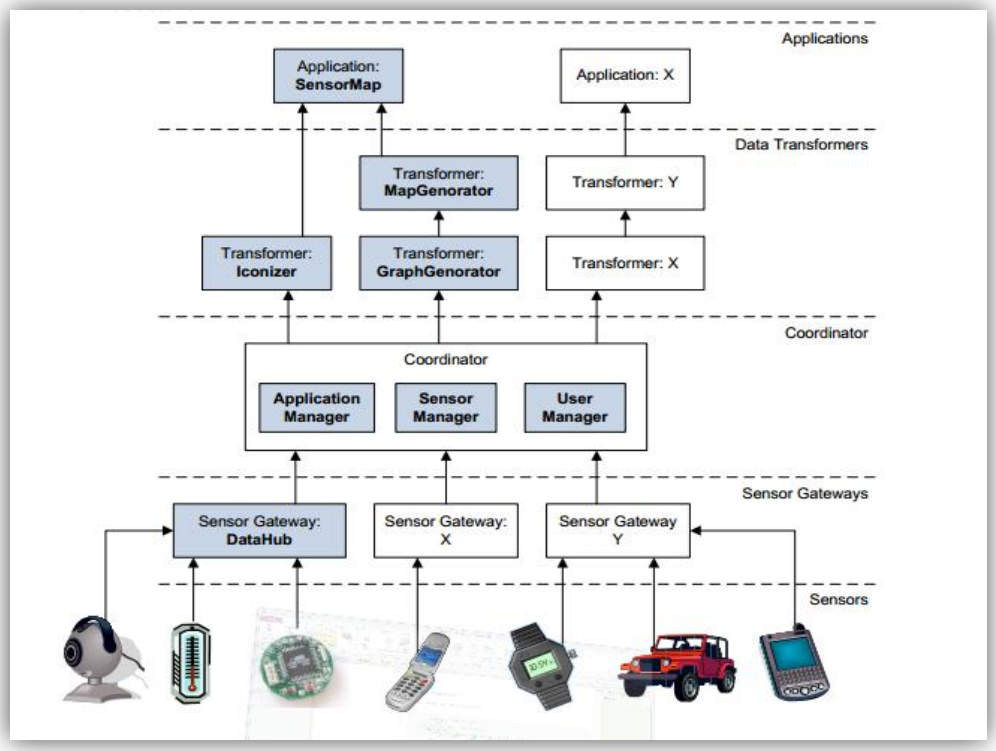


Figure.8: SenseWeb [22] Architecture

The Coordinator module is the main entry to the system for sensing applications and it contains three components:

- User abstraction and manager, web service that provides authentication mechanisms;
- Sensor manager, is an indexer of all available sensors and their properties that maps user sensor descriptions to physical sensor id´s;
- Application abstraction and manager is the central point of entry for sensing data to the upper layers, manages sensor gateways (data providers) and resolves sensing queries from the upper layers by searching existing sensor resources.

Sensor Gateways modules are common interfaces for all sensor data to reach the above layers and abstract sensor differences. The Gateways also implement sharing policies defined by sensor owners. SenseWeb [22] platform already implements a default Gateway, the Datahub. This interface provides network peers who do not want to implement their own gateways to push sensor data into the USP via Datahub´s web service. It´s a messaging abstraction layer.

Finally, Data Management module involves storing and retrieval of sensor data and it's done by invoking the required web methods on the sensor gateway web-service. Also it is possible to query sensors for a specific type and sensor data.

Table 5 resumes these layers and abstraction modules.

LAYERS	MODULES
Coordinator	User abstraction Indexing operations Application abstraction
Sensor Gateways	Sensor abstraction Messaging abstraction
Data Management	Data-persistence Data-management

Table 5: SenseWeb [22] platform abstractions resume

Cosm´s [24] entry point to the platform are its HTTP based RESTful services that it exposes and are accessed via the development API, which is uses cURL as a transfer library for managing application-level protocols.

Cosm [24] is also based on a modular architecture. Environment module, also known as feed is responsible for sensor data and metadata (context) collection and messaging abstractions. DataStream module represents individual sensor abstractions and represents sensor measuring (raw data) within an environment. Trigger module is the communication abstraction layer and is responsible for controlling data push requests to a specified URL. User module is responsible for abstracting users and managing authentication and finnaly, the Data management module for abstraction and management of data-persistence and monitoring.

Table 6 resumes these layers and abstraction modules.

LAYERS	MODULES
Environment	Sensor data and metadata Messaging abstractions
DataStream	Sensor abstraction
Trigger	Data-persistence Data-management
Users	User abstractions and management
Data management	Data storage abstraction Data management

Table 6: Cosm[24] platform abstractions resume

This abstraction analysis on SenseWeb [22] and Cosm [24] applications revealed that they both share a SOA middleware approach. This approach´s characteristic will be object of study on chapter 4.

3.3 Summary

Overviewing the most relevant USP, how they're built, how they work and above all how they interconnect different devices and share sensor data is key for having a global understanding of the middleware approaches that is needed and used for each one. As already mentioned chapter 4 will go deeper into analyzing these approaches and try to find strengths and weaknesses of each one.

4. Sensing middleware

A lot has been said and done concerning sensing applications or sensor sharing information platforms, but what is the most used middleware architectural models that these USP platform use? What distributed programming abstractions are adopted? Here are some of the widely used approaches (the first two) that were mentioned in the previous chapter as well as others common practices (the last two) we encountered along our research

- SOA Platform: [22], [24], [28] and [30] as we will see on the next chapter all have their abstraction layer via a set of web services middleware. By releasing an API that gives developers access to the specifications and encodings (XML, JSON, ATOM, etc) these services need, interfacing can is easily done on all kinds of heterogeneous systems and sensors. This unification is thus made at the messaging level, so we categorize it as message-oriented middleware (MOM).

- Device embedded: [2] and [15] rely on a distributed computing model where the abstraction is located at the device-level, so we call it device-level integration. This approach implies the existence of an n-tier software system. The middleware must act as an entry point to each and every system layer, providing communication services and integration interfaces, software/firmware that operate between embedded applications and the system as a whole. This is the approach we´ve chosen to build Maestroo upon.

- Distributed databases: Some approaches like [33,34,35] allow users to address to the system in a subset of SQL-like queries for a particular sensing task. An example of such technology is TinyDB [36] which has a single sensors database table where each column represents a specific type of sensor or other input data i.e. sensor node identifiers, battery and computational power, etc. This middleware uses a decentralized approach where each sensor node contains its own query processor that processes and aggregates sensor raw data. For query execution the following steps are required, (i) construction of a spanning tree of the network with the root at the user endpoint, which is maintained using a controlled flooding approach, (ii) starting from the root each node makes the decision if queries are executed locally or broadcasted to all network nodes, with a time interval being agreed between parent and child nodes for data listening,(iii) processing of the query at node level with an acquisition periodically performing sensor measurements and finally (iv) data aggregation for resource savings in the network by pushing computing to the network towards the origin of the data to be processed.

- Mobile code and agents: [37] and [38] are representative examples of a class of middleware that relies on mobile code and mobile agents. By small scripting (an extension of Tcl [39]) program injections on the network, sensor data is collected on each node and program replication is made along other nodes. SensorWare [16] is the platform that makes this possible, by implementing a set of additional procedures in the Tcl interpreter. Example commands are query (which takes a sensor name and a command as parameters and perform operations on sensor as the reading of data), send (takes a node address and a message as parameters as sends a message to sensor nodes), wait (suspend

the execution of the script until some event occurs) and replicate (copies the script onto other nodes).

4.1 Middleware feature comparison

Comparing features of middleware architectures and realizing their strengths and weaknesses according to our purpose, was the first step I chose when starting this MSc Thesis investigation. Only afterwards we were able to elect the embedded architecture as the one that better served our ambition for building Maestroo.

The first approach, SOA, as we´ve seen, is the most adopted architecture for USP and that does not happen by chance, it happens mainly because it´s the quickest way that any sensor network can abstract client differences and therefore be more widely adopted.

By relying on well-tested and robust web service messaging standards (SOAP, XML, JSON, ATOM, etc) and having the HTTP protocol at the application level, network constraints like concurrency (handled by http application servers), security (e.g. port restrictions) and interoperability (it doesn´t matter what is the client´s OS or underlying system type as long as messaging standards are respected) are easily managed.

Operating at the messaging-level has however a big catch, the system has to be made high-available as state management mechanisms are delegated to the calling clients, downtime issues can easily compromise the overall system´s trust and functionality. Fallback mechanisms must be a major concern when adopting this option.

These mentioned general characteristics make this middleware model best suited for the creation of static USP with less controlled sensor nodes.

On the other hand, the second approach, an embedded middleware, is a more embracing solution for creating abstractions for USP. As all participating systems and their functioning are known beforehand (as there has to be specific developments on top of each one) there is not only more knowledge of the system as a whole, as well as more control over its distribution (it only works on and for the platforms where the middleware is running).

Also, by layering sensors abstractions, control and communication on top of the targeted OS (with developer SDK´s) in an easy to use way, it´s ensured that every feature that makes a USP more robust is present at the core of each network node. Features like managing resource constraints (energy, computing power, memory and communication bandwidth) by ensuring that all node middleware components are lightweight is something you can´t control on a SOA approach.

This middleware option although exhibits a tradeoff between completeness and complexity (every feature that makes the system more complete has to be custom developed thus increasing overall solution complexity) is also far more suited for the new generation of USP that have highly dynamic, collaborative (almost in a peer-to-peer mode) and volatile network nodes.

The distributed database approach relies on very easy to use SQL-like declarative query syntax and covers issues related with the system´s distribution from the user. Instead of developing for particular sensor node the network is conceived and developed as a single virtual entity (all nodes of the sensor network are integrated into a virtual database system where the stored data is distributed from a node to another). There are no interfaces or API´s at the node level only a custom query relational database engine.

On the other hand this architecture´s extensibility is limited in a way that for adding new query operations to the localized engine requires modifications on the query processor of all sensor nodes.

Scalability is also affected as these types of systems tend to maintain a wide spanning network tree by replicating a query on all network nodes but in contrast, many sensing operations and tasks have few nodes involved.

The mobile code and agents middleware makes use of an imperative language and dynamic scripting for creating sensing tasks on individual nodes. Even a simple sensing operation can result in large scripts that don´t interface with the OS(querying a device for sensors), network(sending and receiving messages).

Contrary to the previous (database) approach there is no need to update every node when there is something new or some sensing task, just by injecting a dynamic Tcl script via a mobile agent this operations can be easily accomplished. However, as the performance of interpreted scripting languages isn´t sometimes optimal these types of middleware implement complex signal processing operations which also imply changes on the runtime environment.

Table 7 highlights the strengths and weaknesses of the analyzed middleware approaches.

Middleware approach	**Strengths**	**Weaknesses**
SOA	Widely distributed; Quickest way to abstract device differences; Built on top of standards (SOAP, XML, JSON, ATOM); Highly scalable; Suited for stable environments;	Messaging-level interface only; High availability and fallback mechanisms are mandatory; State management is delegated to clients, thus making it less reliable; Data-only access and not hardware access; Unidirectional (reactive)communication;

		Online (network connectivity) operational mode only;
Device-embedded	Device-level interface; Interoperability; More control over well-known constraints (energy, computational power and communication efficiency) as all systems are unified to operate in the same manner; Can use standards as well as custom developed protocols; Data and hardware access; Bi-directional communication; Efficient state management; Online and offline operational modes; More knowledge of the system as a whole; Controlled distribution;	Harder to implement and to distribute; Requires a deep knowledge of the systems where it is implemented;

	Highly extensible. Suited for high volatile environments;	
Distributed databases	Database-like network abstraction; Easy to use SQL-like query engine; Suited for more stable environments; Highly limited sensing functionality (pre-defined query operations).	Limited extensibility and scalability; Data focused System updates or upgrades need to be done at each node level;
Mobile code and agents	Dynamic scripting language enable a more "upgradable – friendly" system; Network is reprogrammed by injecting new agents;	- Interpreted languages normally have slower performance for computational intensive operations; - Sensed data focused;

Table 7: Strengths and weaknesses of analysed middleware

As shown in Table 7, embedded middleware despite being a more laborious and complex approach is by far more complete than the others. By abstracting features (devices, sensors and communication interfaces) directly at each hosting device not only more granular device knowledge is obtained but also a complete customization of how these interfaces will be and behave can be achieved.

Contrary to a SOA that functions in a reactive, unidirectional manner, with no knowledge beyond messaging protocols (clients send sensed data to the network obeying a set of predefined messaging standards), an embedded middleware can let USP function in both ways reactively by responding to requests and actively by issuing requests to other nodes.

An embedded middleware can comprehend all of the other approaches best features, but the reverse is not the case.

The embedded approach is thus the approach chosen to build this project´s immersive sensing middleware tool, Maestroo that will be object of this MSc Thesis study on chapter 5.

4.1.1 Major challenges

A proficient embedded middleware should provide application developers the ability to interact between different software modules with a more functional set of interfaces than those the OS´s and network services natively implement.

The specific and exigent requirements of people-centric mobile sensing applications place new challenges on middleware developers. Among the several challenges we present the most important ones:

- Network Inconsistency – Mobile sensor networks are made of divergent nodes with inconstant configurations and uncertain delays, so, more traditional distributed computing models which rely on stability and well known configurations cannot be adopted. Also, this type of networks evolve in such an unpredictable manner that makes almost impossible to know the exact number of resources as well as their exact location. So, more flexible and adaptive approaches are required;

- Data Fidelity – Having uncoordinated mobile sensors can result in a highly time and space heterogeneous sensing coverage which can compromise data fidelity. For example several people can sense a city location several times and rarely sense other parts, or sensing quality can be more precise during rush hour but not on other time periods, so the quality of the produced results should be taken into account when designing the middleware;

- Network Device Naming – On more traditional distributed computing models, bindings where fixed between network device names and node addresses and this approach is very inflexible for mobile sensing networks where mobile nodes are set based on their contextual or environment properties, i.e. available sensors, node location, device computational resources, available energy, etc. Fixed naming schemes are inappropriate for these environments, therefore data-centric or property-based naming becomes a necessary approach.

- Resource Constraints – Sensing on mobile phones must be seen as secondary tasks and not place a heavy computational burden on hardware thus compromising the phone´s primary processes, i.e. making a call or synchronizing e-mail, etc. There is a need to create some sort of dynamical resource management approach that can compensate the trade-off of having constant power intensive sensing tasks by, for instance, reducing the sampling frequencies when battery is low or by any other kind of physical resource measurement strategy.

- Intensive Data Traffic – Handling concurrency is surely not an easy task and that´s why it becomes very important to have a very consistent communication´s
  layer that can manage and prioritize tasks in order to avoid redundancy and also make a distinction between urgent or critical applications. Traffic payload can be
  reduced by doing some sort of data aggregation algorithms in order to reduce the number of packets transmitted.

4.2 Summary

This chapter highlighted the major characteristics of each type of middleware approach, exposing their strengths and weaknesses, and justifying why an embedded middleware is a more complete approach to conceive an USP than the other.

It also revealed the major challenges that must be taken into account when designing a middleware system.

5. Maestroo

This report until now, has tried to identify and understand mainly 2 things. Firstly the functionality of the most expressive USP and secondly what middleware approaches are used in these types of sensor networks, their general characteristics and also other possible types that exist and can be considered.

Maestroo is an embedded type middleware, because it runs locally on the sensing device and has all the layers of abstraction that were identified as a pattern, on chapter 3, recalling, sensors, storage and communications and is intended to work for USP.

But Maestroo takes a step further because neither the MetroSense [2] nor Funf [15] projects are designed to work for more than two different OS´s, iPhone and Nokia for MetroSense [2] and Android for Funf [15] and it is this MSc Thesis pretention to build a custom middleware, entirely with the C# managed language that could be distributed to all devices that can run the .NET Framework on their OS, these devices include Windows phones, Android phones, iPhones, Android Tablets, Windows workstations and Linux workstations.

Maestroo brings novelty factors onto the embedded middleware approach which are the ability to create device profiles, letting users select which types of sensors and network interfaces to expose to the network and also sensor virtualization that enables the borrowing and incorporation of external sensors onto a physical device, as well as a highly dynamic and loosely coupled architecture, built on top of Dependency Injection principles that make the solution very extensible and highly scalable as chapter 5.1.3 will present in more detail.

5.1.1 Proposed architecture

The higher-level conceptual design for our middleware relies on a layered architecture approach. This not only helps to manage and control the development lifecycle in a more efficient way but also because it facilitates to reach the loose couplings and the abstraction levels that are pretended to achieve.

Figure 9 depicts that layered architecture.

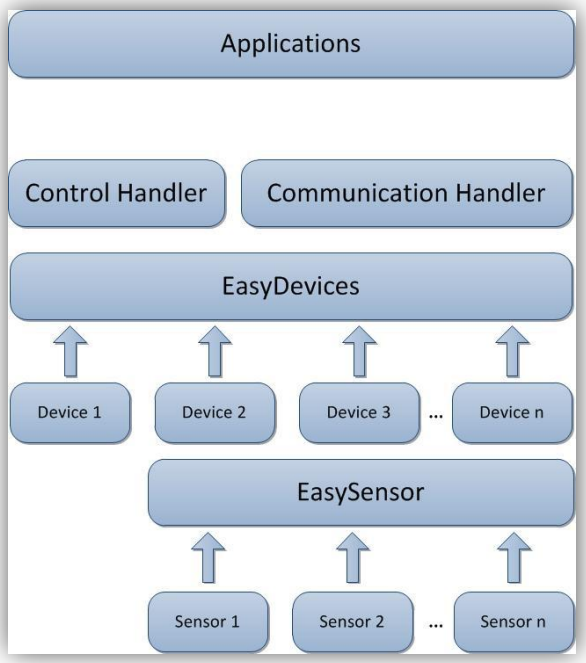


Figure 9: Middleware building blocks

As shown in the above figure, the middleware model is based on four layers:
1. EasySensor;
2. EasyDevices;
3. Control Handler;
4. Communication Handler;

The first layer concerns building a common interface for multiple sensor availability on different devices and will serve as the basis for the creation of the necessary connectors to both real and virtual sensors, these last ones are sensors that are 'borrowed' from other devices on the network.

The second layer allows supporting different devices and their specific capabilities in terms of sensors, communication and computational resources. This layer abstracts their differences and provides transparent access to the device functionalities. Also it enables the creation of different device profiles based on dynamic sensor and communication interfaces discovery during boot time.

The third layer instantiates the devices and sensors and is responsible for the control of messages fluxes between interfaces. This message fluxes are made through an XML protocol with the potential to be encrypted.

Finally, the fourth layer is responsible for managing the communication interfaces and communication protocols.

5.1.2 Development environment

In order to be fully compliant with all the OS that Maestroo is intended to cover, recalling, Google´s Android, Apple´s iPhone, Microsoft´s Windows Phone and Secret Labs NetDuino, only a managed language like C# and .Net Framework make that possible as Java is limited to devices that run it natively.

The last two devices natively run the framework but to reach a cross-platform development environment for Android and iPhone we need another framework that can compile .Net assemblies to Java, and Objective C, respectively and that framework is Xamarin´s Mono for Android and MonoTouch.

As the project evolved and the code development was being done, some practical difficulties were encountered that put on hold porting the code onto the Netduino boards. These boards run a very minified version of the .NET Framework, called .NET Micro Framework that miss important assemblies that were needed, e.g. the use of generics for 'open types' that can be avoided on our custom code but exist on third-party assemblies that are required for this project as a whole.

The IDE of choice was Visual Studio 2010 and the development VM´s were Quad-Core Windows 7 machines with 4096MB of system RAM and 70GB of total hard disk space. All the development platforms have built -in emulators, so the initial development stage was based around them, but as the work evolved, Maestroo was deployed onto real Windows Phone devices but in a near future the planning is to deploy all into real devices.

5.1.3 Maestroo relevant software engineering aspects

This section will describe the most relevant software engineering aspects behind Maestroo middleware, first its dynamic nature, remembering, sensors and communication interfaces are bootstrapped at runtime, they are seen to the application as plugins for creating a device profiles, thus it was necessary to create the least dependencies possible when wiring all the modules together. The way to do this is through component containers, which are part of a common software engineering pattern called Dependency Injection (DI), also known as Inversion of Control (IoC).

DI enables registering concrete interface implementations on a container and injecting them when appropriate, resolving all the dependencies needed whether they are constructor parameters or properties in a class. To be able to do this in a distributed manner TinyIoC [43] library was used because this library compiles on all the subsets of the .NET Framework that Maestroo needs and also because is very simple and lightweight.

Data persistence is also a characteristic that is worth to accentuate. The reason for that was because the Windows phone OS doesn't natively work with SQLite databases that are the standard for all other targeted platforms and this could lead to a totally different implementation of the data handling module for this platform.

Vici.CoolStorage [44] was the library that made this unification possible by providing a set of wrapper methods for windows Phone to work in a SQLite manner with local database storage.

Finally Maestroo object serialization features. The options where use the .NET Framework built-in serializer classes (XmlSerializer, DataContractSerializer, NetDataContractSerializer, BinaryFormatter and JavascriptSerializer) or use Google´s Protocol Buffers [45]. The choice was influenced by a series of benchmark tests which were done by Marc Gravell [45], the creator of protobuf.net [45], a .NET framework library that implements Protocol Buffers [45] and are depicted in Figure 10. Length in the figure is the data payload in bytes after serialization of the same object occurs, and serialize and deserialize mean the duration in milliseconds of that same operations.

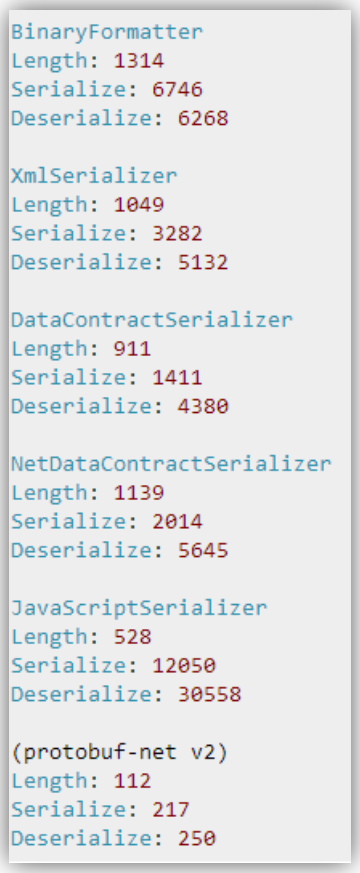


Figure 10: Serializer benchmarks

As benchmarks showed besides being twenty to one hundred times faster the payload is three to ten times smaller which in small resource constrained devices, makes all the difference, so Protocol Buffers [45] was the technology used for serializations and deserializations of objects on Maestroo.

5.1.4 Code structure

Solution-wise Maestroo divides itself into 7 projects as Figure 11 demonstrates. These are:

- Maestroo
- Maestroo.Bootstrapper
- Maestro.Core
- Omnia7
- Omnia7.Accelerometer
- Omnia7.GPS
- Omnia7.Wifi

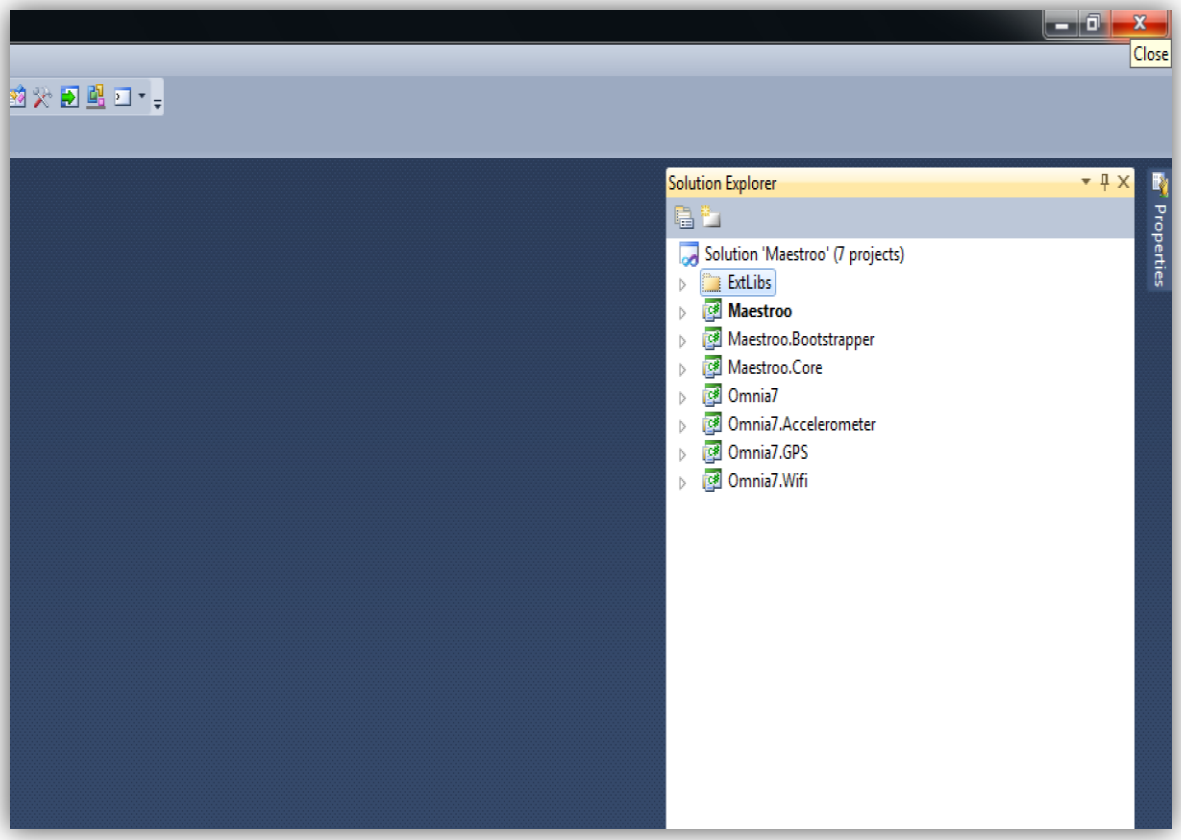


Figure 11: Maestroo project solution

5.1.5 Maestroo module

The first module, Maestroo is the starting project and it is the Windows Phone specific assembly. It´s on this project that all remaining assemblies (internal or third-party) get referenced.

This project contains all the presentation layer (UI). The main user interface has two main screens one for the existing sensors and other for server network nodes (Figure 12). On the sensors screen (Figure 13) it is possible to:

- Enable or disable a sensor (start and stop sensor readings);
- Configure the sensor settings (broadcast interval and see the sensor uuid)
- Add a virtual sensor (encapsulate sensors from other devices on the network).
- Configure general settings (see the device uuid, type, brand and and model, define the current server address and port, see the device profile, if it is a client or a server, set the in memory maxsize and select the export types, xml files, socket or internal database (Figure 14).

On the network screen it is possible to:

- Search the network for server nodes (devices that have a listening socket);
- Dump offline readings (push internal memory persisted readings to server);

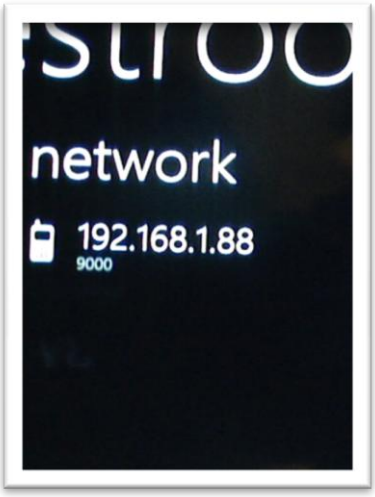


Figure 12: Network screen

Figure 13: Sensor screen

5.1.6 Maestroo.Bootstrapper module

Maestroo.Boostrapper module is responsible for hardware discovery which is done through runtime type bindings previously defined on an existing xml settings file (Figure 15). This settings file act as a device profile, because all available sensors and communication interfaces must be specified here.

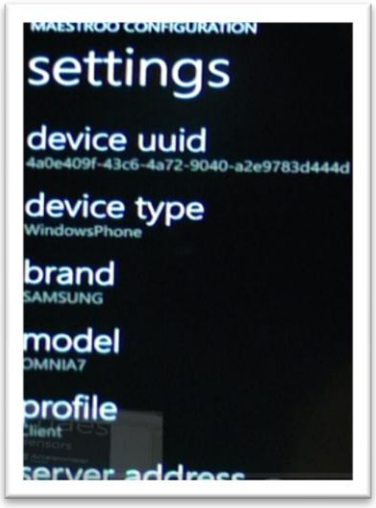


Figure 14: General settings screen

The Run() method (Figures 16, 17 , 18 and 19) on this class injects all the dependencies (properties, constructor and parameters) needed for the three core layers of our application, device layer, sensor layer and communications layer and wraps all those layers into a concrete device.

```
<?xml version="1.0" encoding="utf-8" ?>
<device supportedtype="1" profile="1" outputs="1-2-3" serveraddress="192.168.1.66"
serverport="5000" outputdir="Dumps" maxsize="30" namespace="Device" class="Omnia7"
assembly="Omnia7">
  <sensors>
    <sensor supportedtype="2" interval="8000" namespace="Omnia7.GPS" class="GPS"
assembly="Omnia7.GPS" />
    <sensor supportedtype="1" interval="8000" namespace="Omnia7.Accelerometers"
class="Omnia7Accelerometer" assembly="Omnia7.Accelerometer" />
  </sensors >
  <adapters>
    <adapter supportedtype="1" namespace="Omnia7.Wifi" class="Wifi"
assembly="Omnia7.Wifi" />
  </adapters>
</device>
```

Figure 15: Bootstrapper xml setting

```
public IDeviceAbstraction Run()
        {
            try
            {


                //read settings file that must be embedded as a resource
                StreamResourceInfo xml = Application.GetResourceStream(new
Uri("/Maestroo.Bootstrapper;component/settings.xml", UriKind.Relative));
                var settings = XDocument.Load(xml.Stream);

                //get phone attributes for type creation
                var phoneNameSpace =
settings.Element("device").Attribute("namespace").Value;
                var phoneClass =
settings.Element("device").Attribute("class").Value;
                var phoneAssembly =
settings.Element("device").Attribute("assembly").Value;
                var phoneSupportedType =
settings.Element("device").Attribute("supportedtype").Value;
                var phoneProfile =
settings.Element("device").Attribute("profile").Value;
                var phoneServerIp =
settings.Element("device").Attribute("serveraddress").Value;
                var phoneServerPort =
settings.Element("device").Attribute("serverport").Value;

                var phoneOutput =
settings.Element("device").Attribute("outputs").Value;

                var phoneOutputDir =
settings.Element("device").Attribute("outputdir").Value;
                var phoneOutputMaxSize =
settings.Element("device").Attribute("maxsize").Value;

                //get phone type
                var phoneType = Type.GetType(phoneNameSpace + "." + phoneClass +
", " + phoneAssembly);

                //Register phone type with IoC container
                TinyIoCContainer.Current.Register(typeof(IDeviceAbstraction),
phoneType, phoneSupportedType).AsMultiInstance();
```

Figure 16: Bootstrapping types

```
  if (_manager.DeviceExists()) //Load
                {
                    this._bootstrappedDevice =
GetDevice(_manager.ReadDeviceProperties());
                }

                else //Create new device
                {
                    string[] properties = new string[8];

                    properties[0] = phoneProfile;
                    properties[1] = phoneSupportedType;
                    properties[2] = phoneOutput;
                    properties[3] = phoneOutputDir;
                    properties[4] = phoneOutputMaxSize;
                    properties[5] = phoneServerIp;
                    properties[6] = phoneServerPort;
                    properties[7] = "";

                    this._bootstrappedDevice = GetDevice(properties);

                    //Save to DB
                    _manager.SaveDevice(this._bootstrappedDevice);
                }



                ISensorAbstraction sensor = null;


                    //get sensor attributes for type creation
                var sensorQuery = from elem in
settings.Element("device").Element("sensors").Elements("sensor")
                                    select new
                                  {
                                      sensorNameSpace =
elem.Attribute("namespace").Value,
                                      sensorClass =
elem.Attribute("class").Value,
                                      sensorAssembly =
elem.Attribute("assembly").Value,
                                      sensorSupportedType =
elem.Attribute("supportedtype").Value,
                                      sensorBroadcastInterval =
elem.Attribute("interval").Value

                                  };
```

Figure 17: Bootstrapping device

```
   if (_manager.SensorsExist())
                    sensorsExist = true;



                    sensorQuery.ToList().ForEach((x) =>
                        {
                            //get sensor type
                            var sensorType = Type.GetType(x.sensorNameSpace + "."
+ x.sensorClass + ", " + x.sensorAssembly);

                            //Register sensor type with IoC container

TinyIoCContainer.Current.Register(typeof(ISensorAbstraction), sensorType,
x.sensorSupportedType).AsMultiInstance();

                            if(!sensorsExist)
                            {
                                object[] props = new object[3];

                                props[0] = x.sensorSupportedType;
                                props[1] = x.sensorBroadcastInterval;
                                props[2] = "";

                                //Create real sensor
                                sensor = GetSensor(props);

                                //Save to DB
                                _manager.SaveSensor(sensor);

                                this._bootstrappedDevice.Sensors.Add(sensor);
                            }

                        });

                    if (sensorsExist)
                    {
                        var properties =
_manager.ReadSensorProperties(this._bootstrappedDevice.ID.ToString());

                        properties.ToList().ForEach((p) =>
                        {
                            sensor = GetSensor(p);
                            this._bootstrappedDevice.Sensors.Add(sensor);
                        });


                    }
```

Figure 18: Bootstrapping sensors

```
   IAdapterAbstraction adapter = null;

                //get adapter attributes for type creation
                var adapterQuery = from elem in
settings.Element("device").Element("adapters").Elements("adapter")
                                   select new
                                   {
                                       adaperNameSpace =
elem.Attribute("namespace").Value,
                                       adapterClass =
elem.Attribute("class").Value,
                                       adapterAssembly =
elem.Attribute("assembly").Value,
                                       adapterSupportedType =
elem.Attribute("supportedtype").Value,};

                bool adaptersExist = false;

                if (_manager.AdaptersExist())
                    adaptersExist = true;

                    adapterQuery.ToList().ForEach((x) =>
                    {
                        //get adapter type
                        var adapterType = Type.GetType(x.adaperNameSpace + "." +
x.adapterClass + "," + x.adapterAssembly);

                        //Register adapter type with IoC container

TinyIoCContainer.Current.Register(typeof(IAdapterAbstraction), adapterType,
x.adapterSupportedType).AsMultiInstance();

                        if (!adaptersExist)
                        {
                            object[] props = new object[3];

                            props[0] = _bootstrappedDevice.ServerPort;
                            props[1] = x.adapterSupportedType;
                            props[2] = "";

                            //Create real adapter
                            adapter = GetAdapter(props);

                            //Save to DB
                            _manager.SaveAdapter(adapter);

                            //add to device
                            this._bootstrappedDevice.Adapters.Add(adapter);
                        }
return _bootstrappedDevice;}
```

Figure 19: Bootstrapping communication adapters

5.1.7 Maestroo.Core module

Maestroo.Core module is the main project assembly and contains all the domain-model logic:

- Interfaces for devices (IDeviceAbstraction), sensors (ISensorAbstraction and ISensorReadings) and communication adapters (IAdapterAbstraction) at Maestroo.Core.Interfaces namespace;
- Data layer (DataManager class) for CRUD operations and state management using SQLite internal database and local database on windows phone devices;
- Virtual sensor creation logic (VirtualSensor class);
- Messaging, Messages types definition (unknown, sensordump, sensorconfiguration, deviceinfo, clientlist, beacon, dumpreply, virtualsubscribe, virtualsubscribereply, virtualstart, virtualstartreply, virtualstop, virtualstopreply) and Message Queue classes (Inbound / Outbound) queues, serialization / deserialization logic using Google´s Protocol Buffers;
- Communications handler (AsyncSocketClient and AsyncSocketServer classes);
- Encryption and Decryption classes (Maestroo.Core.Security namespace);
- SensorReadings class for sensor concrete reading type definition;
- Enumerations class for nested supported types creation (devices, sensors, communication adapter, messages and profiles).

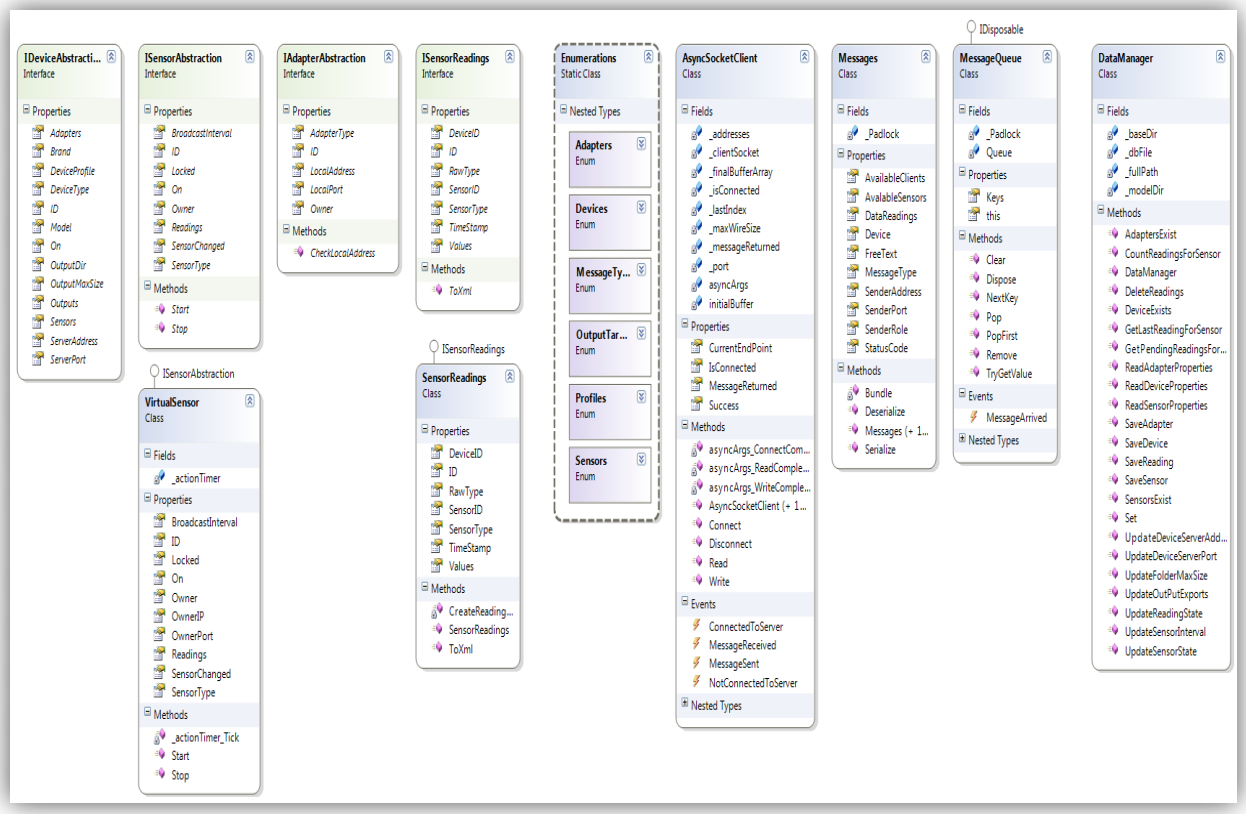


Figure 20 Maestroo.Core assembly class diagram

5.1.8 Remaining modules

All the remaining module assemblies (Omina7, Omina7.Accelerometer, Omnia7.GPS and Omnia7.Wifi, are concrete implementations of the aforementioned abstraction interfaces for a Samsung Omnia 7 Device.

The main idea is to be as dynamic as possible, meaning if someone wants to implement the same interfaces on another device it can easily do it, just by referencing the Maestroo.Core assembly and implementing the logic for the specific device brand. Then at runtime the solution will be intelligent enough to discover these implementations and run under that device profile. Figures 21 and 22 depict accelerometer implementation for a Samsung Omnia 7 device.

```
   [DataContractAttribute]
    public class Omnia7Accelerometer : ISensorAbstraction
    {
        public Guid ID { get; set; }
        public Guid Owner { get; set; }
        public TimeSpan BroadcastInterval { get; set; }
        public bool On { get; set; }
        public Enumerations.Sensors SensorType { get; set; }
        public ISensorReadings Readings { get; set; }
        public Action<ISensorReadings> SensorChanged { get; set; }
        public bool Locked { get; set; }
        Accelerometer _accelerometer;
        DispatcherTimer _actionTimer;
        public Omnia7Accelerometer(Enumerations.Sensors type, Guid owner, TimeSpan
interval, string id)
        {
            this.ID = id == "" ? Guid.NewGuid() : Guid.Parse(id);
            this.Owner = owner;
            this.SensorType = type;
            this.BroadcastInterval = interval;}

        public void Start()
        {
            this.On = true;

            this._actionTimer = new DispatcherTimer();
            this._actionTimer.Interval = this.BroadcastInterval;
            this._actionTimer.Tick += new EventHandler(_actionTimer_Tick);
            this._actionTimer.Start();
             _accelerometer = new Accelerometer();
            _accelerometer.TimeBetweenUpdates = TimeSpan.FromMilliseconds(20);
            _accelerometer.CurrentValueChanged += new
EventHandler<SensorReadingEventArgs<AccelerometerReading>>(accelerometer_CurrentValueC
hanged); _accelerometer.Start(); }
```

Figure 21: Accelerometer implementation on Omnia7.Accelerometer assembly I.

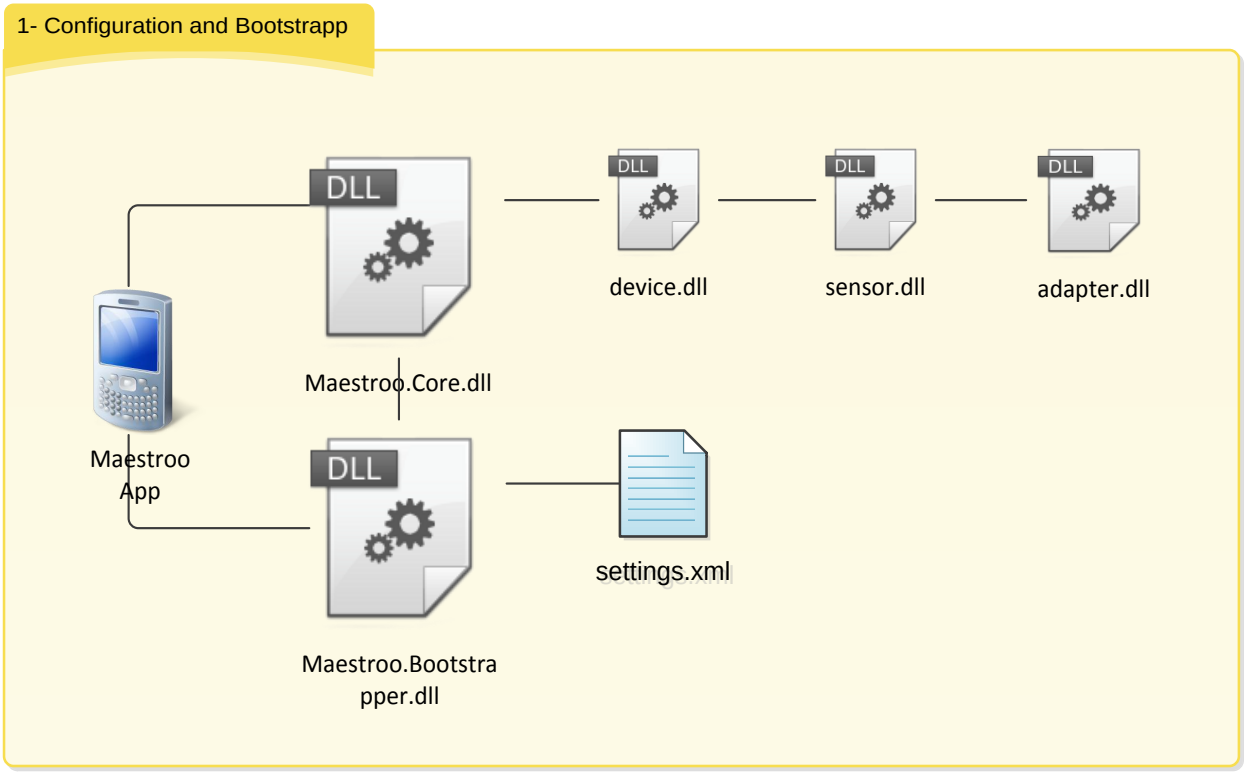
```
        void _actionTimer_Tick(object sender, EventArgs e)
        {
            if (SensorChanged != null)
            {
                SensorChanged(this.Readings);
            }
        }

        void accelerometer_CurrentValueChanged(object sender,
SensorReadingEventArgs<AccelerometerReading> e)
        {
            this.Readings = new SensorReadings();

            List<string> val = new List<string>();
            val.AddRange(new string[] {e.SensorReading.Acceleration.X.ToString(),
e.SensorReading.Acceleration.Y.ToString(),
e.SensorReading.Acceleration.Z.ToString()});

            this.Readings.Values = val;
            this.Readings.DeviceID = this.Owner;
            this.Readings.SensorID = this.ID;
            this.Readings.SensorType = this.SensorType;
            this.Readings.TimeStamp = e.SensorReading.Timestamp.ToString();
            this.Readings.RawType = "float";


        }

        public void Stop()
        {
            this.On = false;
            _accelerometer.Stop();
            _actionTimer.Stop();
            this._actionTimer.Tick -= new EventHandler(_actionTimer_Tick);
        }

    }
        }
```

Figure 22: Accelerometer implementation on Omnia7.Accelerometer assembly II.

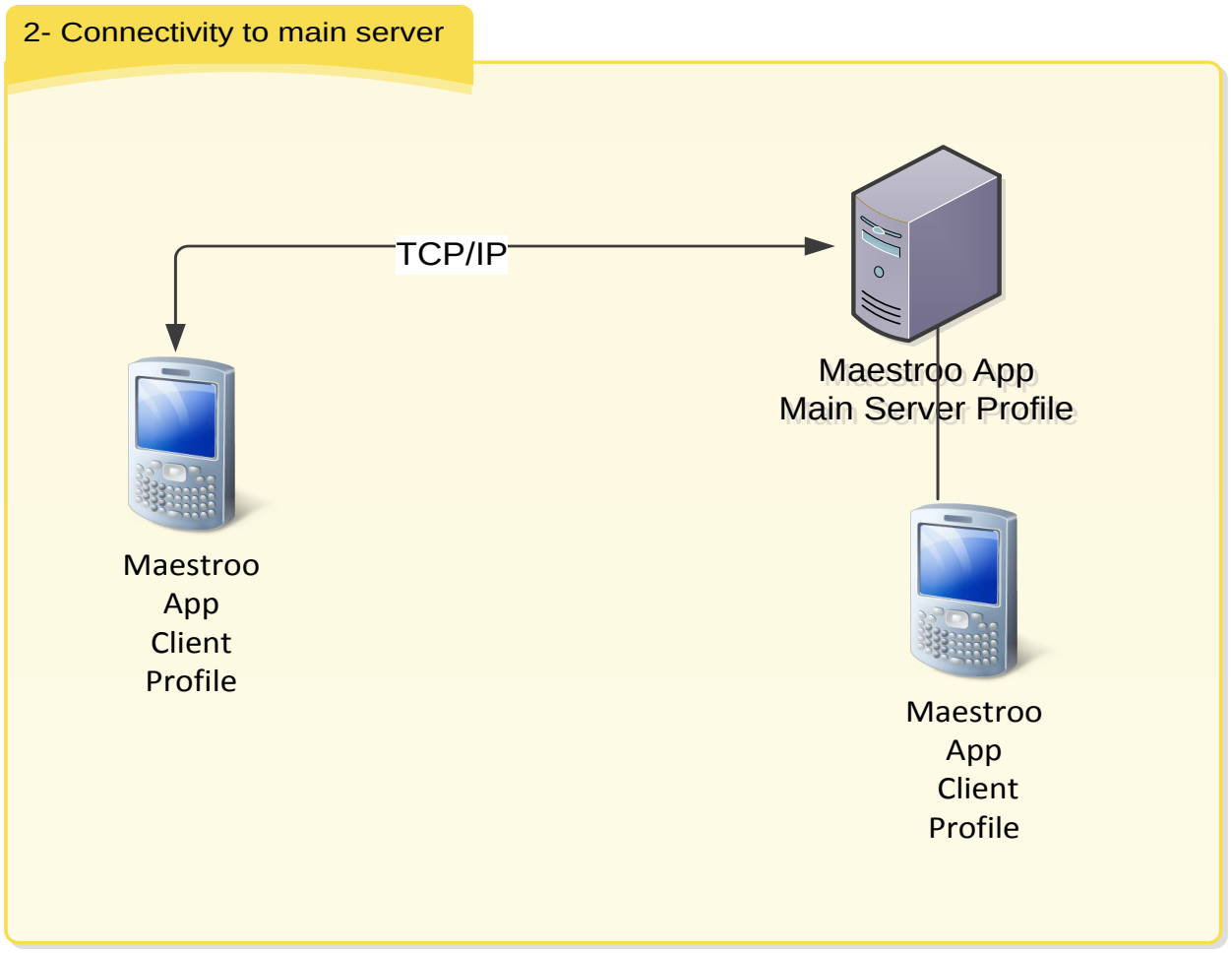
5.2 Functional flows

This section presents Maestroo´s functional flows and tries to explain in a legend what is happening on each and every one of them. These action flows will be categorized into the following:

- Configuration and Bootstrap;
- Connectivity (server);
- Actions (online and offline sensing, and sharing).

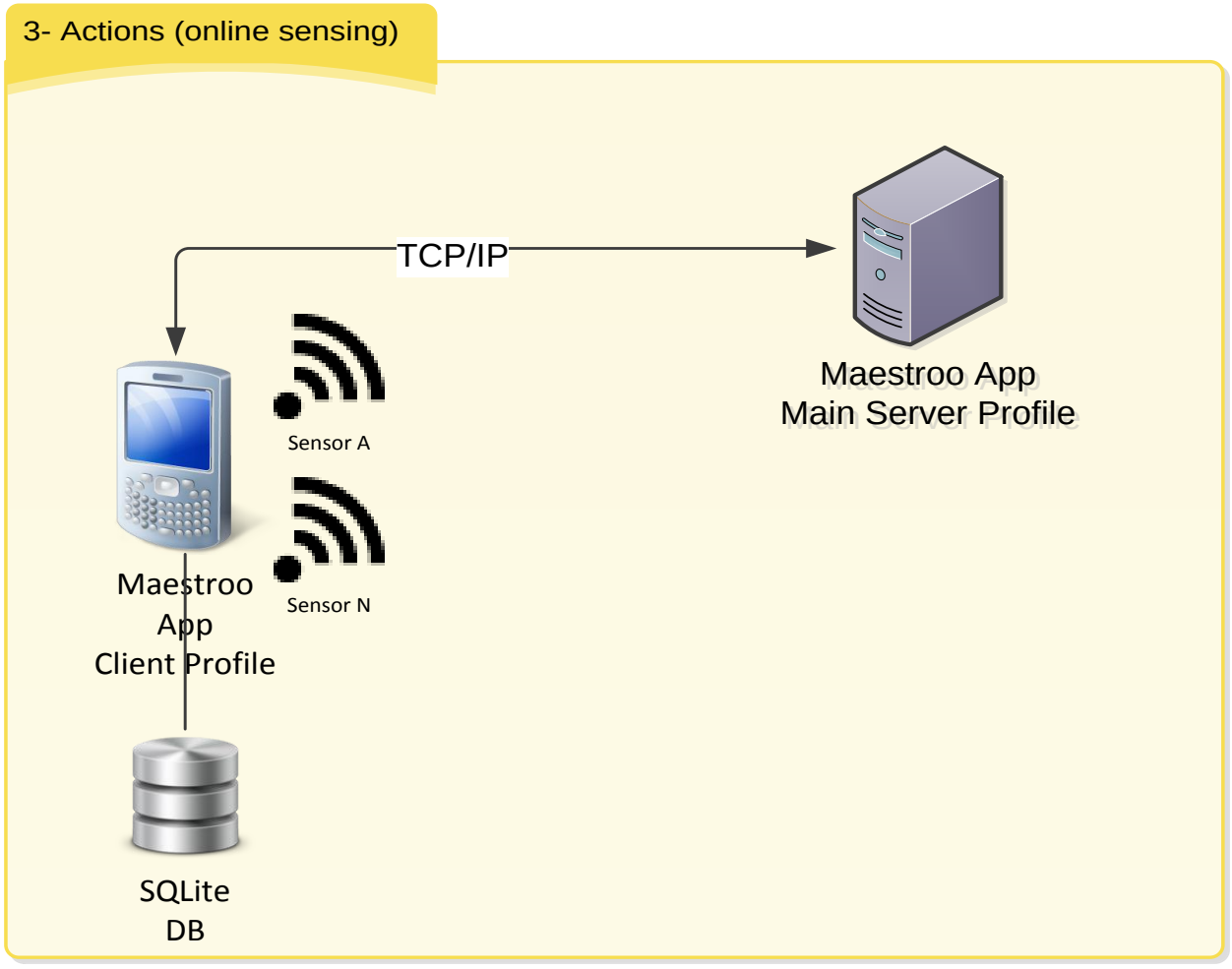


1: On booting up Maestroo middleware application uses the Maestroo.Bootstrapper.dll to search on the file system for an xml settings file that must contains all the references to the assemblies and their types that implement all the needed Maestroo.Core.dll interfaces, and a main server IP address and port in order to create a device at runtime. These assemblies are custom developed for specific device types and for creating profiles on those device types, e.g. A device with 1 or n sensors (accelerometer, gps, temperature, etc) available and 1 or n communication interfaces (Wifi, Bluetooth, Zigby, etc).

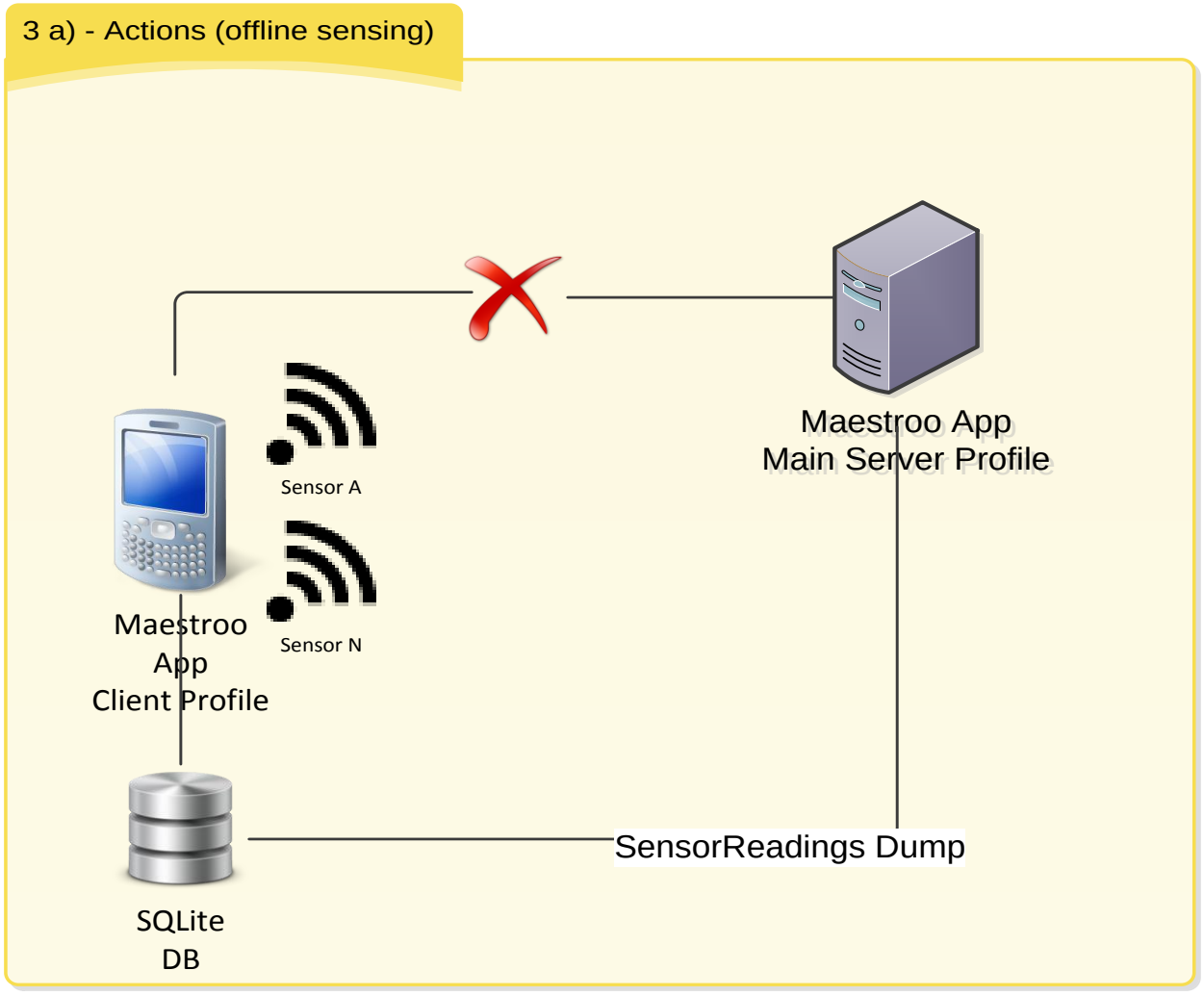


2: After the hardware bootstrapping is done the device will try to find the main server on the network via TCP/IP by sending a beacon type message to the main server address that was specified on the settings.xml file. We shouldn't confuse beacon message with 802.11wireless protocol beacons, this is instead a custom message type defined in the maestro.core assembly module, used internally to register all devices on the main server clients list.

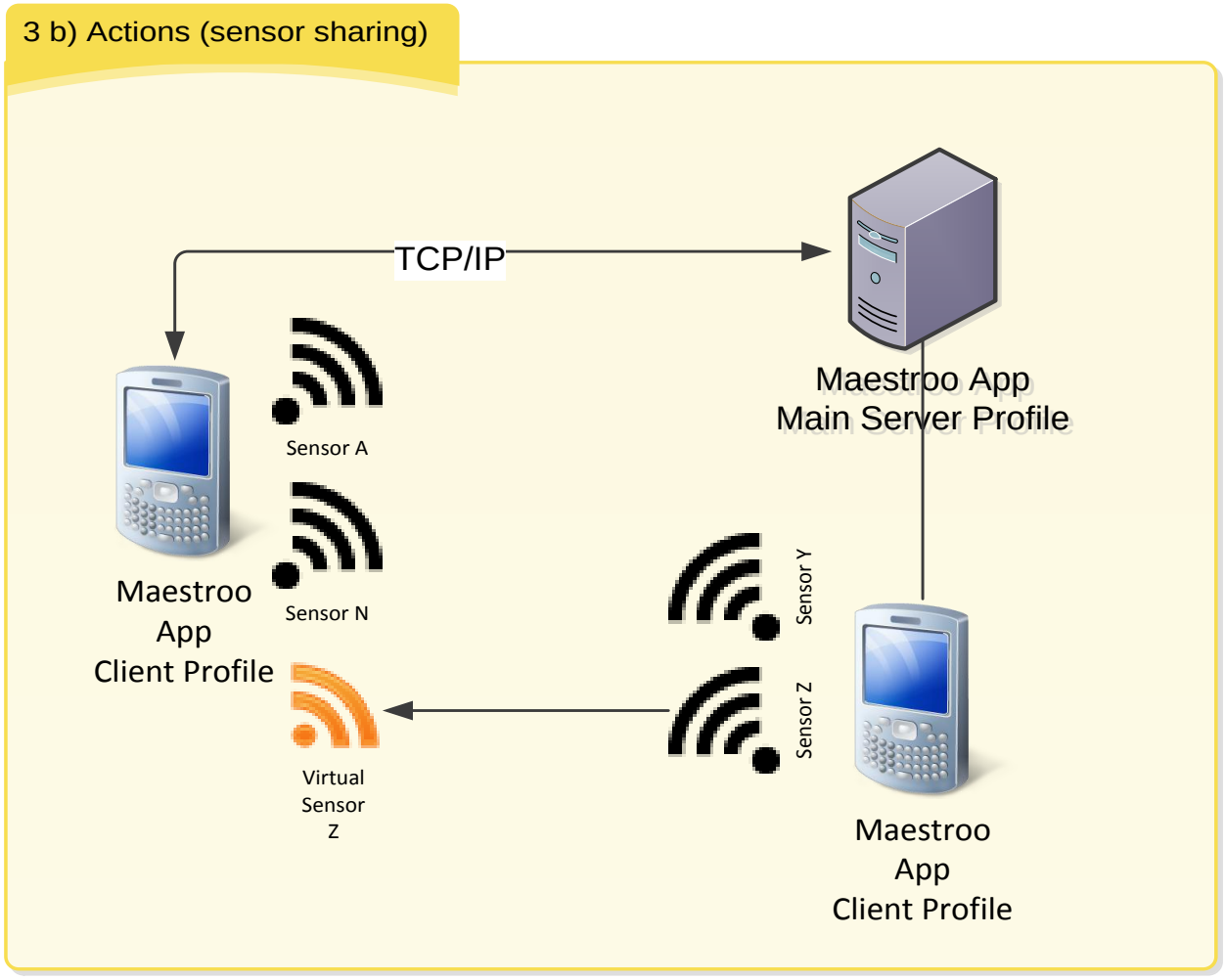
When the server replies back, the client device immediately asks for all the addresses of known network peer nodes and the sensors they have to share and saves that list in internal memory.



3: If connectivity to the main server exists a client can send periodic (defined in the property broadcast interval on the sensor settings menu) sensor readings via TCP/IP to the main server as a serialized SensorReadings message object. The server upon receiving this type of message de-serializes it and handles its persistence to the file system as an xml file. Before sending the readings, the middleware will save the readings object into the internal SQLite Database with a 'pending' state until it receives a '200 OK' message from the server and updates the readings state to 'received' state.



3a) State management is handled by saving to local SQLite Database all sensor readings before sending them to a backend main server. If connectivity to the main server fails all readings that are in a 'pending' state are later (when connectivity is restored) sent to server via a dump command, when the main server address and the export type = 'socket' flag is defined in the xml settings configuration file.



3b) Sensor sharing is made possible when the clients receive the connected peer information list from the main server. Afterwards, when selecting the option to add a new virtual sensor the application will show all available sensors in the network. When this operation is done there is a coupling of a new sensor onto the 'borrower' device. This sharings lasts until one of three events occur, the borrowing devices stops using the sensor or the lending device starts using the sensor or exits the network. What is interesting about this sensor virtualization concept is that as Maestroo abstracts what a sensor really is, by just defining an interface contract and letting clients implement the functionality at their own will, a sensor can be really anything ranging from an output of any software, to a feed reader, to camera images, to an audio stream, etc.

5.2.1 Testing environment

The lab tests that were performed in order to check the functioning and the consistency of the middleware solution and comprised in creating a three device node network, consisting of a real Samsung Omnia 7 Windows phone device (node 1), an Android Emulator running the latest 4.0.3 OS version (node 2) and a Windows 7 workstation (node 3).

Node 1 is configured with a 'client' profile with 2 sensors, 1 accelerometer and 1 GPS, and 1 communication interface, Wi-Fi. A client profile means that communications are held via a duplex socket that remains opened between the 2 endpoints until all connections are closed.

Node 2 has is configured with an 'intermediate' profile with 2 mock sensors, a temperature and a gyroscope and a Wi-Fi communication interface also. Intermediate profiles mean that these devices can also function as servers (have listening sockets).

Node 3 is configured with a 'server' profile; it has no sensors and acts as a backend server for data storage and inference on the sensed data thought an HTLM5 single page application.

The tested action flow was the following:

- Node 1, 2 and 3 boot up;
- Node 1 and 2 register themselves on node 3;
- Node 1 starts sending GPS and accelerometer raw data to node 3;
- Node 3 goes offline and communications is interrupted;
- Node 1 starts to store data locally;
- Node 3 goes online and communication is restored;
- Node 1 sends offline data via a dump to server;
- Node 1 continues sending sensor data to server;
- Node 2 sends mock gyroscope data to server;
- Node 1 queries Node 3 for other network nodes addresses;
- Node 3 responds with Node 2 IP address;
- Node 1 contacts Node 2 directly and queries it for its available sensors;
- Node 2 replies with temperature sensor;
- Node 1 creates a virtual sensor by 'borrowing' Node 2 temperature sensor;
- Node 1 starts using temperature sensors and sends data to server;
- Node 1 stops using temperature sensor;
- Node 2 starts using temperature sensor and beffers data to server;
- Sensed data is visualized in a HTML5 application that moves an object based on accelerometer readings from Node 1 and displays the temperature based on temperature readings from Node 2;

Results proved that for a three node network all system operations (the flows described above) perform in a very stable manner if the broadcast interval sensor property is not below a 7 milliseconds threshold (which by itself it is already a very small and unrealistic value) otherwise network flooding occurs and the system might eventually crash. Boot loading times were less

than 5 seconds on real devices, with the exception of the android emulator that can take up to a minute. Sensor dumps also didn´t caused any network flooding or server crashes.

5.2.2 Problems encountered

On the process of coding Maestroo there were several problems encountered, some that had an immediate solution, and others that had to be postponed for a future analysis.

The major difficulty of all was to deal with each device´s API restrictions. The initial idea, before coding was a little naïve in a sense that the belief was that if the language is transversal everything could be implemented in the same manner, but that is a totally wrong idea.

To start with, each device has its own API and each one of those API´s exposes or hides device functionalities more preeminently than others. Windows Phone SDK is very a complete and easy to use set of libraries but it has a major drawback, it is very restricted in terms of communication interfaces, e.g. there is no way of creating listening sockets on these devices, as well as there is no way to use the Bluetooth interface, as well as other simple things as getting the device IP on the network without some workaround solutions. These practical difficulties were overcome by creating several profiles to the middleware users. A client profile, a server profile and an intermediate profile that can act as a client or a server. In a client profile communications are held via a duplex socket that remains opened between two endpoints (client and main-server) until all communications are closed. Server profiles have listening sockets, and intermediate profiles have listening and client sockets.

Other major drawbacks were very slow debugging times (that were getting better as version updates appeared) on the Mono for Android framework when running on the emulator. Also, the Android SDK´s emulator has its own virtual network address and it´s impossible to connect with it via TCP/IP in other way than the host´s loopback address, so it was necessary to develop a proxy on the host that listens on its network IP and forwards packets to the loopback for the emulator to receive those packets.

Finally as it was already stated the initial idea of porting the code to Netduino, was something we weren´t able to solve at the time of this report, because it requires to implement new logic on some of the most important software features like the use of generics, reflection and state persistence with SQLite Database.

5.3 Summary

In this chapter we analyzed and compared features on the middleware that exist on the most expressive USP and beyond. We used this feature comparison as a start off base for the building of our own middleware, Maestroo.

Maestroo was overviewed, architecture and functional-wise as also the problems encountered during the development stages and we truly believe that our approach, inspired by the strengths of the existing ones and trying to surpass their known weaknesses will provide an alternative scenario, but above all serve as an efficient cross-platform initiative for more sensorial sharing applications to emerge.

6. Final observations

6.1 Conclusions

Pervasive computing and USP are more than intermingled into reality today and this tendency tend to increase as more and more sensors are embedded into everyday devices and these become more and more powerful in terms of processing power. As this happens and after analyzing how the most expressive USP are organized in terms of middleware options used, this initiative of developing Maestroo as a device embedded middleware will bring a new light onto the existing USP because of its highly distributed and versatile nature.

As the end results from out lab experiments proved it is possible to put three totally different devices with distinct OS´s on a USP network, sharing sensor data and sensor hardware in a straightforward way. Because of this characteristic it can be easily adopted and used to form global scale USP, but also, small scale ones as it enables to define different profiles for each device, as seen on our lab, and thus permitting inter-device direct connectivity to form a collaborative sensing network.

By enabling sensor sharing in a very abstract way (by using sensor virtualization), a new wave of possibilities emerge as literally everything can be treated as a sensor, enabling developers to bring innovative ideas into the USP world and bring USP into a more broad range, not as academic research fields as the majority of them are. In the lab experiment moving an HTML5 object based on the data that was broadcasted to the server was a very simple thing to do.

This is still a work in progress and it is possible that some of the concepts, ideas and technical issues we wrote about can suffer changes, but what is important to retain is that this initiative represent an excellent opportunity to bring sensing tasks to the forefront and help mobile users in their daily urban lives.

6.2 Future work

As it was already stated at this stage the solution is only deployed outside the emulated environments on Windows Phone devices. It is out intentions to deploy the code very soon from the Android emulator to an Android device and also to an iPhone, after the two first environments are stabilized. Also porting the code to Netduino boards is a challenge we want to achieve, but it is necessary a lot of code refactorings and using another approaches can quickly lead us to develop something that is radically different at that is something that it has to be avoided.

Although there is still a lot of work ahead we can conclude that Maestroo can definitely be a starting point for the widespread of USP at the granularity of almost the individual level.

References

[1] A. Kansal, F.Zhao, “Location and Mobility in a Sensor Network of Mobile Phones” – Microsoft Research.

[2] MetroSense 2007. MetroSense Project. http://metrosense.cs.dartmouth.edu

[3] M. Weiser, “The Computer for the 21st Century,” *Scientific Am.*, Sept., 1991, pp. 94-104; reprinted in *IEEE Pervasive Computing*, Jan.-Mar. 2002, pp. 19-25.

[4] Rajive Bagrodia, Wesley W. Chu, Leonard Kleinrock, and Gerald Popek. “Vision, Issues, and Architecture for Nomadic Computing” – 1997.

[5] IBM Research. Mobile & Pervasive Computing – 2002.

[6] K Ducatel, M Bogdanowicz, F Scapolo, J Leijten, J-C Burgelman – “ISTAG Scenarios for Ambient Intelligence” – 2010.

[7] Stefan Arbanowski. “I-centric Communications” – Berlin 2003

[8] A. Campbell, N. Lane, E. Miluzzo, R. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, S. Eisenman and G. Ahn, “The Rise of People-Centric Sensing”; reprinted in *IEEE Computer Society, Jul –Aug. 2008, pp. 12- 21.*

[9] N. Lane, S- Eisenman, M. Musolesi, E. Miluzzo amd A. Campbell, “Urban Sensing Systems: Opportunistic or Participatory?” – Nov 2007

[10] SensorLab. http://sensorlab.cs.dartmouth.edu/

[11] BikeNet. http://metrosense.cs.dartmouth.edu/projects.html

[12] SkiScape. http://metrosense.cs.dartmouth.edu/projects.html

[13] SoundSense. http://metrosense.cs.dartmouth.edu/projects.html

[14] CenceMe. http://metrosense.cs.dartmouth.edu/projects.html

[15] Funf: Open Sensing Framework. http://funf.media.mit.edu/

[16] Massachusetts Institute of Technology. http://web.mit.edu/

[17] Behavio, inc. http://behav.io/

[18] MIT Human Dynamics Group. http://hd.media.mit.edu/

[19] Funf journal. https://play.google.com/store/apps/details?id=edu.mit.media.funf.journal

[20] Funf in a box. http://funf.media.mit.edu/inabox

[21] Microsoft Research. http://research.microsoft.com/en-us/default.aspx

[22] SenseWeb. http://research.microsoft.com/en-us/projects/senseweb/

[23] SensorMap. http://www.sensormap.org/sensewebv3/sensormap/

[24] Cosm. https://cosm.com

[25] LogMeIn. https://secure.logmein.com/

[28] SensorPedia. http://sensorpedia.com/

[29] Oak Ridge National Laborartory. http://ornl.com/

[30] SensorPedia API. https://sites.google.com/site/sensorpedia/Home/sensorpedia-api

[31] Open Geospacial Consortium. http://www.opengeospatial.org/

[32] Resseguie and Fairgrieve, "PULSENet: An implementation of Sensor Web standards", reprinted in *IEEE Computer Society, 2010.*

[33] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: "A Tiny Aggregation Service for Ad-Hoc Sensor Networks." In *OSDI 2002*, Boston, USA, December 2002.

[34] C. C. Shen, C. Srisathapornphat, and C. Jaikaeo. "Sensor Information Networking Architecture and Applications." *IEEE Personal Communications*, 8(4):52–59, 2001.

[35] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: "A Tiny Aggregation Service for Ad-Hoc Sensor Networks." In *OSDI 2002*, Boston, USA, December 2002.

[36] A. Boulis, C.C. Han, and M. B. Srivastava. "Design and Implementation of a Framework for Programmable and Efficient Sensor Networks." In *MobiSys 2003*, San Franscisco,USA, May 2003.

[37] P. Levis and D. Culler. Mate: "A Tiny Virtual Machine for Sensor Networks." In *ASPLOS X*, San Jose, USA, October 2002.

[38] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley,1994.

[39] S. Li, S. H. Son, and J. A. Stankovic. "Event Detection Services Using Data Service Middleware in Distributed Sensor Networks." In IPSN 2003, Palo Alto, USA, April 2003.

[40] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. "On the Need for System-Level Support for Ad Hoc and Sensor Networks." *Operating System Review*, 36(2):1–5, 2002.

[41] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy,L. Luo, S. Son, J. A. Stankovic, R. Stoleru, and
A. Wood. EnviroTrack: "Towards an Environmental Computing Paradigm for Distributed Sensor Networks." In *ICDCS 2004*, Tokyo, Japan, March 2004.

[42] Dropbox – http://www.dropbox.com

[43] TinyIoC– https://github.com/grumpydev/TinyIoC

[44] Vici.CoolStorage– http://viciproject.com/wiki/projects/coolstorage/home

[45] Protocol Buffers– http://code.google.com/p/protobuf-net/