INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

PhoneSensing

Smartphone-based Application for Distributed Sensing in Human Networks

Nadir Amaral Türkman

Dissertation submitted to obtain the Master Degree in
Communication Networks Engineering

May 2012

Acknowledgments

I would like to thank my parents for their support during my years as a student. They were fundamental for my growth as a student with the values they passed on to me. They are main reason for my success as a professional and as a human being. I'm very grateful to have such amazing people as my parents and tutors in life, and so, i dedicate this work to them as a token of my appreciation.
I would also like to share my appreciation to my dissertation supervisor Prof. Rui Rocha for his support, knowledge, patience and above all guidance. Without it, it wouldn't have been possible to finish the dissertation in time.
Last but not least i would like to thank all my close friends for their support during the whole process. Their understanding, moral support and guidance were invaluable.

Abstract

The evolution and proliferation of modern smart-phones has opened a door to a new type of Wireless Sensor Networks (WSN). While traditional WSNs consist of fixed sensor nodes, smart-phone based ones have a large mobility factor.

Modern smart-phones also possess a respectable amount of sensors which allied with their mobility and processing power are able to produce a large-scale Opportunistic Sensor Network (OSN). The sensors also provide developers with the possibility to create applications that improve lifestyle by collecting useful contextual information from the user's lifestyle. We propose to explore both the opportunistic and social potential by creating an application that uses an underlying OSN to improve the user's day to day personal and social experience.

To explore the opportunistic potential we propose a decentralized resource sharing protocol to apply to the underlying OSN. It will provide the possibility for mobile nodes to exchange resources and supply the necessary framework for the social sensing application.

Keywords

Smart-phone, Sensor, Sharing, P2P, Sensing, Bluetooth, PhoneSensing

Resumo

A evolução e difusão do uso de telemóveis inteligentes (*smart-phones*) permitiram o desenvolvimento de um novo tipo de redes de sensores sem fios. Tradicionalmente estas são compostas por nós fixos, ao invés das constituídas por smart-phones que oferecem o factor mobilidade.
A quantidade considerável de sensores dos *smart-phones*), aliada à sua mobilidade e poder de processamento, potenciam o desenvolvimento de redes de sensores oportunistas em grande escala. Os sensores também possibilitam a criação de aplicações, que melhorem a qualidade de vida do utilizador, através da recolha de informação contextual.
Este trabalho propõe a criação de uma aplicação e de um protocolo descentralizado de partilha de recursos, que explore as vertentes oportunista e social, através da rede de sensores subjacente.

Palavras Chave

Smart-phone, Sensor, Partilha, P2P, Contextualização, Bluetooth, PhoneSensing

Contents

List of Figures

List of Tables

List of Acronyms

WSN	Wireless Sensor Networks
PDA	Personal Digital Assistant
OSN	Opportunistic Sensor Network
SAP	Sensor Access Point
MS	Mobile Sensor
MB-SJF	Mobility Based Shortest Job First
TCP	Transmission Control Protocol
ER	Entity-relationship
SDK	Software Development Kit
RIM	Radio Interface Module
SM	Sensor Module
SRMM	Sensor Request Manager Module
ACM	Application Core Module
FIFO	First In First Out
SAR	Single Active Request
MAR	Multiple Active Request
SRO	Sensor Request Only
CREQM	Capability Request Message
CRESM	Capability Response Message

SREQM Sensor Request Message

SRESM Sensor Response Message

DTN Delay-tolerant networking

SPL Sound Pressure Level

TTL Time To Live

1 Introduction

Contents

WSN consist of sparsely distributed autonomous sensors that in cooperation monitor the surroundings they're adjacent to. WSNs have been used on a myriad of scientific fields on all sorts of applications but have struggled with inherent limitations commonly associated with the lack of resources that characterize the majority of sensor nodes. Such limitations exist due to the fact that nodes that compose these networks are usually resource limited devices, with low autonomy and also low power and low bandwidth radio interfaces [1]. These limitations prevent large-scale deployments of the aforementioned networks and as such they are very useful in research contexts on closed-door laboratories and small scale experimentations. When ported to real case scenarios that require large deployments they become expensive and very high maintenance costs but it doesn't mean that they have not been used and that they haven't been extremely useful. They have been crucial on all sorts of fields ranging from domestic to professional and or industrial use.

Some examples include volcano activity monitoring [2] [3], where the deployment of vibration sensors, acoustic sensors, temperature sensors and gas emission sensors can aid scientists gather data that helps them in understanding even further the intrinsic details of what triggers such powerful natural events. The have also been used on private enterprise deployments [4], where they are used for example, in large scale monitoring of gas and/or oil pipelines. On both examples, besides the inherent cost of such complex deployments, there is always the maintenance issue would such network fail due to node malfunction somewhere in the structure.

Modern wireless telecommunications started off with big cell phones that had just one simple function, and that was to make and receive calls. Later the possibility to send and receive text messages was added. As the time went on, phones morphed into Personal Digital Assistant (PDA) that incorporated more memory and better CPUs. This yielded a panoply of new applications that improved their usability. Since evolution never stops, eventually smart-phones appeared, and today's smart-phones are a long way from the behemoths that their ancestors where.

One of the particularities of modern smart-phones is the inclusion of an array of sensors that were at first intended exclusively for intended purposes so that some functionalities can be made possible [5]. A proximity sensor exists so that modern touch screens don't get activated when the user puts the phone close to the face while making a call. Orientation sensors and accelerometers are used to rotate the phone's screen every time the user rotates the phone itself but ended up also being used on other applications such as games. Light sensors we're added so that the phone could smartly manage battery consumption by deeming the lights when they were not necessary. Magnetometers and digital compasses were also throw into the mix. The camera has been a part of modern phones for a while now, and the microphone has been on phones since their birth.

With the inclusion of these sensors, smart-phones have become powerful mobile devices and it has not gone unnoticed. There has already been quite a bit of researching into the possibility of using these devices to deploy large scale WSNs [6] [7]. Since smart-phones are managed by their users they're usually kept charged and on for the majority of the day and besides packing quite a punch performance wise there are more than 400 million[1] of them worldwide.

Traditional WSNs are thought out before being deployed and their structures are a byproduct of the project's motivations. In contrast, a WSN composed of smart-phones has no predefined structure, it's a sort of mutating, anarchic network where humans dictate how it morphs by moving about with their smart-phones. Smart-phone based WSNs have a large mobility factor when compared with the traditional WSNs composed by fixed nodes. Most of the work done on this field, but not all [8] [9], has been directed to the human component, mostly the social aspect of it. Social networks have been a growing industry and there is already been a modicum amount of research into the possibility of enrichment of the social networking experience [10] [11] [12] by introducing the capacities of modern smart-phones and WSN composed by them. This is commonly referred to as social sensing.

1.1 Motivation and Goals

What we have here is, in theory, the largest WSN waiting to be explored like any other WSN. In [13] this is called People-Centric Sensing since sensors are not deployed in a traditional form, rather than on trees or buildings, humans become the focus area of sensing. Consequently applications thought out for this network will also be aimed at the human kind with the goal of improving life quality on a day to day basis by providing useful information that would otherwise go unnoticed.

For the information to be useful it has to be correctly collected and interpreted. Having a vast array of sensors, modern smart-phones endorse such a possibility. Context and actions can be inferred using the available sensors, either using a single sensor or cross-referencing data from various sensors. If smart-phones performed these contextual conjectures on their own they would have to depend only on their learning process. The learning process can be done through the user's input or resorting to automated processes. When included in an WSN in the presence of other similar devices, there is a possibility for collaborative inference, where cooperative actions by two or more neighbouring phones can yield better results.

[1] http://gigaom.com/2010/03/24/mobile-milestone-data-surpasses-voice-traffic/?utm_source=gigaom&utm_medium=navigation

One other aspect that should also be mentioned is opportunistic sensing. Such concept is already very useful on typical WSN and can also be used on the context of people centric sensor networks. In [13] opportunistic sensing is regarded as the act of pervasively gathering information, that is, the phone will sense the environment without any necessary input from the user. For the remainder of this document, we will regard opportunistic sensing as defined in [14], where it defines opportunistic sensing has the act of tasking an application request for a given sensor in a remote device within a preferred time frame. This translates to the possibility of a less capable device tasking another device, that has the desired sensor, in the surrounding area to collect data from a sensor that would otherwise be unavailable.

This project proposes a solution that explores the aforementioned possibilities of a smart-phone-based sensor network. This shall be done by creating a framework that's able to process and evaluate sensor data in order for it to be possible to infer contextual information and use it to improve lifestyle. It is also intended for the applications to exploit opportunistic connections with other passing devices. This should help improve sensing accuracy and in some cases, more than improve since these opportunistic connections might lead to resource sharing. A solution that overcomes inherent limitations of less capable devices.

The expected goals are: The development of a framework that contains a service that is able to communicate with the device's sensors, collect data and interpret it with an acceptable accuracy. A communication protocol and network solution that yields the possibility of resource sharing through the use of opportunistic connections.

1.2 Challenges

The stipulated goals have a few challenges attached that need to be solved, and part of this document will focus on solutions for those very same challenges. In order to specify what they are it is best to divide them in two groups, the sensing challenges and the resource/network challenges. The sensing challenges (sample and sensing context) refer to the challenges raised by the need to sense and interpret the surroundings. The resource/network challenges (resource availability and mobility) are associated with resource availability and opportunistic sensing.

1.2.1 Sampling Context

Is the set of conditions required for sampling to take place, including location and orientation. This will be a challenge since it can be very hard to know what are exactly the conditions in which sensing is possible, that is, if high-fidelity is to be reached.

1.2.2 Sensing Context

Is the metadata that describes the conditions to which the sensors are exposed to. These conditions affect the sensor's capability to perform the sensing operation and hence it affects the collected data. The main challenge is to accurately sense the device's context so that it can compare it to the sampling context and decide whether the conditions are inside the required parameters.

1.2.3 Resources

In this document it was mentioned that modern smart-phones come equiped with an array of sensors, but its also true that not all smart-phones have a comprehensive list of sensors available. Some have more than others and as such the framework needs to able to cope with situations in which a certain device does not have the required hardware to perform a given task. For example, let us consider a device that has no GPS but needs to acquire its current position, it won't have the required sensor to gather the necessary data. There has to be a way overcome this limitation.

1.2.4 Mobility

A particularity of the architecture in hand is that nodes are not still, and this makes it harder to sense properly. Mobility can quickly change the sensing context. Hence, the sampling context might not be acceptable for the necessary time frame in order to collect the required data. Node mobility can also affect the opportunistic angle. Since a node that has been tasked might not stay in range enough time to process and send a reply for a given request or might not reach the targeted region inside the preferred time frame (tasking prediction problem). Even having the possibility of requesting a sensor from another device might not be enough, since the available pool of mobile sensors is limitied by the uncontrolled mobility of devices. As such it might not contain the necessary sensor (tasking availability problem).

1.3 Document Structure

This dissertation is organized into six main chapters. The following chapter, chapter 2, provides an overview of related work regarding opportunistic and social sensing. Chapter 3 presents a detailed view of the proposed architecture. Chapter 4 goes over the solution's implementation details. Chapter 5 analyzes the test results and the solution's validation process. Chapter 6 sums up the project and proposes future work on the implemented solution.

2

State of the art

Contents

This chapter will focus on the existing related work referring to opportunistic sensing and social sensing applications. In order for opportunistic connections to be possible the underlying WSN has to be able to supply the devices with the possibility for it to happen. A WSN that provides said possibility is referred to as an OSN. Social sensing applications explore the smart-phone's capabilities of contextual conjecture in order to enrich the online social networking experience.

2.1 Opportunistic Sensing

When developing a network solution for an OSN a few considerations have to be made, one if which is the architecture, and as such this section of the document will focus on a couple of alternatives on which to base a network solution for the OSN. Besides focusing on possible architectures, we will also focus on different techniques used on opportunistic sensing. Both techniques and architectures help circumvent the challenges associated with this sort of the networks.

2.1.1 Techniques

In order to improve opportunistic sensing, especially to meager the effects of the tasking availability problem, there has to be a compromise between what is expected and what is achieved. In the case of the tasking availability problem, the relaxation of what is considered an appropriate sensor, is one possible solution. In [14] two techniques are proposed, sensor substitution and sensor sharing. Sensor substitution has a more literal approach to the relaxation concept and sensor sharing approaches the problem in a more traditional fashion.

2.1.1.A Sensor Sharing

The main idea behind sensor sharing is quite basic. In essence what happens is that, if a mobile device A requires a sensor α to process a determined application task but it does not have available the required resources, be it sensor α or even computational power to perform such a task, it can conscript another device in the vicinity to share the sensed data from a sensor α that it self possesses. In [14] two scenarios for sensor sharing are proposed: (i) mobile device A requires the data from a target zone but doesn't have the required sensor α so it request a mobile device B that possesses a sensor of the same type and it is located in the target region to gather the required data (ii) is a similar situation but the mobile device B isn't at the target zone but rather moving towards that target zone, so mobile device A tasks B to gather the required data. In both scenarios there's also the tasking prediction problem, but it's more present in the second scenario because besides having to predict if the device will stay in range to relay back the results, it also has to predict if the device will move onto to the target region and if it'll stay in that same region for the necessary time to sense the requested data. [15] proposes a solution for the tasking prediction problem in scenarios similar to (ii).

As far a sensor sharing goes there are three different sensor sharing techniques proposed in [16] and they are direct sharing, delayed sharing and authorized sharing.

2.1.1.B Direct Sharing

Direct sharing is when one mobile device node treats another's sensor(s) as its own in real time. For this to happen there has to be sharing initialization between both mobile sensor nodes and as such the communication as to be decentralized, not relying on any back-end infrastructures. Data transfer in direct sharing is done directly between the participating nodes.

2.1.1.C Delayed Sharing

Delayed sharing is aimed at shorter rendezvous between mobile sensor nodes where direct sharing is not possible. With delayed sharing an association between both participating nodes is created via a back-end device. Data transfer, when in delayed sharing, isn't done directly by said nodes, instead, via the backend, for example, using a cookie that indentifies the sharing association.

2.1.1.D Authorized Sharing

Authorized sharing exists when devices keep on a backend previous sensed data. On such occasions it's possible for a determined device to share a cookie that would permit the recipient to access the backend and use any saved data that was previously sensed. Authorized sharing is a generalization of delayed sharing.

2.1.1.E Sensor Substitution

Again in [14] two scenarios are proposed, in this case for sensor substitution, regarding a similar situation as (i) described in the sensor sharing section. In this situation a mobile device C requires sensed data from a target region from a sensor β but instead of tasking another device that contains the sensor β it relies on sensor substitution as an alternative to acquire equivalent or similar data. Sensor substitution can be direct or indirect.

In direct sensor substitution the mobile device C instead of using that sensor β that he doesn't have available it will rely on a sensor σ to collect the equivalent data that would be given by β. In this case the equivalence might not meet expected accuracy and precision requirements. For example, if a mobile device was to measure a slope of a section of a road using a three-axis accelerometer but didn't have such a sensor it could rely on a direct sensor substitution by using a GPS receiver instead. This could be done by using periodic altitude measurements and then extrapolating from them the road's slope. Even though it does provide the requested data it has to be noted that in this example the calculated slope value won't be as accurate with the GPS as it would be with the three-axis accelerometer because of the GPS receivers lacks accuracy when measuring altitude.

The indirect sensor substitution approach does not deviate much from the direct approach. The indirect approach instead of using just one sensor as a substitute to the required sensor β it uses more than one sensor and with the help on inference techniques they generate the desired data. For example, if a mobile device required latitude and longitude measurements from the GPS receiver but did not have it available it could use a three-axis accelerometer and a three-axis magnetometer to infer the GPS coordinates provided that it had the initial coordinates [17]. This can be done because the magnetometer would provide the direction in which the device was traveling and the distance could be calculated with the double integral of the acceleration.

2.1.2 Resource Description

Before employing the above techniques it is important that the devices exchanging the sensors speak the same language. This is paramount to guarantee that the requests are understood by there recipients. In [26] the authors propose a Resource Description Language (RDL) that provides the possibility to describe a wide range of resources in a standardized format. They also provide a Java based framework.

It fits very well into the sensor sharing subject because RDL gives attention to the description of their characteristics for interoperability, sharing and discovery. The RDL in [26] can be specified in two different formats, XML and KLV. The KLV format yields very low sizes which in a mobile environment with short *rendez-vous* and low bandwidth connections can be very useful.

2.2 Architectures

In addition to the aforementioned techniques it is crucial that before the design of an OSN a proper architecture is chosen. The focused architectures will be the centralized and decentralized. Both architectures look to lessen the tasking availability and prediction problems. These two architectures have different requirements, pros and cons and choosing from one and other will depend on the goals of the intended OSN.

2.2.1 Centralized

A centralized OSN implies that device's that want to send a tasking request have to do it through a central node that controls the task scheduling to neighboring devices. These central nodes are called Sensor Access Point (SAP).Centralized OSNs might or might not have a centralized back-end server that also processes scheduled tasks. Even though a centralized approach is a solution for the already mentioned challenges, it does introduce it's own challenges. We selected Halo [18] and BikeNet [19] as two representative examples of centralized architectures. In [18] a centralized approach with the use of SAPs is proposed and in [19] a combination of SAP, mobile SAP and back-end servers is used.

2.2.1.A Halo

In a centralized OSN that relies only on SAPs the main challenges are the action radius of the SAP. The action radius has to be dynamic in order to increase the number of Mobile Sensors (MSs) available to task, reduce tasking delay, increase the amount and utility of sensed data to delay-aware applications, reduce collection delay, and reduce the likelihood of mobile sensor storage overflow. It also has to reduce energy expenditure of mobile sensors when transmitting to a SAP, disruption to communications ongoing between MS in the vicinity of a SAP, increase the security of the system by probabilistically reducing overhearing and explicitly limiting the number of nodes offering (authenticated) proxy service on behalf of the SAP.

In [18] the author proposes an architecture that aims to solve these problems and find the best way to task MSs and collect data from MSs in support of delay-aware applications. The sphere of interaction is implemented by both transmit power control and by multihop signaling between SAPs and MSs that happen to be near the SAP for while. During this time they might be used to funnel packets from data collecting MSs to the SAP, and to relay tasking messages from the SAP to MSs. By doing this it is possible to maintain a sufficient amount of MSs and hence mitigating the tasking availability problem.

This solution also takes into account delay-aware applications and even though it does not provide real time treatment to such applications it does provide a functionality. This permits MSs to stipulate a degree of time sensitivity along with the tasking request in an effort to lessen the tasking prediction problem and offer an extra effort to produce the tasking results within the preferred time frame.

Halo's [18] architecture implies the existence of many triggers for increasing or decreasing the sphere of action but they believe the main driver for sphere interaction should be the fulfillment of application requests. It is suggested that the sphere should increase when the application demands require it and should decrease when possible to reduce power consumption and disruption to communications. It is proposed that changes be made to the traditional lazy approach where the SAP waits for the mobile device carrying the sensed data to reach the sphere of interaction. To achieve such results they propose a mathematical formula that is used to dynamically calculate the sphere of interaction of a SAP depending on deadlines of tasks scheduled and the distance a certain device has to travel in order to reach the desired target location. For example if a MS matching the tasking requirements is inside the sphere of action nothing is done. Otherwise if no matching MS is present or an MS has performed the sensing but is outside the sphere of action then the SAP calculates a new sphere radius in order to fulfill the tasking request.

The authors conducted evaluation procedures to better appraise the performance of such architecture on a real case scenario. A field with no radio obstructions was assumed and multi-SAP mult-MS scenarios were devised using TOSSIM [20] and Tython [21]. TOSSIM was used to simulate the Halo on the TinyOS platform and Tython to simulate mobility and connectivity. The test field was a 500x500 field according to a random walk modifier. Tests showed that the number of rendezvous increase when the MS density is also increased. In order to better understand the impact of the sphere of action 3 scenarios were tested: MIN where the sphere is at its minimum radius, a MAX where it is at its maximum and the ADAP where the sphere uses the proposed formulas to adapt according to the enviorment. Results showed that the MAX scenario got better results but consumed much more energy than the two other schemes. The ADAP scenario had better rendezvous results than the MIN and also consumed less energy than the MAX. Max scenario had a 60% rendezvous rate while the adapt had 40%. No power consumption results were presented.

2.2.1.B BikeNet

BikeNet is an opportunistic network that helps improve cyclist's experience while riding bikes through their usual path. This is done by gathering information about their path's characteristics like air pollution and the number of cars that pass near by. BikeNet also provides users with a web-based portal that incorporates various representations of their data. It also allows for the sharing of cycling-related data (for example, favorite cycling routes) within cycling interest groups. In order to achieve this type of service an OSN has to exist along the path to opportunistically collect the information about the cyclist while he cycles through said path.

This OSN uses a centralized approach resorting to SAPs along the path to work has gateways for the information collected from the sensors installed to the user's bike. The main difference from the approach mentioned in [18] is that the SAPs don't task request to other devices but instead convey the information to a centralized server which then processes corresponding information. This approach solves the tasking prediction and availability problem as long as the path on which the user is has SAPs available or if the user has a mobile SAP with him (e.g., a mobile phone with a wireless data connection) .

It is a simple approach and yet effective but has a downside. Since all the tasking requests are processed in a remote server the mobile device must have a data connection. In case the back-end server fails the entire system will become useless.

2.2.2 Decentralized

As opposed to the centralized solution, a decentralized architecture does not require any additional devices be them SAPs or back-end servers. On a decentralized architecture devices communicate on a Peer-to-Peer (P2P) fashion by signaling other devices using available radio interfaces. As such they require communication protocols so that they can communicate amongst themselves in an organized fashion. Contrary to the centralized approach, devices survey the area for available resources shared by other devices and chose the one that suits their needs the best. Following the same process, when it comes to offload and/or collecting data from opportunistic tasks, the communication is done through direct data transfer using the available radio interfaces.

2.2.2.A Quintet

Quintet [16] suggests a decentralized approach to a sensor sharing protocol that includes a distributed context analysis engine that determines sensors in the neighborhood that best match the query. This sensor sharing protocol dictates how sharing sessions are initiated and maintained on a decentralized fashion with the use of peer-to-peer signaling.

As a consequence of being a decentralized solution Quintet is required to run on each mobile device. Quintet relies on SQL based query messages to perform requests amongst devices. These queries messages are processed by the query processor while the sensor sharing associations are managed by a sensor-sharing controller.

If, for example a mobile device is queried by an application it first has to decide whether it can satisfy the query or should it try to find a better-suited solution in the vicinity. This decision is done by the means of the context analyzer. Considering that the decision is made to find a better-suited solution from a neighboring device, Quintet suggests a three stage process: resource discovery, resource selection and data transfer. In the discovery stage the mobile device seeks one or more nodes that are willing to share sensor data. This is done by broadcasting a sharing solicitation message that contains information about the sensory type and context requirements. A better-equipped mobile device upon receiving a solicitation message replies with a solicitation reply message that has information about how closely their characteristics match with the query requirements.

After receiving the solicitation reply messages the primary node in question is set to enter the resource selection stage. Here the node compares all the replies and using context analysis techniques selects the best responding solicited nodes. After selecting said nodes the primary node sends a unicast message to the best matching candidate set. The sensor sharing association is complete when each of the selected nodes reply with an acknowledgement.

In order to account for the fact that mobility affects the requirements and the characteristics of previously selected nodes, the primary node periodically reevaluates each of its sharing associations. This is done by re-soliciting the same query and again comparing the solicitation replies. If a sensor decides to in fact change an association for a determined query, the select message will act as a cancelation message for the previous association.

The author proposes that the data transfer stage in Quintet is done by direct streaming of the sampled data from the sharing to the primary node. It relies on a per-packet positive acknowledgement so that is its possible to recover from packet loss as quickly as possible. Mobility may bring the primary node and the sharing node out of common radio range at any time.

Comprehensive tests where made to evaluate Quintet's performance on all functionalities. Some focused on evaluating its performance improvement when using the sensor sharing techniques on the decentralized architecture. These test procedures where done using a discrete Markov model to simulate the node's movement. A baseline opportunistic network model was also used as comparison. The results showed that with a fixed number of nodes containing a sensor β and only varying the deadline there were improvements up to 16%. It was also observed that when the deadline increased the probability for improvement decreased. When the deadline was maintained but the number of nodes containing the same sensor β was varied the improvements were up to 70%. Best results were achieved when the number of nodes containing the aforementioned sensor was at a minimum. Increasing the number of nodes with said sensor reduced the probability of improvement when compared with the baseline OSN.

2.2.3 Scheduling

Scheduling is an important factor to take into consideration when a device has many scheduled tasks. On the centralized approach it is probably where this issue assumes a crucial importance since all tasking requests pass through the SAP. Consequently they need to be able to be efficiently managed so that the best tasking throughput is achieved. This should be done while being able to keep tasking times inside the preferred time frames stipulated by the MSs that requested said task.

In [18] a scheduling approach is suggested. The suggested approach determines that scheduling should be atomic. The authors also suggest a scheduling discipline that ascertains the order in which the tasks should be served.

Resorting to previous studies on the matter [22] it was concluded that at walking speeds and with a relatively low density of MSs, when simultaneously uploading, none of the tasking requests where complete. Furthermore it was also observed that because of attenuation caused by the human body the contact time between SAPs and MSs is reduced. These reasons justify the use of an atomic scheduling scheme. Also, the use of simultaneous uploads would increase the MAC layer overhead.

As for the scheduling discipline the authors suggested an approach that discards First-in-First-out (FIFO) as a possibility. The reason for that is that it does not take into account important features such as size (i.e., number of bytes) of tasking and uploading operations, and the MS dwell time in the SAP sphere of interaction. Thus, this naive approache can lead to a lower operation throughput due to non-uniform MS inter-SAP-visitation times, hereafter orbits. The proposed solution is hybrid Mobility Based Shortest Job First (MB-SJF) that decides which MS is automatically served.

Uploading and tasking operations are ordered by Prob(A). It reflects the size of the operation to be completed and the estimated dwell time of the associated MS in the sphere of interaction. The ordering of tasks is also affected by a priority factor that can be used to prioritize certain events (e.g., Toxic Spill).

The authors compared their hybrid MB-SJF with two other schedules, FIFO and Random. They observed that with a fixed upload size when the number of MSs grow their schedule showed better improvement than the other two candidates. The same was observed for medium sized files. As the files got larger the improvements begin to diminish on all three scheduling schemes.

2.2.4 Discussion

The available techniques should suffice in order to improve sensing probability where a desired sensor is not available, that is, when used in tandem with the mentioned architectures. Both architectures yielded promising results in improving sensing capabilities of mobile sensor nodes.

The choosing process of aforementioned architectures will depend on what is intended to implement.

A centralized approach make sense on a confined area, smaller scale deployment since a large scale deployment would make it expensive and hard to maintain. A decentralized approach would fit better on deployment that didn't define any spatial boundaries.

2.3 Social Sensing

In order for social sensing to be possible the mobile device needs to be able to infer the contextual information of the surrounding area and the user. This section will focus on techniques that help infer contextual information while taking advantage of the underlying opportunistic connections. A couple examples of social sensing applications will also be presented as an effort to better understand what are the possibilities and limitations.

2.3.1 Techniques

In [23] the authors propose techniques to achieve better accuracy and scalability of mobile sensing applications. These techniques are a part of a novel collaborative reasoning system.

2.3.1.A Model Pooling

Model pooling is a technique that aims at reutilizing models that have already been built and are possibly on other phones. With pooling, phones can exchange models whenever the model is available from another phone, thus allowing mobile phones to quickly expand their classification capabilities.

2.3.1.B Collaborative Inference

The combining of the classification results from multiple phones can be used to achieve better and more robust inference. This can be done after all phones pooling, and hence, having all the same classifiers. At this point they can all run the same inference algorithm and at the end they can combine the output and reach an agreement.

2.3.2 People-Centric Applications

2.3.2.A SoundSense

SoundSense [12] is one of few people-centric applications on smart-phones. SoundSense revolves around the only ”sensor” present on any phone, the microphone. The authors consider the microphone to be the most ubiquitous of all available sensors and that it is capable of making sophisticated inferences about human activity, location, and social events.

The authors propose an architecture and a set of algorithms for multistage, hierarchal classification of sound events on mobile phones. They also propose to address the scaling problem through the introduction of an adaptive unsupervised learning algorithm to classify significant sound events. In SoundSense [12] the authors expose an interesting observation explaining that with these sort of architecture and algorithm it is not possible to train every sound event that exists because the data set would be too big. As such, they propose that only sound events related with the user's life get classified.

The SoundSense architecture is divided into four main stages: Preprocessing, Coarse Category Classification, Finer Intra-Category Classification and Unsupervised Adaptive Classification. In the preprocessing stage the audio stream is segmented into frames of uniform duration. Spectral Entropy and energy measurements are used to filter frames that are silent or are too hard to classify accurately due to the context (e.g., far away from the source or muffled inside the user's pocket). The coarse category classification then makes an assessment of the frames and classifies them into one of the following categories, voice, music or ambient sound (i.e., everything other than voice and music). Since all stages are hierarchical this assessment dictates which further classifications stages should be applied. For example, some applications might only require that a category is inferred, therefore discarding an audio frame as useful. On the next stage (finer intra-category classification) the classification is broken down even more. If, for instance, on the previous stage a frame was classified as music, this stage could classify the genre. The last stage is responsible for learning sound clips that are significant in a person's life over time in terms of frequency of encounter and duration of the event. It also adapts the classifier itself to recognize these sounds when they occur. In this stage, the user is prompted to provide a textual label to associate with the sound.

Performance tests where made to evaluate the application's resource consumption and also to assess the application's success rate when classifying sound events. Regarding the performance results the authors observed that the maximum CPU load was reached when processing ambient sounds. CPU load values ranged from 11% to 22%. When no sound event was detected the CPU usage hovered around the 1% and 5% mark. Memory consumption never exceeded 5MB. As for the sound event classification success rate ambient sounds where correctly classified 90% of the time. Speech and music never passed 80%. Some of the slow and smooth sections of music are mistaken for ambient sound, and some allegro tunes are confused with speech. Similarly, around 14% of the speech samples are misclassified as music.

2.3.2.B CenceMe

In [11] a different approach into social sensing is taken. The CenceMe implementation proposes a network composed of mobile devices and the use of a back-end servers. The main difference from the SoundSense approach is that instead of focusing on just one sensor, this approach intends to take advantage of all sensors available. The classification of the raw data is used to produce primitives that represent the user's context. These primitives are achieved through: (i) the classification of sound samples from the microphone using a discrete Fourier transform (DFT) and a machine-learning algorithm to classify the nature of the sound; (ii) The classification of on board accelerometer data to determine activity (e.g., sitting, standing, walking, running); (iii) scanned Bluetooth MAC addresses in the phone's vicinity; (iv) GPS readings; and finally (v) random photos. The more complex algorithms run on backend servers.

The CenceMe application evaluation was done during a one-week period with the participation of 8 users using the application on a Nokia N95 device. The annotated results every 15 to 30 minutes. Only the activity events and audio events, using the accelerometer and microphone respectively, were evaluated.

After analyzing the results, the authors observed that they achieved a 20% lower accuracy when compared with another study using a custom device [24]. It should be noted that only the accelerometer was used in the CenceMe study. The application had some difficulty differentiating sitting from standing. On the audio evaluation process a high rate of false positives where observed. A phenomena that the users attributed to the classifier design and "miss-annotations". On the matter of performance the CPU usage ranged from 2% when idle up to 60%. Memory consumption ranged from 34 to 39.56MB.

2.3.3 Discussion

During this section two techniques for improving mobile sensing and applications regarding this same aspect where mentioned. Regarding the techniques not much can be said since they are not exactly options. They are techniques that can and should be used together in order to improve the sensing accuracy. They could even be considered a single technique divided into two stages. As for the example of social sensing applications a few conclusions can be drawn. Firstly that it is possible to implement such an application on a modern smart-phone without completely draining the device's resources to a point that the primary functions stop working properly.

Results showed that the CenceMe application has a bigger CPU and memory consumption, a factor that will probably lead to worst energy efficiency. This higher CPU consumption can be attributed to the complexity of the architecture that features more than one sensor and more complex event classifiers. On the other hand SoundSense proved to be highly efficient even when processing the more complex sound events.

SoundSense also showed better results when classifying events while the CenceMe application showed some difficulties especially during activity classification using the accelerometer. Nonetheless it is clear that such social sensing applications are possible to implement with enough accuracy to make them interesting and worthwhile developing.

2.4 General Discussion

This chapter focused on the two main areas of interest for this project, opportunistic sensing and social sensing. It was intended to survey the existing technologies and techniques in order to further understand what is the best direction to follow in order to overcome the previously mentioned challenges.

Regarding the opportunistic angle of the project the approach taken is tightly bonded with the problem specification. In this case, where the pretended OSN will be boundaryless with a very variable node density across a possibly very large area, the centralized approach doesn't seem to be the most appropriate solution. The decentralized solution seems to fit nicely. It is also safe to conclude that the use of aforementioned techniques will with high probability improve and help implement the a decentralized OSN.

On the subject of social sensing the use of only one sensor seems to be the best way to achieve acceptable performance and accuracy levels. Since the whole idea of the pretended OSN is to explore the opportunistic angle in order to help less capable devices overcome their limitations, a social sensing application that infers events using more than one sensor seems to be appropriate. The problem of complexity and accuracy should be considered secondary.

3 Architecture

Contents

This chapter will focus on understanding the project's requirements and on discussing the chosen architectural approach. All layers and modules will be presented in a top-down approach. This well help contextualize the next chapter, where the implementation details will be looked at in detail. All decisions made were based on tests made with the ONE simulator [25] and/or because of software/hardware limitations. Some decisions will be backed by results presented in the Tests and Validation chapter. Others will be explained in the Implementation chapter.

3.1 Requirements

The project description implies two main, already mentioned, requirements. Firstly, a network architecture should be designed and implemented to support the opportunistic connections amongst devices. It must also support activity logging from all devices. Furthermore a social sensing application should be created to run on top of the opportunistic network to better understand and evaluate the network's performance. Each requirement has other corresponding sub requirements that will be dully mentioned in the following paragraphs.

As referred in the general discussion section of the previous chapter, the proposed architecture will be a decentralized architecture and consequently all mobile devices have to run an instance of the developed code. The device's resources have to be considered since it is crucial that they are not depleted by the application itself, be it the CPU, memory or battery. These requirement are vital since it is imperative that the smart-phones maintain normal functionalities even when processing the most complex events. To satisfy the activity logging requirement all devices will also communicate with a Transmission Control Protocol (TCP) server. This part of the architecture is composed of a centralized topology.

As pertaining to the opportunistic solution it is required that devices are able to communicate in ordinary everyday circumstances. As such, it is important for the suggested solution to consider the short rendezvous situations that devices will most probably be confronted with. This is due to the nature of human behavior and movement. Moreover, communication costs with neighboring devices should be kept at a minimum level. Since the architecture will be presented in a decentralized manner, a great deal of messages will be required in order for devices to be able to query the surrounding ones, get to know their capabilities, perform requests and if possible receive requested data. To prevent flooding and excessive message exchange an efficient yet stable communication protocol should be envisioned. The network solution should improve the possibilities of devices completing tasking requests from the social sensing application.

The centralized server must be able to handle concurrent requests in two different levels. First it must be able to process concurrent requests from two or more mobile devices. It must employ a first-in first-served scheduler when it comes to the logging process. Second, since the PhoneSensing application will be multithreaded there might be concurrent requests to the TCP server from the same device. The server must comply with such situations. All information gathered via this centralized solution must be logged into the a Entity-relationship (ER) database. A web server provides access to the logged information.

As for the social sensing application the requirements should be considered less critical. Its function is to provide an example of how a opportunistic sensing network using smart-phones can improve application functionality in a real case scenario. It is also true that said application should be able to provide an acceptable accuracy when inferring the smart-phone's contextual information with the use of the available sensors. The possibility of collaborative inference should be available to further explore the underlying OSN.

3.2 Architecture

Figure 3.1 is an overview of the implemented architecture. It shows the two main interest groups. The decentralized communication between mobile devices and the centralized communication used for logging and result analysis.

They communicate amongst each other using bluetooth and with the centralized TCP server by way of Wi-Fi. Inter-Device communications are done with Bluetooth because it is the only resource available that allowed ad-hoc connections. Due to driver restrictions, the smart-phones used during the development don't yield Wi-Fi ad-hoc connections.

Also, using bluetooth left the Wi-Fi radio interface free for the centralized communications with the TCP server. It would have been very complex to schedule the use of the WiFi for both protocol communications and for logging purposes.

The application will be the tool used to prove that the concept works. The idea behind it is a simple application that has various configured profiles. Each profile is suited to a different situation, and the sensors gather information process it, and decide wether the surroundings are suitable for the selected profile. This pertains to the social sensing part of the solution.

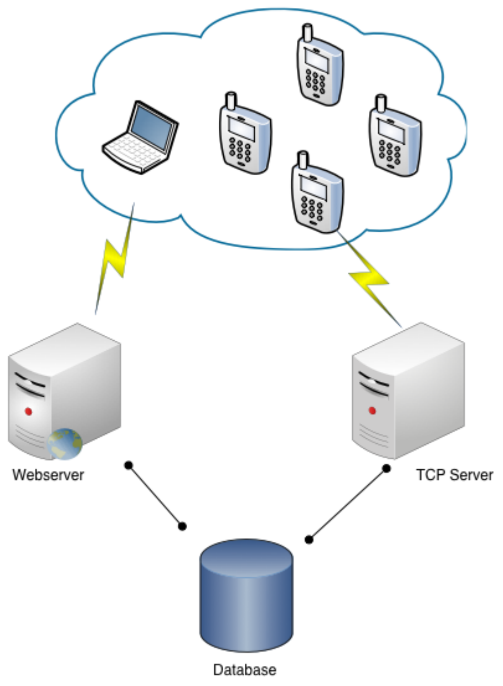


Figure 3.1: Architecture Overview

When a device is less capable than others, and doesn't have a required sensor it would then use the underlying OSN composed of other devices running the same application, to acquired said resource.

The application uses a framework that consists of functions that implement the OSN's communication protocols. The framework can be used as a base for any other that would find it useful to use the resource sharing protocol to enhance it's capabilities.

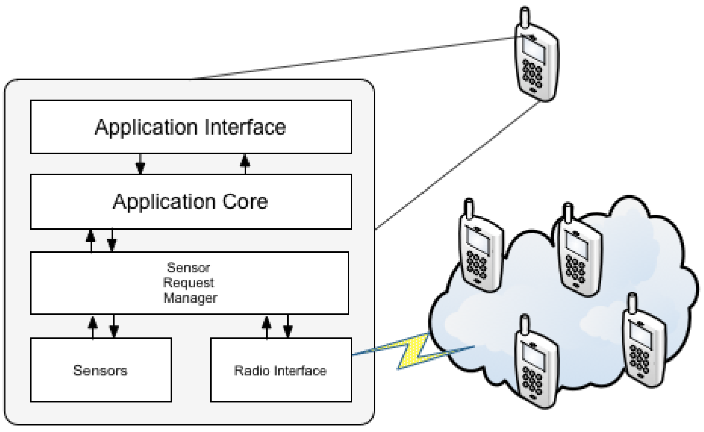


Figure 3.2: General view of the proposed architecture

Figure 3.2 shows a global view of the architecture's stack structure and its communication paths between modules. The Application Interface and the Application Core are relative to the social sensing application. All other modules refer to the application's support framework.

We'll start deconstructing the application layers in a bottom up approach, starting the the OSN framework.

3.3 OSN Framework

The OSN framework provides applications with the necessary tools to seamlessly use an underlying OSN to it's one agenda. It abstracts the application from any logic associated with sensor requests.

3.3.1 Radio Interface

This module manages the incoming and outgoing bluetooth connections. It acts both as a client and server. It guarantees that only one connection is active at any given time.
Consequently the scheduling process is extremely important. The Radio Interface Module (RIM) guaranties that when a connection is active that a lock is enabled. This lock prevents that any other incoming or outgoing connections are established.
The locks are acquired with a message lock request through service messages. The lock strategy isn't blocking. If the request isn't granted then algorithm moves on and tries again later to establish a connection.

Besides managing the bluetooth access strategy the RIM implements also part of the OSN protocol logic. It does so by deciding what application module to message information after handling an incoming connection from a remote device. As part a solution to isolate PhoneSensing user's the bluetooth device listens on a specified port.
Further details on this module will be explained in the next chapter.

3.3.2 Sensor

The sensor module, like the RIM, is an independent service that runs in the background. The sensors play an important part in the application's process and are a key instrument in the OSN protocol.

This module registers the number of different sensors that the device has. The registered sensors are configured in the application's menu. Upon registering a sensor, the Sensor Module (SM) registers a handler for that specific sensor.

The handlers are asynchronous processes that run in tandem with the SM. They have isolated executions because each sensor has a specific data analysis algorithm. For example the microphone's handler has to calculate every few seconds the surroundings noise value (in db).

Upon registering a handler, the SM receives periodical information from that sensor. That information is processed according to the profile configurations for that specific sensor. The periodicity on which the information is received depends on the sensor.

The SM also handles requests from the Sensor Request Manager Module (SRMM). Since all sensor handlers are asynchronous the SM is always available to handle a sensor request from a remote device. Requests are received through service messages.

3.3.3 Sensor Request Manager

The SRMM is quite similar to the SM. It also works in an asynchronous fashion. The SRMM handles incoming and outgoing requests, from and to remote devices. This module acts as a bridge from RIM to the SM. As incoming requests arrive they're issued and queued. The queue employs a First In First Out (FIFO) scheduler.

The outgoing requests are processed in parallel with the incoming requests. The only concurrent resource they both share is the bluetooth device, since like already mentioned, it does not support concurrent connections even though they're in opposite directions and with distinct remote devices. Once an outgoing request has been queued, other outgoing requests won't be accepted. This is done to ensure that duplicate requests aren't emitted. This prevents flooding of the surroundings with protocol messages.

3.4 PhoneSensing Application

The Application Core Module (ACM) sits in between the user interface and framework modules. It has the job of orchestrating all the applications background operations. These background operations include handling results from the SRMM, deciding what's the process next in the algorithm, user interface flux and settings management.
The figure 3.3 decomposes it into macro blocks.

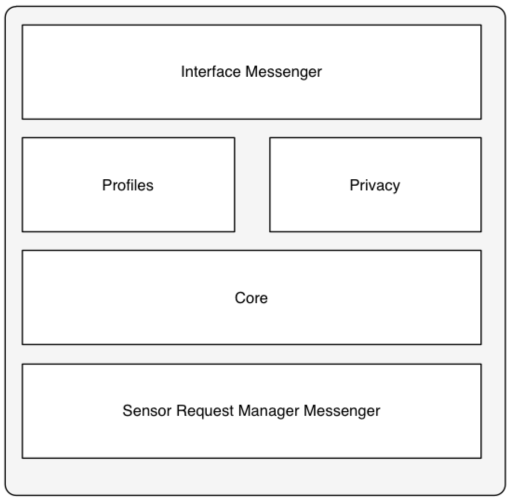


Figure 3.3: Decomposition of the Application Core

3.4.1 SRMM Connector

The SRMM Connector is no more than a message handler registered with the SRMM. It acts has a message pipeline so that both modules can exchange service messages.

3.4.2 Core

The Core block is where most of algorithmic logic is placed. All logic passes through this block, one way or another. For example when a sensor is requested by a remote device part of the algorithm is ran in the SM but, after that is done, the Core is messaged through the SRMM connector.
The core then processes the message and triggers the next step in the process.

The remote logging processes that communicate with the TCP server are also included in this macro block. This block is purely representative since implementation wise most of the code is divided up in to independent classes so scalability isn't an issue.

3.4.3 Profile

The profile block is the most visible block on the application stack. It is responsible for all profile configuration. This configuration is where all algorithm and protocol decisions are made on.
Light and Sound thresholds are chosen according to the profile's context. These values are then used by the SM to set the triggering mechanism.

3.4.4 Privacy

A small part of the application development is concerns privacy. It exists to exemplify how the user's privacy could be easily configured. This has some relevance since the application proposes a solution where sensible contextual information is shared with anonymous remote devices without any user input. It provides the user with the possibility to select which sensors he wants to keep active at any given time.

3.4.5 Interface Messenger

Similar to the SRMM Connector, but it connects with the interface instead. This messenger is native to the Android Software Development Kit (SDK).

3.5 TCP Server

The TCP server is part of the centralized part of the solution. It is used for two reasons. First it allows for on the fly logging of all processes running on the PhoneSensing OSN at any given moment. This logging process helps with debugging and workflow tracking of all requests.

Second, it maintains a status record of hardware variables from each device. This variables provide valuable insight on the PhoneSensing application's performance. These variables will be looked at in some detail in the Test and Validation chapter.

The TCP server runs on a Java VM. The database connectivity is established with JDBC and uses the Hibernate framework to map the ER database's tables to Java objects.

The Client Handler manages all incoming connections from the remote devices. Listens on a single port and for each connection creates a new thread that handles the reminder of the request.
This guarantees that it can handle multiple requests at the same time. The Hibernate framework manages concurrent access to the database with it's transactional manager. When it comes to the debugging logs an extra synchronization process is implemented.

The logs are written in the same order of their arrival. This is essential in order for the debugging process to yield a precise snapshot of the resource sharing process.

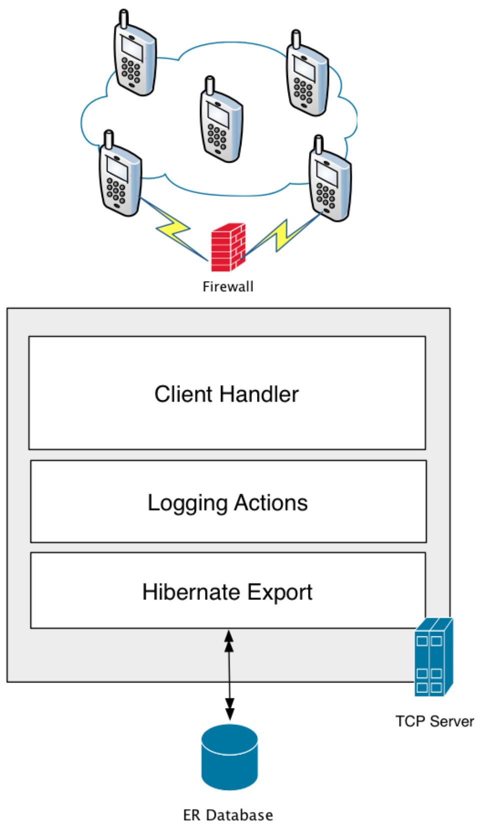


Figure 3.4: TCP Server Overview

3.6 Resource Sharing Protocol

As previously referred, the basic idea behind the proposed architecture above is to support a decentralized resource sharing protocol, something very different of what has already been done. Other solutions that were reviewed in chapter 2 focus on resource sharing solutions that rely on centralized infrastructures [18] [19].

PhoneSensing proposes a different approach to the challenge. The base for the proposed solution, apart from the architectural organization that was just presented, is the protocol itself. Three different approaches were initially developed. After a round of intensive testing one was eventually chosen.

All three protocols follow the next basic guidelines.

1. They can not rely on any centralized infrastructure to communicate, identify and exchange resources with neighboring devices

2. Message exchange can not be high in order for it to scale in high node density scenarios

3. Conversion times have to be low so that in low *rendez vous* times the protocol still allows for resources to be shared
4. Exchanged messages have to be small so that transfer times aren't long

The first guideline is already guaranteed for all protocols that use the PhoneSensing framework, since it depends only on P2P bluetooth connections amongst devices.
The forth guideline is guaranteed with RDL. RDL is a unified description language for network resources. It is able to describe a resource both qualitatively as quantitatively [26].
The RDL API exposes two formats to describe the resources, XML and KLV. The KLV format is very lightweight, as a matter of fact due to it's size it can consume as little as 14 times less than the counterpart format [26].

The second and third guidelines are the draw breakers, these are the guidelines that define the most suitable protocol for the selected architecture.
The message exchange will vary depending on the discovery process, handshake process and resource sharing. This message exchange will also affect the third guideline, where the protocol must converge to resource share in the lowest time possible. They should be both tightly bonded since if less messages are exchanged the connection locking times will also be lower. Hence unavailable signals will be less frequent and conversion times will be lower.

All three protocols use the same messages. During the tests only four different messages where considered, but during the implementation a couple more had to be used. In the next chapter this aspect will be analyzed in greater detail.

3.6.1 Messages

3.6.1.A Capability Request Message

The Capability Request Message (CREQM) is sent to a device when the sender wants to query the remote device's available resources. During the testing phase some attributes had to be added to the message for reporting reasons.

- messageId
- host
- requestId

During the implementation of the Resource Sharing Protocol in the PhoneSensing application, the message was represented simply by the character array, CAP_REQ. The next chapter will explain the reason behind this decision.

3.6.1.B Capability Response Message

This message is sent as a response to the Capability Request Message.

The Capability Response Message (CRESM) during the testing phase in the simulator has the same attributes used for reporting reasons as the previous message. Besides those attributes it always carries a list with all the available sensors.

In the implementation phase this message is represented by a byte array. This byte array is the RDL describing the available resources in KLV format.

3.6.1.C Sensor Request Message

The Sensor Request Message (SREQM) has the same attributes as the CREQM.

In the implemented application this message is represented by a RDL describing the requested sensor. This RDL descriptor is in the KLV format.

3.6.1.D Sensor Reply Message

The Sensor Response Message (SRESM) has the same attributes as the previous message.

In the next chapter these messages will be looked at again, but in a different context. For the case study scenario they had to undergo a few changes. During the analysis of the three different protocols the message structure that shall be considered is the structure used during the testing phase.

3.6.2 Protocols

The goal of the protocol is to in the end receive data from a sensor that isn't available locally at the device. Hence, a request reply handshake must exist. The flux of sending a sensor request and receiving a sensor reply is unavoidable. All three protocols share that phase of the message exchange. All three don't resend requests or replies if the message was successfully delivered. Messages are only resent if they failed to reach the destination. They all have a Time To Live (TTL) value associated, this is so if a request reaches the destination and the destination goes out of range for a long time, the reply

isn't sent when / if they connect in the future.
None of the protocols use ACK or NACK messages. The sender is always blind, it never knows if the messages where successfully sent. On the implementation phase this turns out to be false.

The process that can vary is the handshake phase, the phase that leads to the sending of a SREQM. The three protocols that follow have three different approaches.

3.6.2.A Sensor Request Only

Requesting Device:
The Sensor Request Only (SRO) protocol is the most basic of them all, has almost no logic associated to it. It completely skips the handshake process, and when it requires a sensor it sends a SREQM to all neighboring devices.
It ignores the fact that the remote device might not even have the requested sensor. This floods the system with what might be useless messages. Behind this anarchy there is some logic, since it avoids sending CREQMs.

After sending all the SREQMs, the device waits to receive a SRESM. Upon receiving it, it will discard all SRESMs associated with that request.

Requested Device:
With this protocol when a device receives a SREQM it replies if it has the requested sensor, else the message is discarded and ignored.

3.6.2.B SingleActiveRequest

Requesting Device:
The Single Active Request (SAR) protocol does not skip the handshake process. When a device requires a sensor it does not have he sends a CRESM to all neighboring devices. It then waits for a CRESM. Upon receiving one it checks if the device has the desired sensor.
If this is the case, then a SREQM is issued to that device. With this protocol, after sending a SREQM all other CRESM are ignored. The protocol assumes that since this node has the required sensor there is no need to send any other requests.

In the case the remote device does not have the desired sensor then that message is ignored, and the device is flagged as not having the required sensor. Other CRESM will be accepted.
The protocol converges when the device receives the SRESM. After that, all other messages referring to that request are ignored.

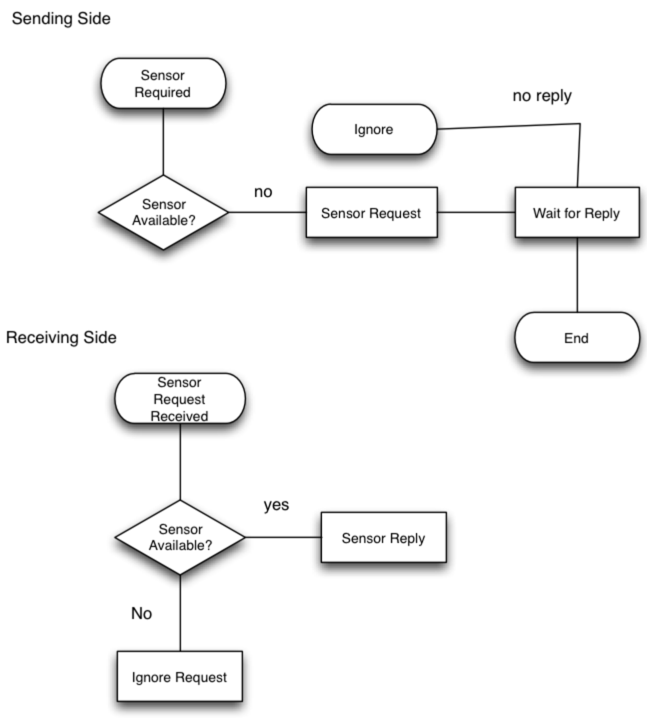


Figure 3.5: SRO protocol algorithm

Requested Device:
When a CREQM is received, a reply is sent to the sender. If a SREQM is received, then a positive reply is sent if the required sensor is available. Otherwise, the message is ignored.

3.6.2.C MultipleActiveRequest

The Multiple Active Request (MAR) protocol is practically the same as the SAR protocol. The only nuance is that this protocol won't stop sending SREQM even if one has already been sent. It assumes that the message might fail, or the remote node might not comply with the request. Hence, it sends as many requests as possible, and then waits for the first one to reply.

This protocol might send more messages, but it might converge faster since it does not depend on just one remote device.

All three protocols have their strengths. The SRO protocol aims at simplicity. In theory it looks like it will send less messages. The SAR protocol, is very optimistic and assumes that no messages will be

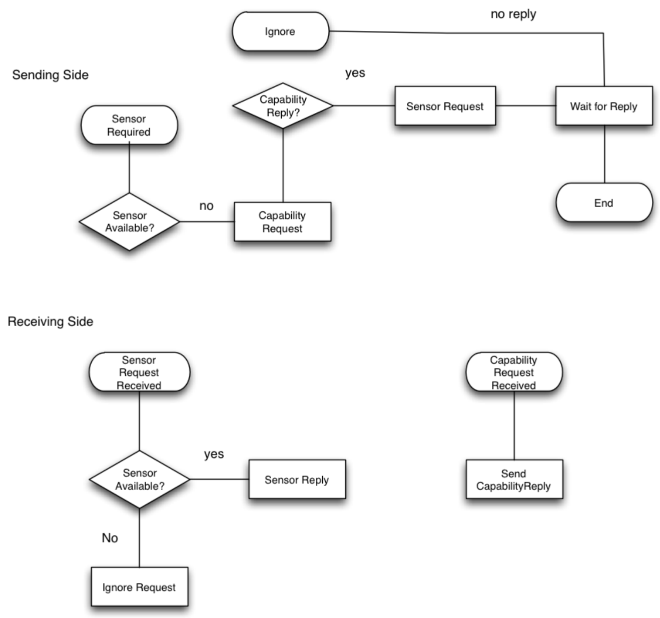


Figure 3.6: MAR and SAR protocol algorithm

dropped or ignored and that the remote device will never leave its range before complying to the request. If this assumptions are true, then it will send less messages then the MAR protocol. Saying that, it does not mean that it will converge any faster. The MAR protocol has more possible reply nodes, and may in some cases converge faster.

In order to reach a scientific conclusion to which of the three protocols was better suited for the pre-established requirements a series of tests were done. These tests focused on convergence times, message sizes, number of messages created and protocol efficiency (messages used / messages sent). They were also done in different environments, varying node density and also mobility factors. They were done using the ONE simulator and will be seen in greater detail in Chapter 5.

The test revealed that the best all rounder across all test scenarios was the MAR protocol. Like it was expected it did send the most messages, but at the end it was the one that converged faster. Another factor that was on the MAR's protocol advantage is that it was the one that had less ignored messages, in a way it was the most efficient.

All simulation results and details of all different test scenarios will be analyzed in detail in the Test and Evaluation chapter. The conclusions above are just to contextualize the choice made for the implementation process.

3.7 Discussion

This chapter focused on describing the implemented architecture. It showed how the physical environment is set up to support a decentralized resource sharing protocol using an underlying OSN. It also shows how the application communicates with the necessary peripherals to build the necessary framework and to gather data to feed the sensing application.
All three possible protocols were also described in detail. One was selected as the protocol to be used during the implementation phase. All results that led do that decision will be presented in chapter 5.

4 Implementation

Contents

The last chapter explained in some detail the proposed architecture for the solution. It was a high level overview that did not go into details, specially of how some processes were implemented.
The implementation of the proposed solution uncovered some platform and real world limitations, that had to be overcome with different approaches and workarounds.

This chapter focuses on giving a low level snapshot of the developed solution, highlighting the changes made to the original plan. Detailed analysis of all the main processes and solutions. The development process followed a structured workflow. The analysis will follow the same workflow.

The first development step was to set up a simulator to test the three protocols mentioned in the previous chapter.
After the tests were done, and the protocol chosen. The next step was to build a framework and an Android application to use as a proof of concept for what was proposed. Part of this proof of concept was to incorporate the social sensing angle to use as an example of a real world use of the resource sharing protocol on an OSN using smart-phones. This is the bulk of the implementation process, and will reviews as a whole.

Part of the validation process is to compare the theoretical results obtained from the simulator and compare with the case study sceario results obtained from the tests ran on the developed Android application. To do so, the logging tools for all processes running on the application had to be developed.

The rest of the chapter will focus on these three main implementation processes.

4.1 Simulator

The ONE simulator [26] was used as a base for the tests. It is a JAVA based simulator for Delay-tolerant networking (DTN) protocols. Since the proposed protocol have some characteristics of a DTN protocol, and mostly because the simulator implements different real world scenarios with mobile devices it is a very good baseline.
The real world scenarios include a scaled map of a city, where it is possible to configure if the mobile nodes use only sidewalks, only roads or both. It also possible to tweak the transmission speeds and range to emulate the radio interface. This made it easy to simulate the bluetooth's behavior.

The simulator also has a very complex random walk path generator for the mobile devices. This is very important when simulating the protocols behavior when mobility is a factor. It allows to simulate critical points of the protocol, like for example, short *rendez vous* situations.

Some adaptations had to be made considering the simulator wasn't made to test the same type of protocol as the PhoneSensing resource sharing protocol. The simulator was made with propagation protocols in mind, where mobile devices act as routers, more for mesh networks than P2P. The event generator was made for this purpose and it did not suit the requirements at all. So, both the routing protocol and the event generator had to be developed for the simulator.

It is possible to track the protocol's progress using custom reports built on the pre-existing simulator reports. Some development went into building custom reports.

4.1.1 Event Generator

The original event generator had a different paradigm than what was need to simulate the PhoneSensing resource sharing protocols. Before the application started it loaded an event file that had events that should occur at a given timeframe in the simulation. As the simulation progressed events would be triggered accordingly to what was configured in the file.

To test the resource sharing protocol it was required that events be triggered according to the resource they needed opportunistically and not with a predefined timeframe. Hence, the event generator was changed to run only once at the beginning of the simulation.

Basically this event generator loads two .txt files. One is the capability file, which contains the sensors available in nodes. The other is the event file, containing the sensor that each node requires. Consequently at simulation time 0, all nodes are loaded with the available sensors, and the required sensors. From that moment on, they will opportunistically try to acquire the required sensor, if they require one, from neighboring devices using the configured protocol.
Appendix A show's an example of both files. The files are generated in random fashion using a simple python application as seen on Appendix B.

4.1.2 Routing

The changes made to the routing algorithm of the simulator were simple. The developed class extends the ActiveRouter class already implemented by the simulator. This class implements the basic functions, like message forwarding and connection management. It also validates if the mobile device is available to transfer or to receive files. It emulates the same bluetooth limitation where only one active transfer is possible per device.

The PhoneSensingRouter for the simulator implements a one hop router, where the message is only sent to the destination, and isn't forwarded to any other device. It was a simple change to mimic a simple file transfer from one device to another.

4.1.3 Reporting

The ONE simulator has bundled a few basic reports that help with the data analysis of the tested protocol. Some specific reports had to be developed to ensure that all pertinent data was logged during the simulation.
All simulations run the following reports:

- CreatedMessagesReport
- DeliveredMessagesReport
- MessageStatsReport
- PhoneSensingApplicationListener
- PhoneSensingMessageListener

The first three reports are native to the ONE simulator. The CreatedMessagesReport logs all sent messages. The DeliveredMessagesReport logs all successfully delivered messages, logging also the delivery time and size. The MessageStatsReport reports global message stats, like for example, total number of created message, delivered messages, and delivery probability.
The other two reports were developed to convey information pertinent to the PhoneSensing protocols that the native reports didn't log.

4.1.3.A PhoneSensingApplicationListener

This reports on message delivery information. It is used to analyze the protocols efficiency and how many messages are considered useless. When a SRESM is delivered to a node, but that request has already been fulfilled by another reply that message is ignored and marked as such.
This property is common to all three protocols. The SAR protocol and the SRO protocol have unique situations where messages are ignored.

In the case of the SAR when a CRESM arrives, but a request has already been sent that message is ignored, and logged in as requested.

The SRO protocol has a unique message, the unavailable. This is logged when a SRESM is sent to a device that doesn't have the requested sensor.

4.1.3.B PhoneSensingMessageListener

This report logs the following global stats:

- Simulation Time
- Messages Transferred
- Bytes Transferred
- Average Message Size

This helps understand how the protocol might perform in a real world situation.

4.2 PhoneSensing Application

This section will approach all pertinent details about the PhoneSensing application, starting with a global vision. In chapter 3 the architecture focused on two divided macro blocks, the PhoneSensing application and the OSN framework. During the implementation phase they were joined up together as one. The PhoneSensing application has the framework built in.

This choice is a result of the development platform. The application was built for Android smart-phones, using the Android SDK. They run Android 2.1. Due to structural aspects related to android application development it made more sense that the framework was built in with the application.

Figure 4.1 gives a high level snapshot of the application's building blocks.

The red blocks (Android Middleware, Service Messengers, Receivers) represent native Android functionalities. The white blocks are the blocks that were developed during the implementation phase. The blue blocks represent the device's peripherals.

Before approaching the application's workflow and dissecting the implementation details we will talk about the main challenges that had to be solved before any implementation started. There were two main challenges.

4.2.1 Pre-Implementation Challenges

4.2.1.A Discoverable Mode

For the application to work it had to always be in a discoverable mode, so that neighboring devices could discover them and try to establish a connection. Natively this wasn't possible since the devices have a discovery limit time of 300 seconds.

So, when the application starts, it triggers a process that enables the device to be discoverable for 20s. When the 20s expire that Android OS will send a global notification with an ID that represents that the discoverable mode has expired. That notification will be intercepted by a receiver previously registered that extends the discoverable mode by another 300s. That receiver is part of the red block, tagged receivers in figure 4.1.

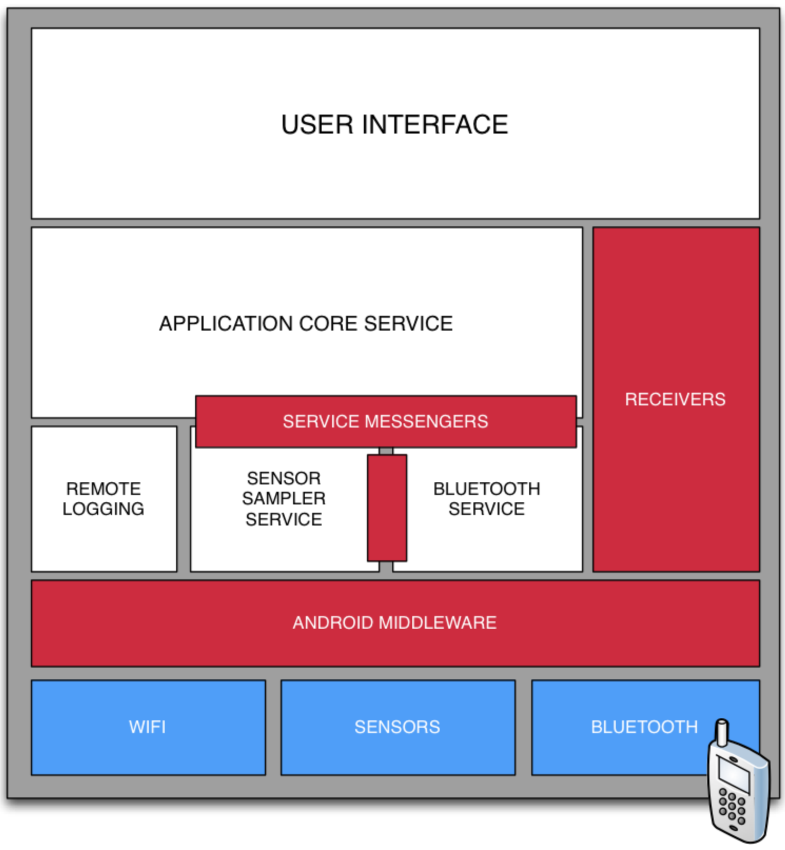


Figure 4.1: Application Overview

```
@Override
public void onReceive(Context ctx, Intent intent) {
```

```
Integer sm = intent.getIntExtra(BluetoothAdapter.EXTRA_SCAN_MODE, -1);
if(sm.equals(BluetoothAdapter.SCAN_MODE_CONNECTABLE) || sm.equals(BluetoothAdapter.SCAN_MODE_NONE))
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
if (mBluetoothAdapter == null) {
Toast.makeText(ctx, "Bluetooth Unavailable", Toast.LENGTH_LONG).show();
}
Intent discoverableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
discoverableIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
ctx.startActivity(discoverableIntent);
}
}
```

Unfortunately every time the discoverable mode is enabled the user is prompted to allow the action. This cannot be overridden.

4.2.1.B Discovery Process

The discovery process had two challenges to be solved. First, when the discovery process was done the devices weren't filtered, there was no way to know if the device was running the PhoneSensing application or not. This had to be done, since it made no sense to try to acquire a resource from a device that wasn't part of the OSN.
Second, the pairing process had to be avoided. Not avoiding pairing meant that every single connection had to be approved by both devices. This is unrealistic.

Both challenges were solved with the same technique. The Android SDK has a private method that isn't exposed. This method allows bluetooth devices to listen on a designated port and accept connections on that port without pairing. This is called an InsecureRFCOMM connection. Since the method isn't available it has to be invoked using JAVA reflection. This is implemented in the BluetoothService block. We'll talk about it in more detail later the in chapter.

Hence every device running the PhoneSensing application listens on the same port on a InsecureRFCOMM connection.

```
BluetoothServerSocket tmp = null;
try {
Method m = mAdapter.getClass().
```

```
getDeclaredMethod("listenUsingInsecureRfcommOn", new Class[] {int.class});
m.setAccessible(true);
tmp = (BluetoothServerSocket) m.invoke(mAdapter, 22);
...
```

With this it is possible to filter devices by just connecting to that port. If the connection is established with success, that the device is running the PhoneSensing application. This solves both the pairing and the filtering problem.

Challenges set aside, we'll now focus on the application's workflow during a event where a sensor is required and has to acquire it from a remote device running a PhoneSensing instance. Figure 4.2 shows how the application processes from the moment it is executed. The red blocks represent child threads.

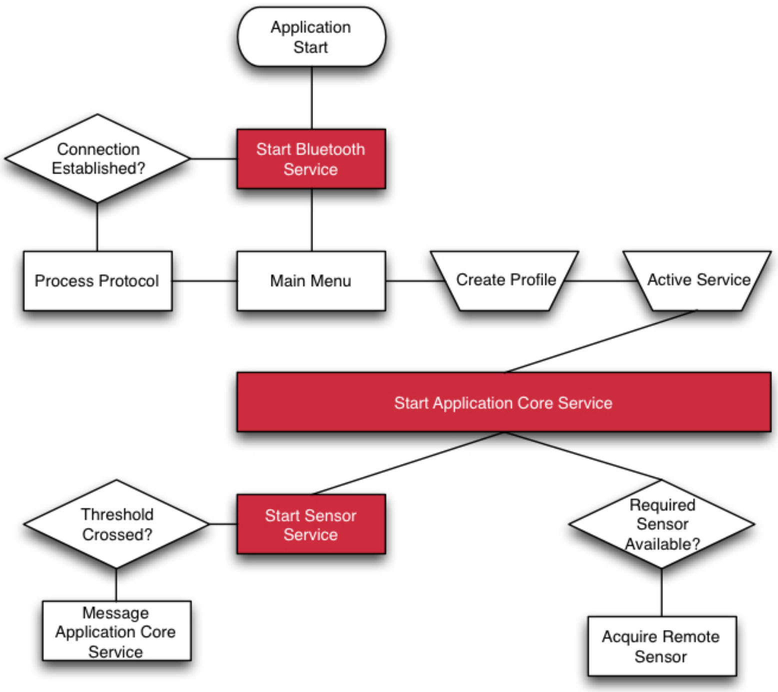


Figure 4.2: Application Workflow

4.2.2 Application Blocks

As seen on figure 4.1 the application is subdivided into blocks, and the service blocks are fundamental to the workflow process as also seen in figure 4.2. The first to be started is the BluetoothService.

4.2.2.A BluetoothService

As all other service blocks, the BluetoothService extends Android's native Service class. This makes it easier to integrate with the rest of the application, since it's managed by the OS.

```
public class BluetoothService extends Service {
...
```

Messages are exchanged with other service blocks through the Service Messengers show in figure 4.1.

```
class IncomingHandler extends Handler {
@Override
public void handleMessage(Message msg) {
...
```

The BluetoothService as said previously manages the incoming connections. To do so it has a very strict locking strategy to comply with the single connection restriction. It is composed of two types of threads, the AcceptThread and the ConnectedThread.

The AcceptThread has four different states:

- Listen
- Connecting
- Connected
- None

When its started the default state is Listen. It keeps that state until a connection arrives or a connection lock is requested. When a remote device tries to establish a connection on the listening port a new socket is created, and the state goes to Connecting. At this point the ConnectedThread is created and any other connection attempts are rejected.

When the ConnectedThread is created it starts waiting for bytes to arrive through the input stream. When the process ends the socket is closed the thread is killed and the AcceptThread goes back to the Listen state.

The BluetoothService accepts lock requests from other services that might want to use the bluetooth device. This distributed locking strategy guarantees that the concurrent services never access the bluetooth device when it's already processing another request. If a remote device tries to connect when the bluetooth device is handling another task, the remote device will get an error message. It will interpret it as the device being unavailable.

4.2.2.B SensorService

The SensorService is in charge of managing the available sensors. It acts as a bridge between the ApplicationCoreService and the sensors. Communication is done as in the BluetoothService through the Service Messages. In this version of the PhoneSensing application it only manages two of the available sensors, the microphone and the smart-phone's light sensor.
The managing process is twofold: First, it starts and stops listening the sensor's information depending on the application's state. This state is passed down by service messages. Second, when it is analyzing information it must also decide whether inputed date is inside the configured profile parameters. Each sensor has a different handling approach.

Light Sensor:

When a message to activate the light sensor arrives to the SensorService a listener for the Light Sensor is registered. At this time, every time there's a sensor change event, the listener will receive a set of data. From the received data the luminosity, (in percentile) is extracted and compared with the configured value. If it's below the configured threshold, the ApplicationCoreService is notified of the event.

Microphone:

The microphone is trickier to handle. As seen before the Light Sensor only triggers the Listener when there's a change in the sensor readings, which can be a very rare occurrence. The microphone sensor works in a different way. First, it does not have a Listener associated; it is a sensor that one taps into when needed. Second, the profile configuration value is set to dB. The Android Middleware doesn't provide that information, it only provides the possibility to record audio and then analyze it. Hence, the SensorService, has a specific thread for the Microphone because the processing is somehow heavier.

Like the Light Sensor, the Microphone sensor is activated when a message is sent to the SensorService requesting that it is done. When that message arrives the SplEngine thread is started. This thread will calculate in configured increments the Sound Pressure Level (SPL). This value is in db's. The SPL is a logarithmic measure of the effective sound pressure of a sound relative to a reference level. The usual reference level is 0.

To avoid draining the battery the value is calculated every few seconds. This is configured by the SAMPLING_RATE value. Recordings are done in mono with a 16bit PCM encoding at 44100KHz. This is the baseline used in audio recording on smart-phones. The Android SDK provides a method that calculates the minimum buffer size for this configurations.

The calculation process is done every SAMPLE_RATE/1000 seconds. This is done by reading BUFFSIZE bytes into a temporary buffer. This buffer contains the peak value for each sample. The average peak value (RMS) is calculated. The SPL value is calculated using formula 4.1. The CALIBRATION_VALUE is set according to the environment. When the tests were done the CALIBRATION_VALUE was -105.

$$20 * \log_1\left(\frac{RMS}{REF\_VAL}\right) + CALIBRATION\_VALUE \tag{4.1}$$

4.2.2.C ApplicationCoreService

Like the other two services it extends Android's Service class and communicates through the Service Messages already seen in figure 4.1.
The ApplicationCoreService contains most of the logic behind the application 4.2, both for the implementation of the protocol, but also for core processing. As we've seen it coordinates with the BluetoothService and the SensorService. This section focuses on explaining the most relevant features incorporated in the ApplicationCoreService. One of the more important features is the RequestManagerThread.

RequestManagerThread:

The RequestManagerThread is created when the ApplicationCoreService is started and works as an independent thread, like the SensorService. The smart thing about this thread implementation is that it's a private class inside the ApplicationCoreService. This allows it to share JAVA objects with the ApplicationCoreService, and that's how the requests are queued to the thread.

Figure 4.3 describes the request handling process. The process is very simple. The queue is based on a FIFO scheduler, and one request is served at a time. Next we'll focus on the red box labeled Send Sensor Reply. In this box is where the sensor reply part of the protocol is implemented.

Send Sensor Reply:

In the previous chapter the protocol specified that the SRESM was send blindly, the sending device had no knowledge of the outcome of the process. It was also said that there were no ACK messages involved. In the implementation phase that was changed.

First the device knows if the message was sent successfully since this is a stateful connection, and if something fails, i.e. sending bytes or establishing a connection, or if the connection is closed, both devices will be aware of this. Second, to synchronize the sending and receiving process there have to be acknowledgement messages exchanged.

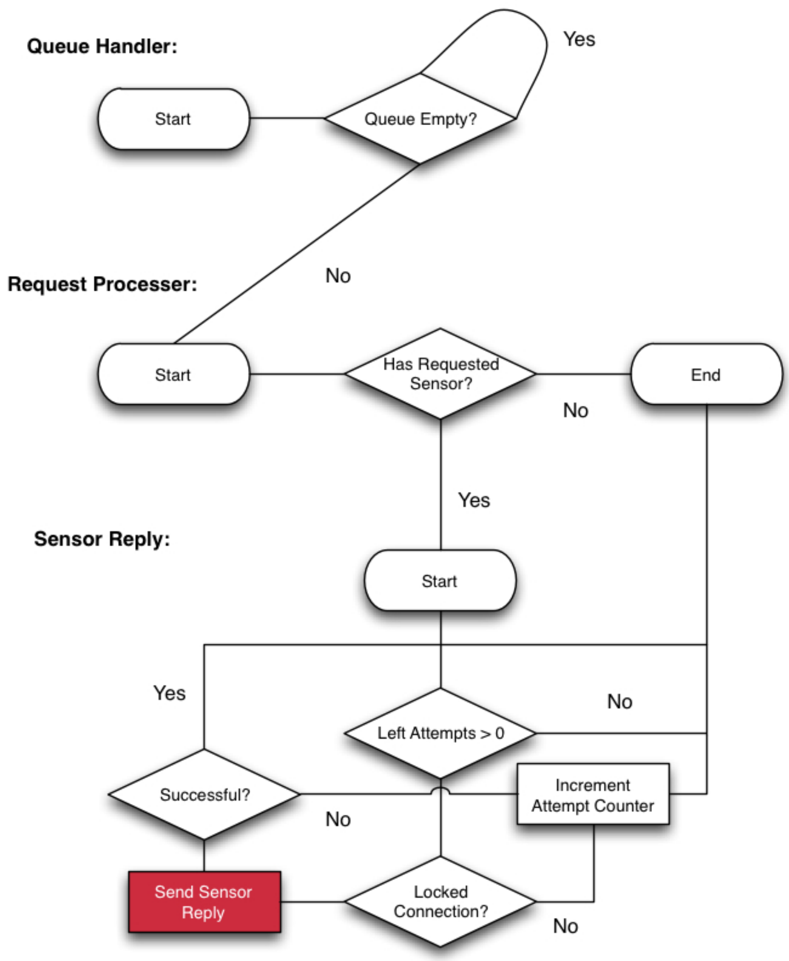


Figure 4.3: Request Handling Process

Figure 4.4 exemplifies the messages exchanged between both devices. At any point if the transfer fails, the connection is unlocked so that the BluetoothService can start accepting connections again. Here the SEN_REP is just a string, but in a real world situation it would be the RDL object in KLV format. At that point the sending process would have an extra synchronization process. That process will be described in this section when the rest of the protocol implementation is analyzed. Here the RDL isn't sent because the shared sensor (GPS) isn't available in the used devices, so it's emulated this way.

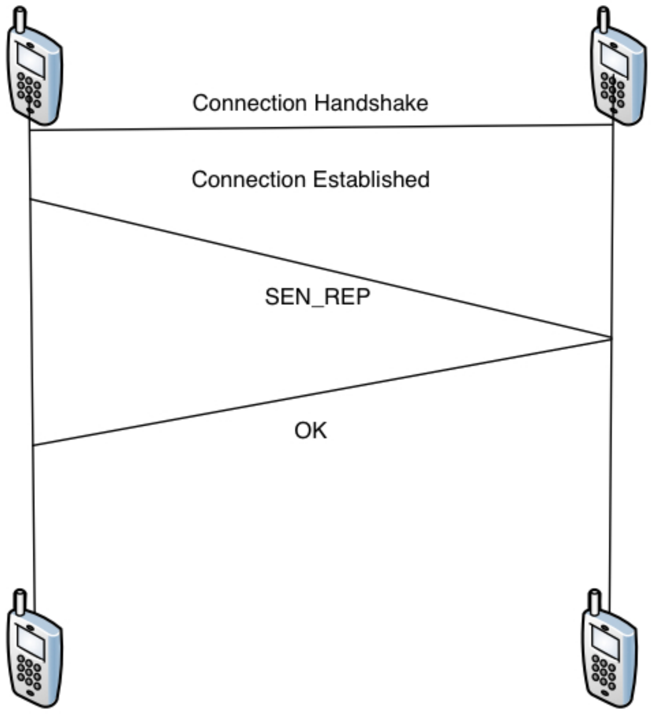


Figure 4.4: Sensor Reply Process

Now that the request handling process has been explained, the focus goes on how to these requests arrive at the device. This process also takes place in the ApplicationCoreService but also, has it has been said, in the BluetoothService.

Sensor Request Process:

The sensor request process starts when the SensorService messages the ApplicationCoreService notifying that one of the configured threshold has been crossed. In this application the message received is a RequestPositionChange message. When this message arrives the application tries to access the GPS, if one isn't available then the Sensor Request Process will start.

When the process starts, it validates if the discovery process hasn't already started. This is done to contend with duplicate messages sent from the SensorService. If it is not the case, then the Discovery Process is initiated. The discovery process has already been discussed in this chapter. The Application-CoreService has a receiver registered that intercepts the event that is triggered when a device is found and when the discovery process finishes. When a device is found, it's added to a list of devices. At the end of the discovery process the list of found devices undergoes the filter process discussed in section 4.2.1.B.

The filtering process is done with an asynchronous task. The asynchronous task is necessary so that the process doesn't block the ApplicationCoreService while it's running. This kind of task performs the filtering process, and in the end has a Callback function that starts sending the CREQM to the filtered devices.

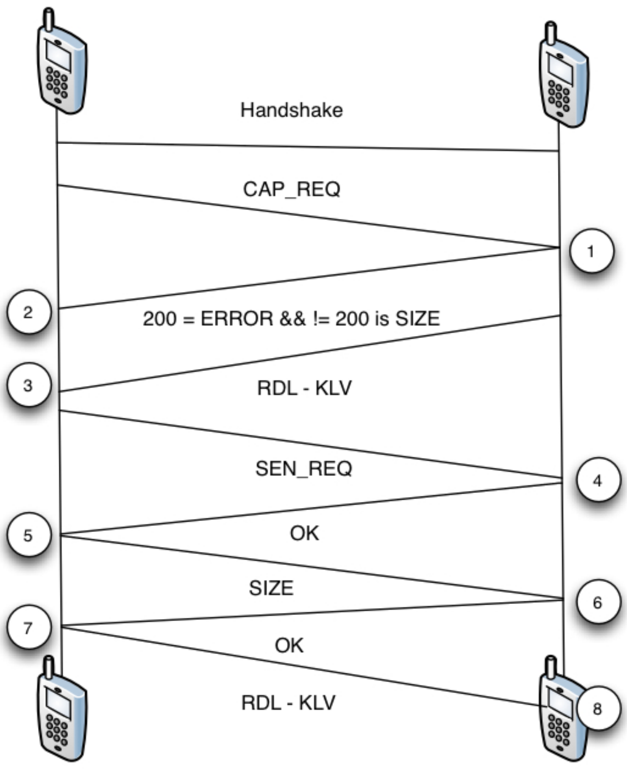


Figure 4.5: Sensor Request Process

The process has a few changes compared to the original MAR protocol described in the previous chapter. This changes were made to overcome bluetooth limitations and performance. The left side of the process is executed in the ApplicationCoreService and the left side is executed on the remote device, on the BluetoothService.

Refer to the checkpoints on the figure 4.5 marked with numbers from 1 to 8. These checkpoints mark the crucial points. On the checkpoint 1 the remote device after receiving the CAP_REQ character array replies with 200 if it's not available to reply, or replies with the size of the KLV RDL object. If he replies 200 the connection ends on both sides. If he replies with the size of the KLV RDL object, it sends the object right after.

In the original protocol the connection would end here and a new connection would be established to send the SREQM but that would add an unnecessary overhead. This overhead is associated with the handshake process and the locking strategy. So, with this implementation instead of closing the connection the connection is kept open and in 3 if the remote device has the requested sensor, the device will send a SEN_REQ character array. If the remote device replies with OK then the device will continue and send the RDL object in KLV format describing the requested resource. In 8 the request is added to the RequestManagerThread on the client side.

4.3 User Interface

This section will give an overview of the PhoneSensing's user interface. Figure C.1 in Appendix C shows the first options that are presented when the application is loaded. The Devices button the user start the discovery process and in the end check from the list the available devices in range.
The configure button takes the user to the Preferences Menu shown in figure C.2. This is where the user selects the new profile C.3 and configures the light and sound thresholds C.4. The user also has the privacy setting where he selects which sensors he wants active. The sensor, only works when it's active, but then it must also be shared. The ApplicationCoreService is launched by checking the Activated checkbox.

4.4 TCP Server

The TCP server was implemented using JAVA and the Hibernate framework. The hibernate mapping is done in a separate project, PhoneSensing-Core. The used database is a MySQL instance and the PhoneSensing-Core maps the database's tables in to JAVA objects using a typical DAO Factory design pattern. The connection is a JDBC connection using the default mysql JDBC driver.

Like said in chapter 3, the TCP server can handle multiple concurrent connected devices during the logging process. The server logs debug messages, message transfers, and batter consumption. It also keeps track of the online devices.

4.5 Web Server

The web server is also implemented using JAVA and uses the same Hibernate framework for mapping the tables. It also uses the Spring MVC framework in the web project design and dependency injection. The web server runs on a Glassfish3 server.

4.6 Conclusion

This chapter focused on the implementation of the architecture depicted in the previous chapter. The TCP Server and the Web Server were implemented as planed with no problems. The application and the protocol suffered a few changes due to the inherent software and hardware limitations. The most significant change was the protocol, that does not close the connection after receiving the CREQM, but instead waits for the reply. This was done to improve performance and reduce connection overhead. The bluetooth discoverable state was a problem. A solution was found, but still the user has to always give permission when the discoverable mode is extended by 300s.In a real world application wouldn't be acceptable. It's a flaw.
The Social Sensing aspect of the implementation ended up being quite simple compared with what was planned. This was mainly due to the complexity associated with context awareness algorithms using sensors. As seen in chapter 2 these algorithms involve quite a lot of research and development. Most of the development and research resources were deployed on the resource sharing protocol development.

5

Tests and Validation

Contents

5.1 Introduction

In chapter 3 we analyzed 3 protocols for sensor sharing in OSN and one was chosen according to a certain criteria.In chapter 4 it was explained how it was implemented. Part of this chapter will focus on going over the tests that lead to that decision to use that protocol. The other part will focus on analyzing the case study and comparing the theoretical results with the ones obtain using the implemented PhoneSensing application. Besides comparing, this chapter will aim at explaining the reasons behind the differences in the obtained results.

In order to study complex scenarios simulator tests were performed in the ONE simulator. The main goal in the protocol testing is to ascertain which of the three is the best all-rounder. It is not required for the protocol to be the best in all tests, but to be the one that suits the best in a vast majority of situations. The tests were prepared so that they represented most of the situations in daily use.

The case study is based on the first simulator test. There are obvious limitations when it comes to density and mobility tests, so it will be based on the simplest and lower node density test. Besides comparing the convergence times, the case study will also focus on testing performance. All results used were gathered through the logging process using the TCP server described in chapter 3 and 4.

5.2 Simulator

5.2.1 Testing Strategy

All tests will follow the same testing strategy. The goal is to keep the results as contextualized by isolating variables as much as possible. Hence, each test will have fixed configurations, and only the the protocol will change, so for each test scenario each protocol will have one run. The simulator yields deterministic results, so multiple runs for each test scenario aren't necessary. Test run times depend on the configuration values, and were adjusted to fit the longest running protocol.
There are a few important configuration settings to discuss.

5.2.1.A Number of nodes

This represents the number of global nodes in the test scenario. All nodes belong to the same group, meaning that they all run the same protocol, and use the same global settings. It is possible to configure different groups.

5.2.1.B Movement

The movement represents the node's movement pattern during the test. Only two patterns were used. One is the StaticMovement, where nodes don't move, and the other is ShortestPathMapBased-Movement. The ShortestPathMapBasedMovement makes the node move from point A to point B using the shortest path. This movement pattern was chosen because it yielded deterministic values, just like the static movement pattern.

5.2.1.C Transmit Speed

This configuration defines the radio interface configuration speeds. This value is constant through out all test scenarios.

5.2.1.D Update Interval

The simulations are discrete events. This update interval defines the interval between each event. This value is set depending on the movement pattern. Static movement patterns have shorts event intervals.

5.2.1.E Transmit Range

This value defines the maximum transmit range of the radio interface. This means that every node inside this range value will be accessible for transmission.

5.2.2 Test Scenarios

5.2.2.A Test Scenario 1

Test Scenario 1		
Configuration Values	Number of Nodes	10
	Movement	StaticMovement
	Transmit Speed	2Mbps
	Update Interval	0.01s
	Transmit Range	10m
	Run Time	100s

Table 5.1: Scenario 1 configuration values

Table 5.1 lists the test scenario's configuration values. This a low density test, 10 nodes, with no movement. This test was envisioned to be a benchmark and to guarantee that the protocols where

correctly developed. Being a low node density test made it easier to track every exchanged message an guaranteed that there were no incorrect message exchanges or strange behaviors. As a real world comparison this test could relate to a cafe scenario where a few people are sitting at tables reading or socializing as seen in figure 5.1.
This test will be a baseline to the case study test using the developed PhoneSensing application. Some changes will be made due to hardware restrictions and logistics.

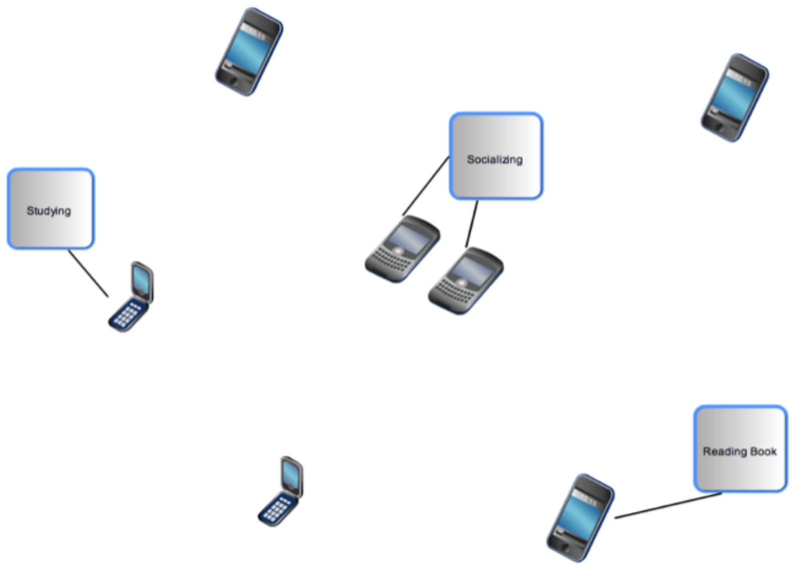


Figure 5.1: Test Scenario 1 Overview

Figure 5.2 shows each device's range. During the test each device will only be able to communicate with other device's in range, represented by the green circle.
Table 5.2 shows the node's capabilities and Table 5.3 lists the requests for each node. As shown in the tables only nodes 0 2 and 3 will have requests.

Nodes 0 and 2 are requesting the same sensor. Node 0 can only get that sensor from node 1 but node 2 can get it from node 1 and node 4. Node 3 requires a GPS, and he can only get it from node 1.

Results:

Figure 5.3 shows the convergence times in seconds for each protocol in this test scenario. The output value is the time since the protocol started for that request until the mode a reply arrives. In this figure we can see that for node 0 the convergence times were the same for all protocols. In Node

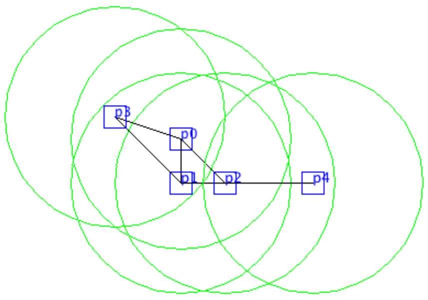


Figure 5.2: Device Range Overview Test 1

Node Capabilities	
Node	**Sensors**
0	LightSensor
1	GPS, 3Axis-Accelerometer,LightSensor
2	GPS, LightSensor
3	LightSensor
4	GPS, 3Axis-Accelerometer,LightSensor

Table 5.2: Node Capabilities Scenario 1

Node Requests	
Node	**Request**
0	3Axis-Accelerometer
2	3Axis-Accelerometer
3	GPS

Table 5.3: Node Requests Scenario 1

2 convergence times only changed with the SingleActiveRequest where it was slower. This happened because it only requested to one node, and it wasn't the node that would give the fastest reply. In the MultipleActiveRequest Node 2 sent the request to both available nodes and got a faster reply.

The biggest noticeable variation is with Node 3 where the convergence time using the SensorRequestOnly protocol was about 1/3 slower than the other protocols. This happens because sensor request messages are much bigger, compared to the CREQM hence, occupy more of the remote device's time.

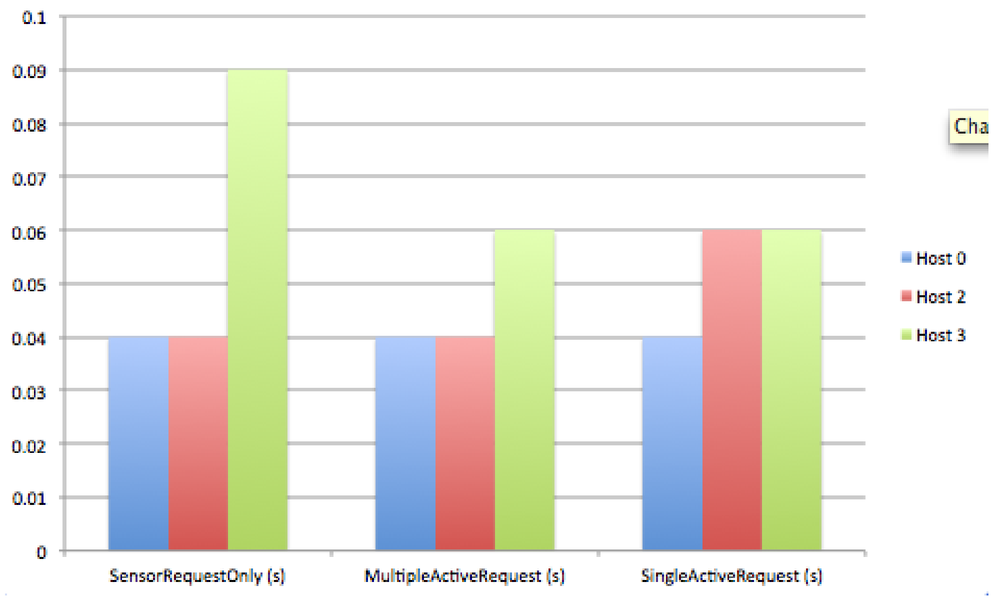


Figure 5.3: Response Times

On the other hand, the SensorRequestOnly protocol got much better results on the number of messages created 5.4 and total size transferred.

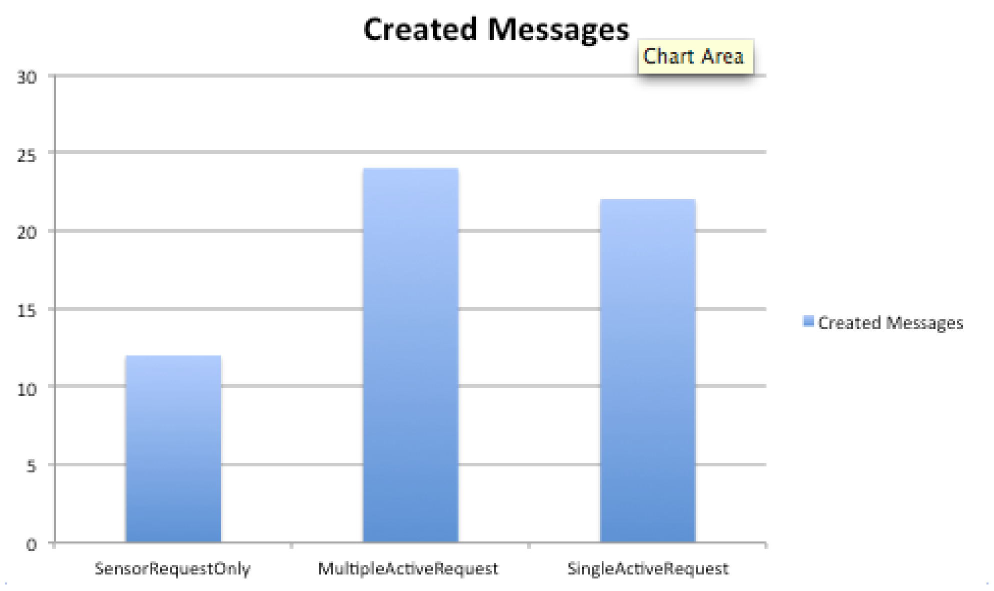


Figure 5.4: Created Messages

Figure 5.6 shows the number of ignored messages. The SensorRequestOnly will always have more ignored messages since it sends SREQM to all devices without knowing if they have or not the required sensor. SingleActiveRequest protocol has the same number of ignored messages has the Multiple-

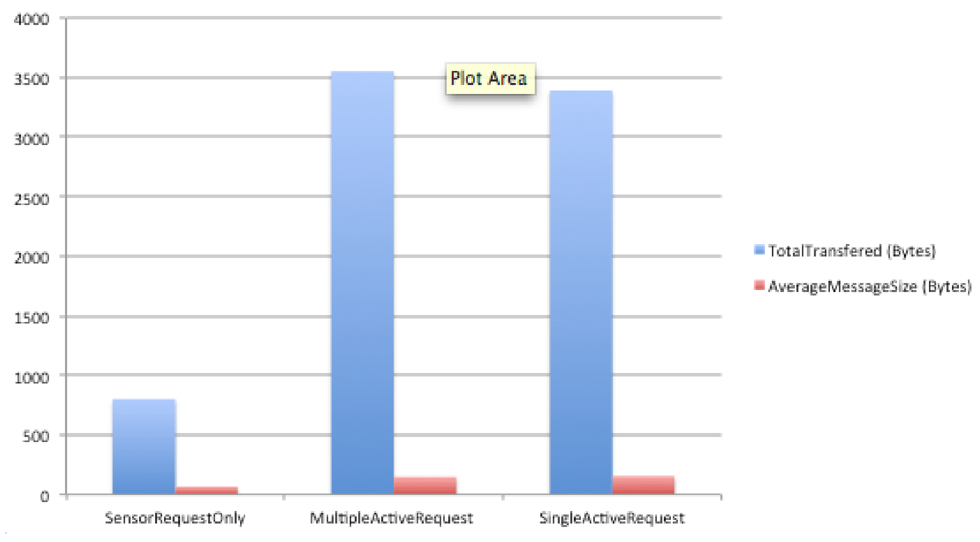


Figure 5.5: Total Transfered

ActiveRequest protocol. But one of them is very important. The red part represents a CRESM that's ingnored because the SREQM as already been sent. The problem here is that the protocol could be ignoring the device that would reply the fastest.

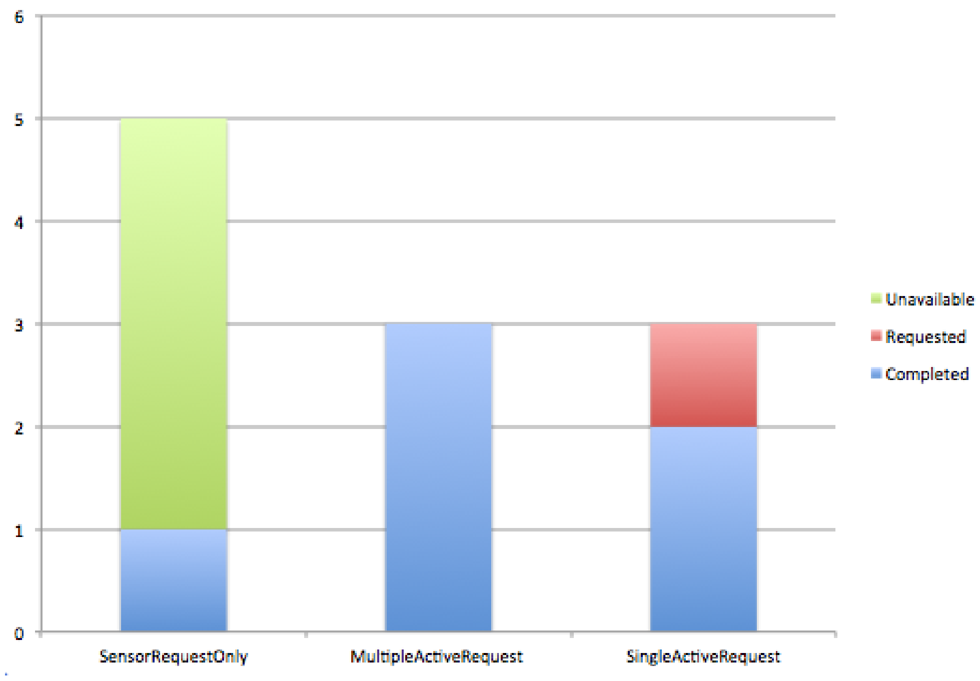


Figure 5.6: Ignored Messages

Conclusion: From the observed results, the SensorRequestOnly seems to be the best solution since it sends the less messages and less total bytes, but like it was said before, this is all about request completion times. In the next cases this difference will be a bit more accentuated.

5.2.2.B Test Scenario 2

Test Scenario 2		
Configuration Values	Number of Nodes	100
	Movement	ShortestPathMapBasedMovement
	Transmit Speed	2Mbps
	Update Interval	0.1s
	Transmit Range	10m
	Run Time	2000s

Table 5.4: Scenario 2 configuration values

This test, is very different from the previous one. As seen on table 5.4 it has more nodes, 100, and they aren't static. The'll go from the starting position to a pre-defined destination using the shortest path possible. This is means the simulation results are deterministic. This simulation is longer because as seen in figure 5.7 the nodes are very dispersed.

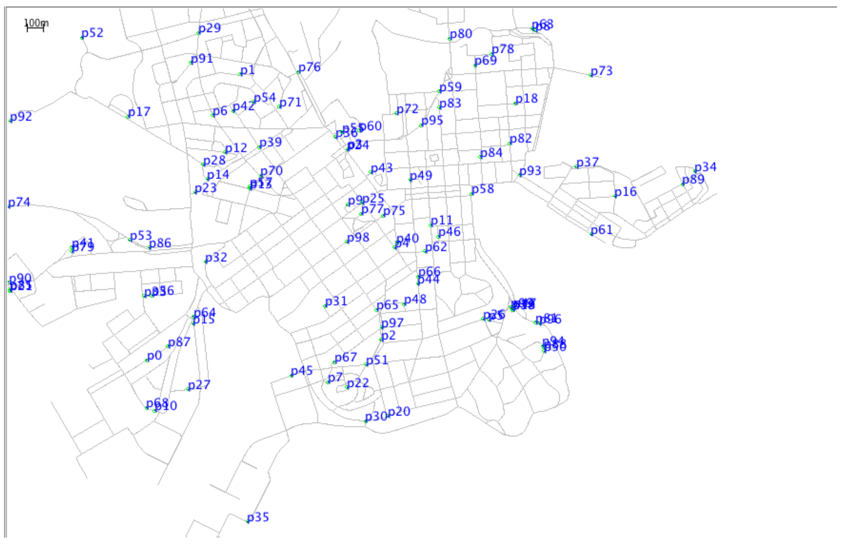


Figure 5.7: Device Range Overview Test 2

Results:

Again in this test it's seen that the response times in some cases are a lot longer on the SensorRequestOnly protocol when compared with the other two. This is evident by looking at figure 5.8. Figure 5.9 explains these results. The reason that in some requests the response time spiked is because with

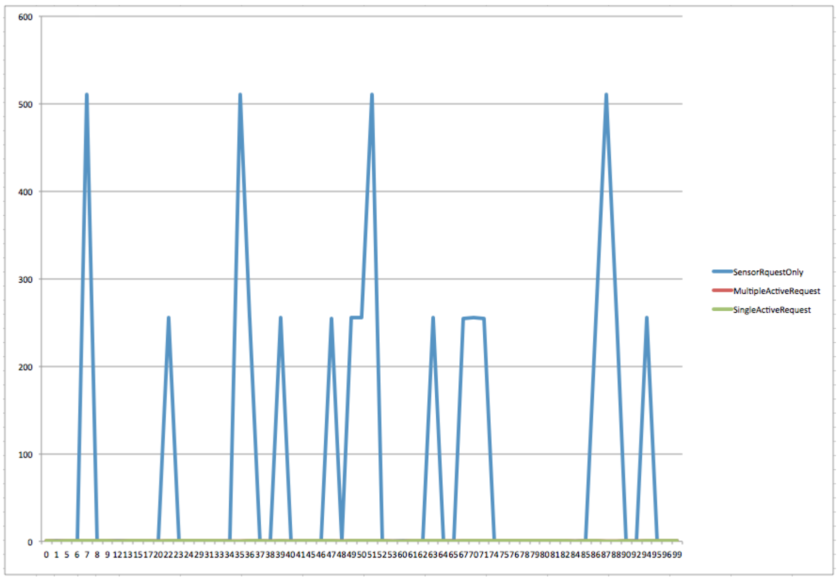


Figure 5.8: Response Times Test 2

mobility the SensorRequestOnly protocol has a lot of timeouts. Timeouts are requests that expired the TTL value after being sent. TTL value is 255 seconds.

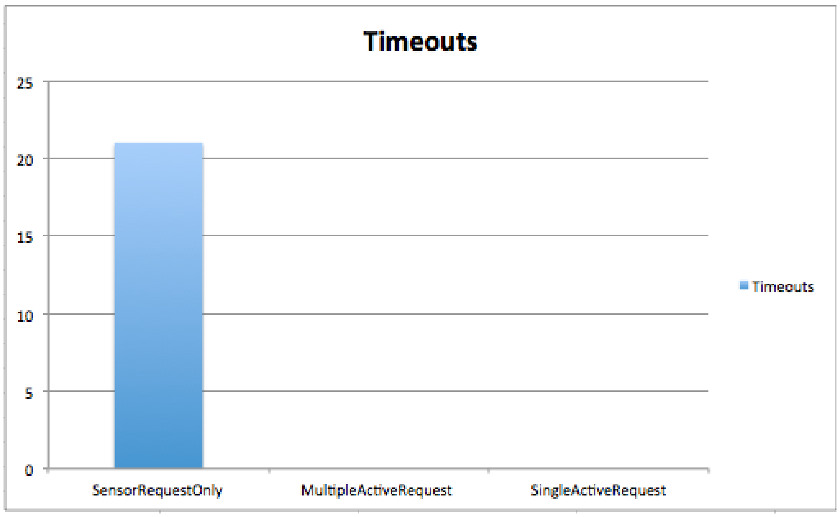


Figure 5.9: Timeouts Test 2

In Appendix D figures D.1 D.2 and D.3 show similar results to the ones analyzed in the previous test scenario. The SensorRequestOnly performed better when it came to the number of created messages and total bytes sent. But it was outperformed on the ignored messages, where it had much more ignored messages.

Conclusion: This test showed that in a mobility context the SensorRequestOnly performs much worst than the other two.

5.2.2.C Test Scenario 3

Test Scenario 3		
Configuration Values	Number of Nodes	1000
	Movement	ShortestPathMapBasedMovement
	Transmit Speed	2Mbps
	Update Interval	0.1s
	Transmit Range	10m
	Run Time	2000s

Table 5.5: Scenario 3 configuration values

This test scenario is exactly the same as the previous one except for the node density. This test has 10 times more nodes than the previous. In this test will focus on comparing the MultipleActiveRequest protocol and the SingleActiveRequest protocol that have been very similar in previous tests.

Results:

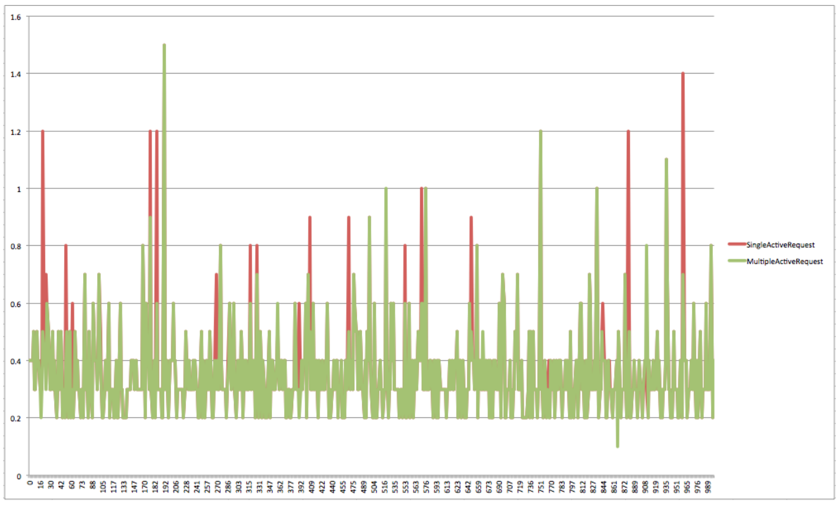


Figure 5.10: Response Times Test 3

Appendix D shows again that the SensorRequestOnly protocol has some timeouts that lead to high response times in some requests. As for the MultipleActiveRequest and the SingleActiveRequest protocols they both have similar response times, but the SingleActiveRequest, like in the first test scenario, in some requests doesn't perform quite as well as it's counterpart.

5.2.2.D Test Scenario 4

Test Scenario 4		
Configuration Values	Number of Nodes	200
	Movement	StaticMovement
	Transmit Speed	2Mbps
	Update Interval	0.01s
	Transmit Range	10m
	Run Time	300s

Table 5.6: Scenario 4 configuration values

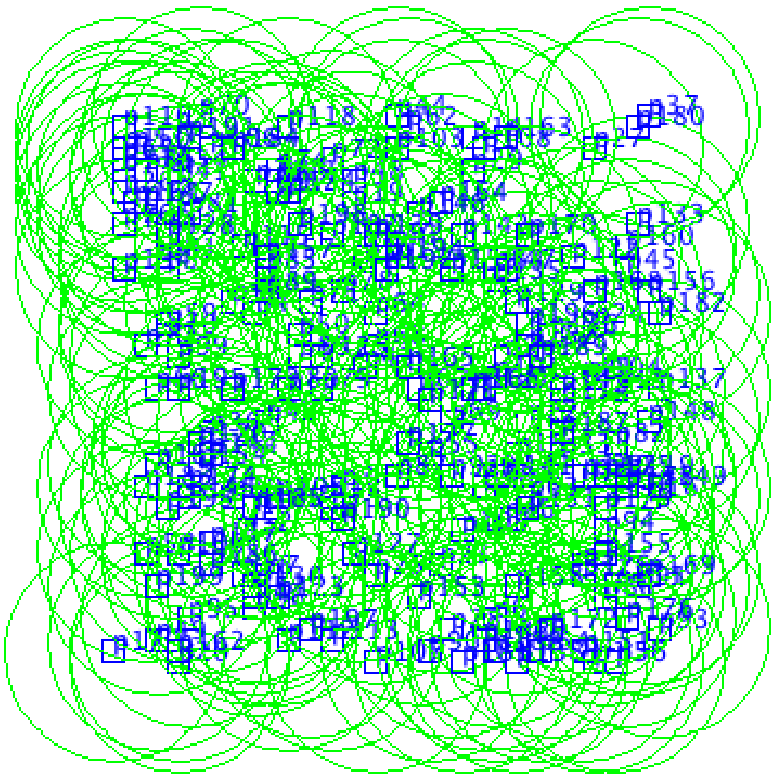


Figure 5.11: Device Range Overview Test 4

This test scenario is similar to the first, but it has a higher node density. It could be a real world equivalent to a busy shopping mall or a club. This test will expose the message exchange saturation in the medium since the devices will have many more nodes in their transmission range area.

Results:

This test obtained different results from the previous tests. Where as in previous tests the Single-

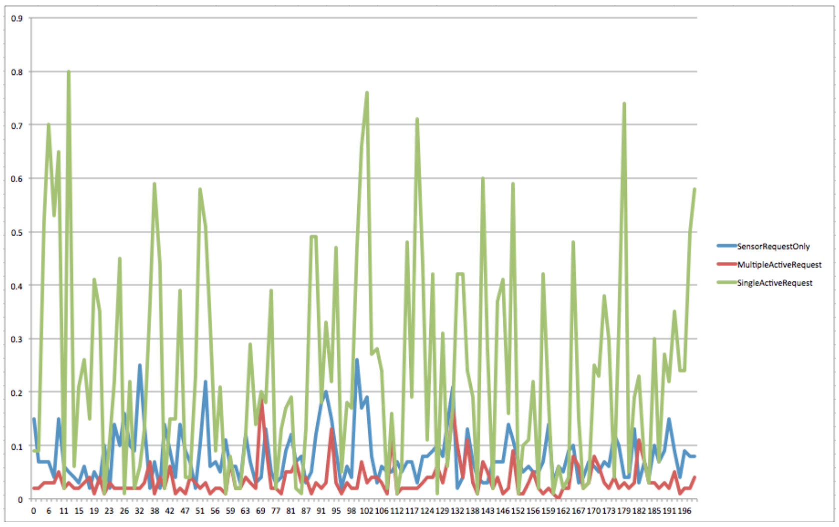


Figure 5.12: Response Times Test 4

ActiveRequest and the MultipleActiveRequest had the best results and the SensorRequestOnly was the worst, and here the SensorRequestOnly and the SingleActiveRequest switched places. Since mobility isn't a factor anymore, the SensorRequestOnly doesn't have timeouts and performs much better in general. The MultipleActiveRequest still remains the best performing of the lot.

In this test the number of ignored messages D.10 also became more even so as the number of messages created D.8 even though the SensorRequestOnly still creates much less messages than the others.

5.2.2.E Test Scenario 5

Test Scenario 5		
Configuration Values	Number of Nodes	1000
	Movement	StaticMovement
	Transmit Speed	2Mbps
	Update Interval	0.01s
	Transmit Range	10m
	Run Time	40s

Table 5.7: Scenario 5 configuration values

Test Scenario 5 is very similar to the previous one, the only change is the node density, it's 1000 nodes, in the same space. This would be a real world equivalent of a football stadium or some other event where there are large groups of people in the same place. Figure 5.13 shows a close up of some of the nodes.

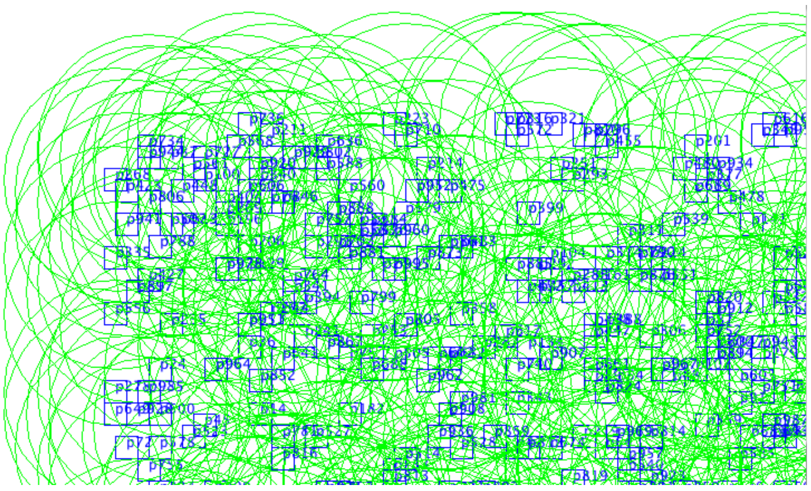


Figure 5.13: Device Range Overview Test 4

Results:

As expected the results here are the same. Increasing the number of nodes just increased the response time D.11 for each request and the number of messages created D.12 and ignored D.14.

5.2.3 Conclusion

In all 5 test scenarios only one of the protocol was able to perform better on every request time completion test, and that was the MultipleActiveRequest. It was outperformed by both other protocols when it came to the number of messages created and total bytes transferred. The SensorOnlyRequest performed very well on both those tests, coming first on all test scenarios. It didn't perform so well in the tests where the nodes weren't stationary, it was the only protocol to have request timeouts.
The decision on which protocol to chose would always be between the SensorRequestOnly and MultipleActiveRequest, since the SingleActiveRequest didn't out perform any protocol. As mentioned at the beginning of this chapter, the main choosing criteria is the request completion times, and hence the chosen protocol is the MultipleActiveRequest. On another network where time wasn't so critical, the SensorRequestOnly would be the better choice. Apart from the timeouts when mobility was a factor, it would still be a very good choice. Convergence times when compared to the MultipleActiveRequest weren't so different to make it an obvious decision.

5.3 Case Study

The main goal behind the case study tests is to analyze how the chosen protocol performs after it's implementation compared to the theoretical results ran in the simulator. This section will run a battery of tests similar to the first test scenario seen in the previous section. The results will be interpreted and discrepancies explained.

The scenarios reflect the underlining idea that the PhoneSensing represents. It represents an social sensing application that would help the user augment their activities by being aware for the surroundings ate letting the user know when they are not optima.
For example, would the user be at a coffee place working on he's dissertation the profile would be setup so that it would trigger an event when the lighting and/or conditions were not optimal. If such an event was triggered it would try to find a suitable place nearby using the GPS. If the GPS was not available it would use the sensor sharing framework to request it from any neighboring devices using the same applications. Had the application have connectivity with social networks it could also post that the user was about to change the location where he was studying.

The test scenarios are simple cases used to showcase the developed protocol and application. All device's will run the same instantiation of the PhoneSensing application. Each phone will have a different profile setup. The profiles, as mentioned in chapter 3 and 4, are used to harbor the different sensor configurations. These configurations will set the trigger thresholds on the different sensors. For example the studying profile will have a high light and noise threshold. The phone's will setup different profiles to simulate different activities.

5.3.1 Testing Strategy

None of the tests will involve device moment and all devices will be in range. This is, if a device runs the discovery process, it should yield in normal circumstances, every single device in the test except the device itself. The tests will be done in a single round, meaning that each device can only request one sensor.
Each test will incorporate different external variables that might affect the performance, and behavior of the protocol considering the real world limitations that the simulator did not take into consideration. It's important to understand where the simulator is real and where it isn't.

5.3.2 Test Scenarios

5.3.2.A Test Scenario 1

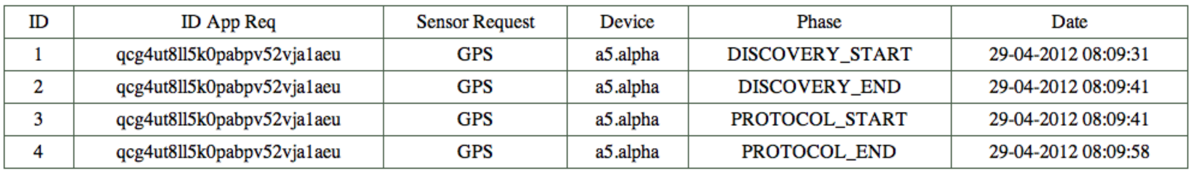
The first test scenario is the most basic possible. It's composed of two devices running the phone sensing application. One has a GPS sensor available to share, and the other has a studying profile available with a light threshold of 90%, so the lightest dimming of the lights will trigger an event to change location. That will require a GPS, which the device doesn't have.

Steps:

1. Create a profile for both devices
2. Edit profile in device with no GPS, set Light Threshold to 70%
3. Click the Light Sensor checkbox
4. Click the GPS checkbox and Activated checkbox on the other device
5. Click the Activated checkbox on the remaining device
6. Cover the phone so no light arrives at the sensor
7. Wait for the Change Location notification to arrive

Results:
Figure 5.14 shows the duration of each process since the device requested the GPS sensor so that he could chose a new location to go to. It took 27s from the discovery process until the device receiving the SRESM. The first important thing to point out, is that each Discovery Process will take exactly 10s, to return all the available devices. This time doesn't include the time it takes to cycle through every device and identify the ones that actually run the PhoneSensing application. At a first glance it takes much, much longer than the simulator to handle a single request. In the first test scenario described in the previous section, the longest request took 0.09s, and this was with other devices requesting at the exact same time. So the first conclusion that can be taken is that the simulator is far from yielding realist results. But that will be analyzed a bit more later.

Table 5.8 is an extract of the debug output generated by the devices during the test scenario. Has already mentioned, the process from the discovery until the SRESM took 27s. During the simulator tests the time logged in didn't count with the discovery process, because there wasn't any thing of the sort,



ID	ID App Req	Sensor Request	Device	Phase	Date
1	qcg4ut8ll5k0pabpv52vja1aeu	GPS	a5.alpha	DISCOVERY_START	29-04-2012 08:09:31
2	qcg4ut8ll5k0pabpv52vja1aeu	GPS	a5.alpha	DISCOVERY_END	29-04-2012 08:09:41
3	qcg4ut8ll5k0pabpv52vja1aeu	GPS	a5.alpha	PROTOCOL_START	29-04-2012 08:09:41
4	qcg4ut8ll5k0pabpv52vja1aeu	GPS	a5.alpha	PROTOCOL_END	29-04-2012 08:09:58

Figure 5.14: Protocol Execution Overview

the devices always knew at all times the surrounding devices. In this real world scenario, even excluding the discovery process, the message exchange takes much longer. In the simulator the longest request delay was 0.09 seconds. Other requests about halved that time.

ID	Log Type	Log Msg	Device	Date
3	DEBUG	Starting MultipleActiveRequest protocol	a5.alpha	29-04-2012 08:09:31
4	DEBUG	Starting Discovery Process	a5.alpha	29-04-2012 08:09:31
5	DEBUG	Discovery Process Finished	a5.alpha	29-04-2012 08:09:41
...	...	...	...	...
8	DEBUG	Sending CREQM to a5.beta	a5.alpha	29-04-2012 08:09:57
9	DEBUG	Received CREQM from a5.alpha	a5.beta	29-04-2012 08:09:58
10	DEBUG	Sending CRESM to a5.alpha	a5.beta	29-04-2012 08:09:58
11	DEBUG	Received CRESM from a5.beta	a5.alpha	29-04-2012 08:09:58
...	...	...	...	...
18	DEBUG	Sending SREQM to a5.beta	a5.alpha	29-04-2012 08:09:58
19	DEBUG	Received SREQM from a5.alpha	a5.beta	29-04-2012 08:09:58
24	DEBUG	Received SRESM from a5.beta	a5.alpha	29-04-2012 08:09:58

Table 5.8: Scenario 1 Debug Table

Looking at the table 5.8 another process that seems to take along time is the filtering process. Since the discovery process finishes until the first request is sent takes about 16s. This filtering process was one of the first challenges mentioned during the implementation process. This time will vary depending on how many bluetooth devices are around. During this test two more devices that weren't running the PhoneSensing application were returned in the discovery process. So adding up the discovery process and filtering process take about 96% of the duration of the process.

A big drawback to this is that during this 26s the device is unaccessible to answer to other requests because when it's in Discovering mode the bluetooth doesn't accept any other requests. During the filtering process it will be establishing connections with other devices. So in a high density node situation devices would struggle to establish any connection at all due to the out time that each of them has to endure during the discovering process and filtering process.

5.3.2.B Test Scenario 2

The previous test uncovered a few limitations of the implementation compared with what were the theoretical previsions. This second test will aim to expose even further such limitations. In this test there'll be 3 devices 5.15. One will have the GPS available for sharing (a5.alpha), and the other two (a5.charlie, a5.beta) in a point in time will need that available GPS sensor. Hence, the'll compete for it. For that to happen, the test requires that they're both triggered to request the GPS sensor at the same time. It's done by covering both devices at the same time so that they both go under the configured light threshold.

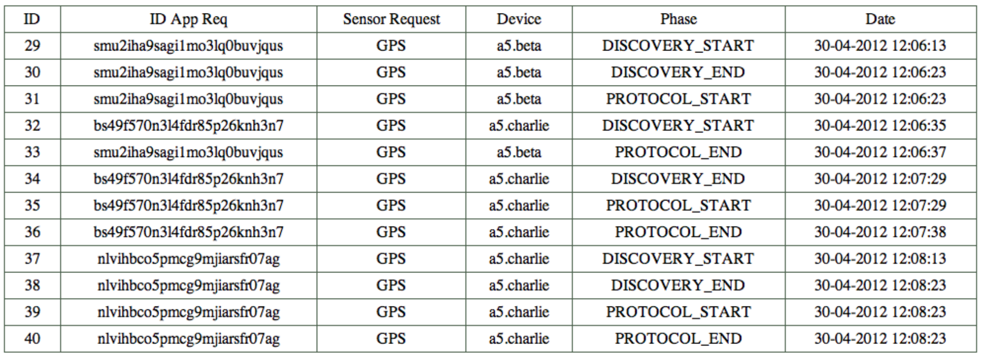
Figure 5.15: Test Scenario 2 Overview

Steps:

1. Create a profile for all 3 devices
2. Enable GPS sharing for device a5.alpha
3. Click the Light Sensor checkbox on the remaining devices
4. Click the Activated checkbox on on all 3 devices
5. Cover a5.charlie and a5.beta at the same time
6. Wait until the protocol converges

Results:

Figure 5.16 is a macro representation of the protocol execution. The import part to analyze here is the concurrent discovery process between the a5.beta and a5.charlie devices. In the previous test it was observed that the discovery process takes 10s. For a5.beta which is the first one to start the discovery process, it takes the 10s, but lets look at entry ID = 32. This shows that a5.charlie starts the discovery process at 12:06:37 and only finishes at 12:09:29. The discovery process took almost 1m to complete.



ID	ID App Req	Sensor Request	Device	Phase	Date
29	smu2iha9sagi1mo3lq0buvjqus	GPS	a5.beta	DISCOVERY_START	30-04-2012 12:06:13
30	smu2iha9sagi1mo3lq0buvjqus	GPS	a5.beta	DISCOVERY_END	30-04-2012 12:06:23
31	smu2iha9sagi1mo3lq0buvjqus	GPS	a5.beta	PROTOCOL_START	30-04-2012 12:06:23
32	bs49f570n3l4fdr85p26knh3n7	GPS	a5.charlie	DISCOVERY_START	30-04-2012 12:06:35
33	smu2iha9sagi1mo3lq0buvjqus	GPS	a5.beta	PROTOCOL_END	30-04-2012 12:06:37
34	bs49f570n3l4fdr85p26knh3n7	GPS	a5.charlie	DISCOVERY_END	30-04-2012 12:07:29
35	bs49f570n3l4fdr85p26knh3n7	GPS	a5.charlie	PROTOCOL_START	30-04-2012 12:07:29
36	bs49f570n3l4fdr85p26knh3n7	GPS	a5.charlie	PROTOCOL_END	30-04-2012 12:07:38
37	nlvihbco5pmcg9mjiarsfr07ag	GPS	a5.charlie	DISCOVERY_START	30-04-2012 12:08:13
38	nlvihbco5pmcg9mjiarsfr07ag	GPS	a5.charlie	DISCOVERY_END	30-04-2012 12:08:23
39	nlvihbco5pmcg9mjiarsfr07ag	GPS	a5.charlie	PROTOCOL_START	30-04-2012 12:08:23
40	nlvihbco5pmcg9mjiarsfr07ag	GPS	a5.charlie	PROTOCOL_END	30-04-2012 12:08:23

Figure 5.16: Request Tracking Overview

Figure 5.17 has two important clues for this. Examine ID 156. That debug entry shows that a5.alpha received a !END! message. That message is sent during the filtering process, to signal the remote device that the test is done, he's been identified as a PhoneSensing device. Now looking at message ID 182, we can see that only 11seconds later does a5.charlie receive the !END! message. This happens because this message was sent while a5.charlie was performing the discovery process. This conflict, made a5.charlie's discovery process last longer. Mean while a5.beta performed the protocol without any problems with the device a5.alpha. It took 2 seconds since the first CREQM was sent.

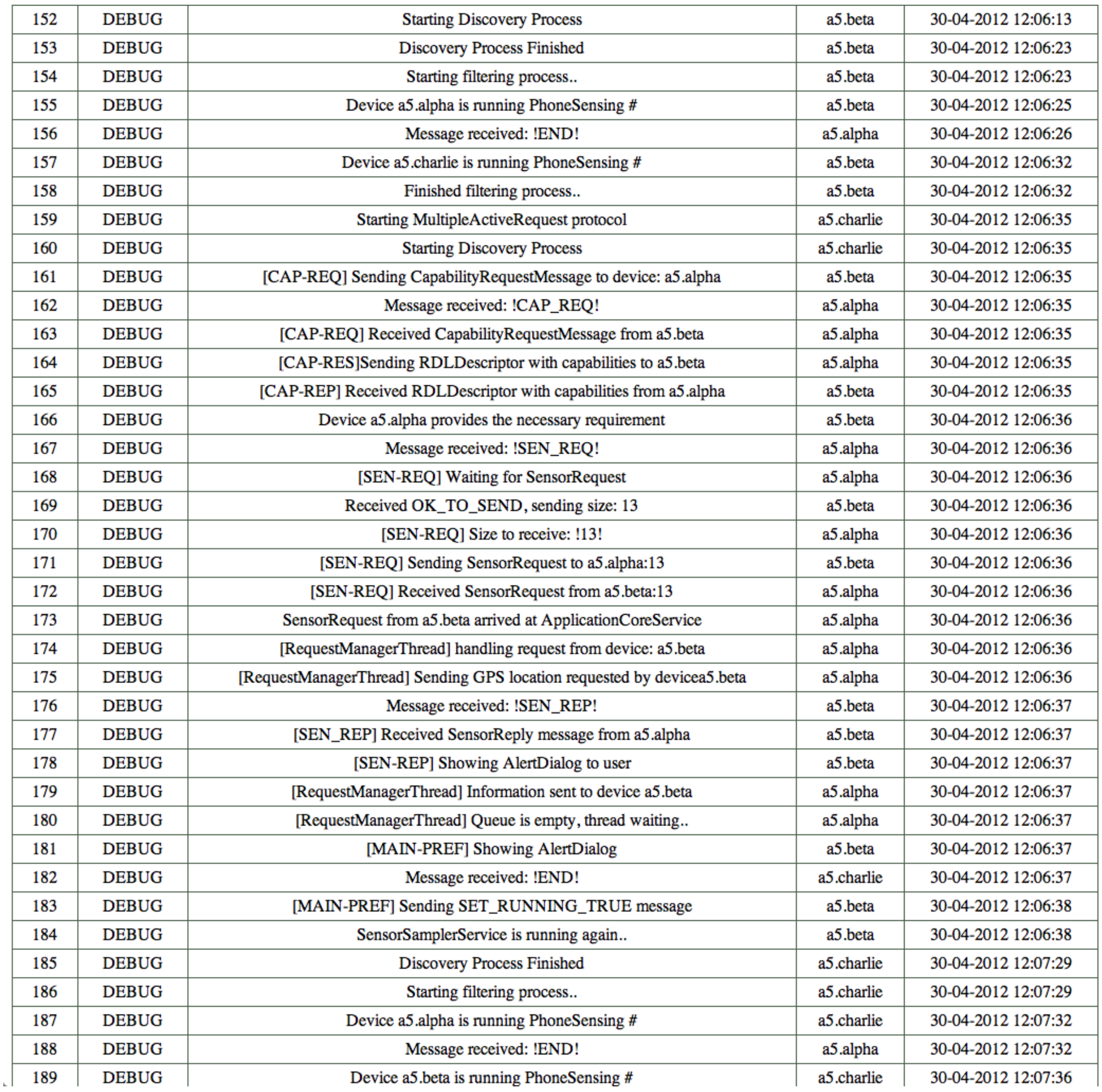
After that a5.charlie finished his discovery process he then filters the devices and is left with a5.alpha and a5.charlie. The protocol is executed normally taking 1 second from the moment the CREQM is sent.

Conclusion:

This test made even more evident the concurrency problems associated with the bluetooth communication. Even with the locking strategy employed the Discovery Process has conflict problems with the listening socket. This hints to the possibility of having to add the Discovery Process to the locking strategy. Apart from these problems, the protocol had the same performance as on the previous test. Taking 10s for the Discovery Process plus another 9s for the Filtering Process.

5.4 Conclusion

Time response times that were obtained during the simulation tests when compared with the real case scenarios can be considered optimistic. The simulator does not take a lot of factors into consideration. First, when the simulator runs the tests it always has knowledge of all the test scenario's devices. It knows where they are and within whom's range they are. Just this factor reduces drastically the protocol's convergence times. In the real world scenario a great part of the protocol execution time is spend in discovering the neighboring devices, this including the filtering of PhoneSensing devices.



152	DEBUG	Starting Discovery Process	a5.beta	30-04-2012 12:06:13
153	DEBUG	Discovery Process Finished	a5.beta	30-04-2012 12:06:23
154	DEBUG	Starting filtering process..	a5.beta	30-04-2012 12:06:23
155	DEBUG	Device a5.alpha is running PhoneSensing #	a5.beta	30-04-2012 12:06:25
156	DEBUG	Message received: !END!	a5.alpha	30-04-2012 12:06:26
157	DEBUG	Device a5.charlie is running PhoneSensing #	a5.beta	30-04-2012 12:06:32
158	DEBUG	Finished filtering process..	a5.beta	30-04-2012 12:06:32
159	DEBUG	Starting MultipleActiveRequest protocol	a5.charlie	30-04-2012 12:06:35
160	DEBUG	Starting Discovery Process	a5.charlie	30-04-2012 12:06:35
161	DEBUG	[CAP-REQ] Sending CapabilityRequestMessage to device: a5.alpha	a5.beta	30-04-2012 12:06:35
162	DEBUG	Message received: !CAP_REQ!	a5.alpha	30-04-2012 12:06:35
163	DEBUG	[CAP-REQ] Received CapabilityRequestMessage from a5.beta	a5.alpha	30-04-2012 12:06:35
164	DEBUG	[CAP-RES]Sending RDLDescriptor with capabilities to a5.beta	a5.alpha	30-04-2012 12:06:35
165	DEBUG	[CAP-REP] Received RDLDescriptor with capabilities from a5.alpha	a5.beta	30-04-2012 12:06:35
166	DEBUG	Device a5.alpha provides the necessary requirement	a5.beta	30-04-2012 12:06:36
167	DEBUG	Message received: !SEN_REQ!	a5.alpha	30-04-2012 12:06:36
168	DEBUG	[SEN-REQ] Waiting for SensorRequest	a5.alpha	30-04-2012 12:06:36
169	DEBUG	Received OK_TO_SEND, sending size: 13	a5.beta	30-04-2012 12:06:36
170	DEBUG	[SEN-REQ] Size to receive: !13!	a5.alpha	30-04-2012 12:06:36
171	DEBUG	[SEN-REQ] Sending SensorRequest to a5.alpha:13	a5.beta	30-04-2012 12:06:36
172	DEBUG	[SEN-REQ] Received SensorRequest from a5.beta:13	a5.alpha	30-04-2012 12:06:36
173	DEBUG	SensorRequest from a5.beta arrived at ApplicationCoreService	a5.alpha	30-04-2012 12:06:36
174	DEBUG	[RequestManagerThread] handling request from device: a5.beta	a5.alpha	30-04-2012 12:06:36
175	DEBUG	[RequestManagerThread] Sending GPS location requested by devicea5.beta	a5.alpha	30-04-2012 12:06:36
176	DEBUG	Message received: !SEN_REP!	a5.beta	30-04-2012 12:06:37
177	DEBUG	[SEN_REP] Received SensorReply message from a5.alpha	a5.beta	30-04-2012 12:06:37
178	DEBUG	[SEN-REP] Showing AlertDialog to user	a5.beta	30-04-2012 12:06:37
179	DEBUG	[RequestManagerThread] Information sent to device a5.beta	a5.alpha	30-04-2012 12:06:37
180	DEBUG	[RequestManagerThread] Queue is empty, thread waiting..	a5.alpha	30-04-2012 12:06:37
181	DEBUG	[MAIN-PREF] Showing AlertDialog	a5.beta	30-04-2012 12:06:37
182	DEBUG	Message received: !END!	a5.charlie	30-04-2012 12:06:37
183	DEBUG	[MAIN-PREF] Sending SET_RUNNING_TRUE message	a5.beta	30-04-2012 12:06:38
184	DEBUG	SensorSamplerService is running again..	a5.beta	30-04-2012 12:06:38
185	DEBUG	Discovery Process Finished	a5.charlie	30-04-2012 12:07:29
186	DEBUG	Starting filtering process..	a5.charlie	30-04-2012 12:07:29
187	DEBUG	Device a5.alpha is running PhoneSensing #	a5.charlie	30-04-2012 12:07:32
188	DEBUG	Message received: !END!	a5.alpha	30-04-2012 12:07:32
189	DEBUG	Device a5.beta is running PhoneSensing #	a5.charlie	30-04-2012 12:07:36

Figure 5.17: Test Scenario 2 Debug

Second the simulator doesn't considered the overhead associated with creating a connection. In the simulations the connections are established without any overhead. In the case study scenarios, there is an inherent overhead time associated with each bluetooth connection setup or teardown.

Third and finally, the remote logging process can't ignored. The setup and teardown of TCP connections with the server might slow down the process as well, since many debug messages are sent during the protocol execution.

6 Conclusion

Contents

6.1 Conclusion

This project proposed a decentralized solution for sensor sharing on a OSN. A decentralized approach allows for large-scale OSNs that don't require any extra infrastructural deployment. The motivation behind this is that by simply installing software on a mobile device, e.g. smart-phone, they become capable of sharing resources amongst them. With such a framework it is possible to deploy Social Sensing applications on mobile devices without them having to have a minimum number of required sensors. To prove that the concept works it was proposed to develop a Social Sensing application.

To achieve the proposed goals we started by creating a resource sharing protocol that would fit the project specifications. The protocol focused on fast convergence times to ensure that the protocol would work in scenarios where nodes were mobile and on small messages to ensure that the medium would not become saturated. Three different protocol's were thought up and put through various test scenarios using the ONE Simulator in order to select the one that guaranteed a high success rate in shorter time frames. To achieve small message sizes the protocol uses RDL to describe the available resources upon a device being queried. RDL is also used when requesting a sensor.
To test the protocol the Social Sensing application for Android smart-phones was developed. The message exchange was done using the smart-phones bluetooth radio interface in a P2P fashion guarantying the decentralized architecture requirement. The application worked as a proof of concept and also made it possible to test the protocol and compare the results with the previously ran simulations.

In the end the protocol was successfully implemented and the original idea of a decentralized OSN was maintained. It was shown that it is possible for such a concept to work on a real world situation, and that it does help support a social sensing application, or any other application that would benefit from resource sharing. The messages exchanged were small enough to be sent in acceptable times frames through the bluetooth interface. The simulation times diverged from the ones observed during the real case scenario tests, but not due to bad implementation but due to inherent limitations to the hardware in hand.

The main system limitations are associated with the data delivery process to the OSNs devices. Using bluetooth presented numerous limitations both associated with the bluetooth's implementation as a protocol but also with the bluetooth's implementation in the Android's operation system.

One very big drawback is that bluetooth doesn't support parallel connections so the application implementing the resource sharing protocol has to undergo a complex lock strategy. This implemented lock strategy has flaws so it doesn't have the expected performance. This is more obvious when the node density is higher and the number of concurrent connections is higher.

Another drawback is the discovery process. During the simulations this factor wasn't considered and hence the results were extremely optimistic. In the case study it became the obvious bottleneck to the whole solutions. The time needed to discover a device were very high.
Associated with the discovery process was the filtering process. It represents another big drawback since the discovery process might return devices that don't run the PhoneSensing application. This devices have to be filter, and the process is time consuming. In fact the discovery process added up with the filtering process consumed up to 96% of the protocol's time. After that process was done the message exchanges ranged from 1 to 2 seconds. More than the simulation results, yes, but still a very reasonable value considering.

If the aforementioned system limitations were duly solved and the proposed future work was done this decentralized resource sharing concept could be very useful for smart-phone developer's. It would allow for developers to work on an application without having to worry if the devices have the required sensors since they could just use available resources from neighboring peers. The protocol could also be ported into other devices such as cars.
Summing up, it was an interesting approach to an already studied topic. It solved some existing limitations on centralized architectures and proved it is a viable solution.

6.2 Future Work

The first improvement the system could undergo is the locking strategy for the bluetooth. It could be tweaked to yield better performance. It would also be interesting to understand what advantages or disadvantages would come as a result of employing the locking strategy during the discover process. The bluetooth implementation for the Android's OS does handle the connections during the discovery process. But disabling the listening socket all together during discovery might yield better performance.

As for the discovery process there are two things that should be looked into. First the possibility to broadcast messages via bluetooth, something that with the bluetooth driver wasn't possible. If that wasn't possible, it would be interesting to try and tweak the discovery query messages so that only devices running the PhoneSensing application would be able to interpret and reply. This would reduce the discovery and filtering process by half, since the filtering process would be done at the same time as the discovery process. To reduce the number of discovery processes application might implement the algorithm that would infer if the environment was volatile or not. In highly volatile environments where nodes come and go it would make sense to run the discovery process every time it was a outgoing request was queued. But, on stable environments that might not make sense since the discovery process would be yielding always the same devices. They could just be stored in memory and used again.

An interesting improvement would be to use different protocols in different scenarios. Since the preposed protocols seemed to work better in different scenarios, the application could have an algorithm that would select the most suitable protocol for the scenario in hand.

The choice to use bluetooth does not have to be rigid. Other approaches could be an option to solve the inherent problems associated with the bluetooth transfer and discovering process. In this dissertation bluetooth was chosen because of the device's limitation to establish ad-hoc WiFi connections. There weren't any other wireless P2P communication technologies available. But considering that devices in the future have the possibility to use WiFi for P2P connections it would be very interesting to have some work done in that path. Specially now that data connections via service providers tend to be more present on all legacy devices and with great upstream connections.

As for the social sensing aspect of the application a lot can be done. This example didn't go where this subject can go. Much can be done with sensorial inference.

Bibliography

[1] G. Pottie, “Wireless sensor networks,” in Information Theory Workshop, 1998, 1998, pp. 139 –140.

[2] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh, “Monitoring volcanic eruptions with a wireless sensor network,” in Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on, 2005, pp. 108 – 120.

[3] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, “Deploying a wireless sensor network on an active volcano,” Internet Computing, IEEE, vol. 10, no. 2, pp. 18 – 25, 2006.

[4] I. Stoianov, L. Nachman, S. Madden, and T. Tokmouline, “Pipeneta wireless sensor network for pipeline monitoring,” in Proceedings of the 6th international conference on Information processing in sensor networks. ACM, 2007, pp. 264–273.

[5] H. e. a. N. Lane, E. Miluzzo, “A survey of mobile phone sensing,” in IEEE Communications Magazine, 2010, pp. 140–150.

[6] B. P. Pál, “Sensorplanet: An open global research framework for mobile device centric wireless sensor networks,” in New York, 2007, pp. 2006–2008.

[7] E. Paulos, R. J. Honicky, U. C. Berkeley, and E. Goodman, “Sensing atmosphere,” in Interface, 2007, pp. 9–11.

[8] N. D. Lane, D. Lymberopoulos, F. Zhao, and A. T. Campbell, “Hapori: context-based local search for mobile phones using community behavioral modeling and similarity,” in Proceedings of the 12th ACM international conference on Ubiquitous computing, ser. Ubicomp ’10. New York, NY, USA: ACM, 2010, pp. 109–118.

[9] D. Peebles, H. Lu, N. D. Lane, T. Choudhury, and A. T. Campbell, “Community-Guided Learning : Exploiting Mobile Sensor Users to Model Human Behavior,” Computer, 2008.

[10] A. T. Campbell, S. B. Eisenman, K. Fodor, N. D. Lane, H. Lu, E. Miluzzo, M. Musolesi, R. A. Peterson, and X. Zheng, “Transforming the social networking experience with sensing presence

from mobile phones," in Proceedings of the 6th ACM conference on Embedded network sensor systems, ser. SenSys '08, New York, NY, USA, 2008, pp. 367–368.

[11] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell, "Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application," in Proceedings of the 6th ACM conference on Embedded network sensor systems, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 337–350.

[12] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell, "Soundsense: scalable sound sensing for people-centric applications on mobile phones," in Proceedings of the 7th international conference on Mobile systems, applications, and services, ser. MobiSys '09. New York, NY, USA: ACM, 2009, pp. 165–178.

[13] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, and G.-S. Ahn, "The rise of people-centric sensing," IEEE Internet Computing, vol. 12, pp. 12–21, 2008.

[14] S. Eisenman, N. Lane, and A. Campbell, "Techniques for improving opportunistic sensor networking performance," in Distributed Computing in Sensor Systems, ser. Lecture Notes in Computer Science, S. Nikoletseas, B. Chlebus, D. Johnson, and B. Krishnamachari, Eds. Springer Berlin / Heidelberg, 2008, vol. 5067, pp. 157–175.

[15] E. Miluzzo, N. D. Lane, and A. T. Campbell, "Virtual sensing range," in Proceedings of the 4th international conference on Embedded networked sensor systems, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 397–398.

[16] S. B. Eisenman, "People-Centric Mobile Sensing Networks," Ph.D. dissertation, Columbia University, 2008.

[17] C. Randell, C. Djiallis, and H. Muller, "Personal position measurement using dead reckoning," in Wearable Computers, 2003. Proceedings. Seventh IEEE International Symposium on, 2003, pp. 166–173.

[18] S. Eisenman, H. Lu, and A. Campbell, "Halo: Managing node rendezvous in opportunistic sensor networks," in Distributed Computing in Sensor Systems, ser. Lecture Notes in Computer Science, R. Rajaraman, T. Moscibroda, A. Dunkels, and A. Scaglione, Eds. Springer Berlin / Heidelberg, 2010, vol. 6131, pp. 273–287.

[19] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, and A. T. Campbell, "Bikenet: A mobile sensing system for cyclist experience mapping," ACM Trans. Sen. Netw., vol. 6, pp. 6:1–6:39, 2010.

[20] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in Proceedings of the 1st international conference on Embedded networked sensor systems, ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 126–137.

[21] M. Demmer, P. Levis, A. Joki, E. Brewer, and D. Culler, Tython: a dynamic simulation environment for sensor networks. Citeseer, 2005.

[22] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, and R. A. Peterson, "People-centric urban sensing," in Proceedings of the 2nd annual international workshop on Wireless internet, ser. WICON '06. New York, NY, USA: ACM, 2006.

[23] E. Miluzzo, C. T. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. T. Campbell, "Darwin phones: the evolution of sensing and inference on mobile phones," in Proceedings of the 8th international conference on Mobile systems, applications, and services, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 5–20.

[24] J. Lester, T. Choudhury, and G. Borriello, "A practical approach to recognizing physical activities," Pervasive Computing, pp. 1–16, 2006.

[25] A. Keränen, J. Ott, and T. Kärkkäinen, "The ONE Simulator for DTN Protocol Evaluation," in SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques. New York, NY, USA: ICST, 2009.

[26] A. C.Santos, L. D. Pedrosa, and R. M. Rocha, "RDL: A unified description language for network resources," RTCM seminar, March 2010.

A

Event Files

Contents

A.1 Capability Configuration File

The first number represents the node where it should be loaded into. The remaining values are the available sensors to that node.

```
#,9,10
0,3Axis-Accel,LightSensor
1,LightSensor
2,LightSensor,GPS
3,LightSensor,GPS,3Axis-Accel
4,3Axis-Accel,LightSensor
5,GPS,3Axis-Accel
6,LightSensor,GPS,3Axis-Accel
7,LightSensor,GPS,3Axis-Accel
8,LightSensor,GPS,3Axis-Accel
9,3Axis-Accel,LightSensor
```

A.2 Required Sensors Configuration File

Like the previous file, the first number representes the node id. The following value represents the sensor that the node requires.

```
#,9,0
0,GPS
1,GPS
2,3Axis-Accel
4,GPS
5,LightSensor
9,GPS
```

B

Event and Configuration File Generators

Contents

B.1 Configuration File

This python application generates the configuration for the ONE simulator.

```
'''
Created on May 10, 2011

@author: nadirturkman
'''
import random
import static_event_func

#static values
file_output_name = "../output/"
points = list()

#Query user for necessary values
nodes = int(input("number of nodes: "))
end_time = int(input("event end time: "))
protocol = input("protocol: ")
file_output_name += input("output file name: ")

print "output_file: " + file_output_name

#output file - top
file_output = open(file_output_name, "w")
file_output.write(static_event_func.file_top(end_time, nodes, protocol))

#output file - middle
i = 0

while i < nodes:
    pos = [0,0]
    pos[0] = 50 + random.randint(0, nodes/10)
    pos[1] = 50 + random.randint(0, nodes/10)
    if pos in points:
        continue
    else:
        points.append(pos)
        file_output.write(static_event_func.create_group(pos, i))
        i += 1

#output file - bottom
file_output.write(static_event_func.file_bottom(nodes))

file_output.close()

'''
Created on May 10, 2011

@author: nadirturkman
'''

def file_top(end_time, nodes, protocol):
    input = """
# PhoneSensing OSN Simulation Settings.
#
## Scenario settings
Scenario.name = PhoneSensing_OSN
Scenario.simulateConnections = true
Scenario.updateInterval = 0.01
# 43200s == 12h
Scenario.endTime = """ + str(end_time) + """

###########################################"

## Interface-specific settings:
# type : which interface class the interface belongs to
# For different types, the sub-parameters are interface-specific
# For SimpleBroadcastInterface, the parameters are:
# transmitSpeed : transmit speed of the interface (bytes per second)
# transmitRange : range of the interface (meters)

# Bluetooth interface for all nodes
btInterface.type = SimpleBroadcastInterface
# Transmit speed of 2 Mbps = 250kBps
btInterface.transmitSpeed = 250k
btInterface.transmitRange = 10

###########################################

PSRouter.PSAppId = ps.osn.PhoneSensingApplication
PSRouter.nrOfHosts = """ + str(nodes) + """
PSRouter.protocol = """ + protocol + """

###########################################

psApp.type = PhoneSensingApplication

###########################################

Scenario.nrofHostGroups = """ + str(nodes) + """
default=infinite
default=all

Group.nrofApplications = 1
Group.application1 = psApp
"""
    return input

def create_group(pos, id):
    input = """

# Settings for Group """ + str(id) + """ (Mobile Nodes)
Group""" + str(id+1) + """.movementModel = StationaryMovement
Group""" + str(id+1) + """.router = PhoneSensingRouter
```

```
Group""" + str(id+1) + """.bufferSize = 5M
Group""" + str(id+1) + """.waitTime = 0, 120
Group""" + str(id+1) + """.nrofInterfaces = 1
Group""" + str(id+1) + """.interface1 = btInterface
Group""" + str(id+1) + """.nodeLocation = """ + str(pos[0]) + ', ' + str(pos[1])  + """
Group""" + str(id+1) + """.msgTtl = 300
Group""" + str(id+1) + """.nrofHosts = 1
Group""" + str(id+1) + """.groupID = p
"""
    return input

def file_bottom(nodes):
    input = """

## Message creation parameters
# How many event generators
Events.nrof = 1
# Class of the first event generator
#Events1.class = OneToEachMessageGenerator
Events1.class = PhoneSensingMessageGenerator
# (following settings are specific for the MessageEventGenerator class)
# Creation interval in seconds (one new message every 25 to 35 seconds)
Events1.interval = 25,35
# Message sizes (500kB - 1MB)
Events1.size = 500k,1M
# range of message source/destination addresses
Events1.hosts = 0,""" + str(nodes-1) + """
Events1.tohosts = 0,""" + str(nodes-1) + """
# Message ID prefix
Events1.prefix = M
Events1.eventFile =
Events1.capabilitiesFile =

##########################################
```

```
## Movement model settings
# seed for movement models' pseudo random number generator (default = 0)
MovementModel.rngSeed = 1
# World's size for Movement Models without implicit size (width, height; meters)
MovementModel.worldSize = 4500, 3400
# How long time to move hosts in the world before real simulation
MovementModel.warmup = 1000

##########################################

## Map based movement -movement model specific settings
MapBasedMovement.nrofMapFiles = 4

MapBasedMovement.mapFile1 = data/roads.wkt
MapBasedMovement.mapFile2 = data/main_roads.wkt
MapBasedMovement.mapFile3 = data/pedestrian_paths.wkt
MapBasedMovement.mapFile4 = data/shops.wkt

##########################################

## Reports - all report names have to be valid report classes
# how many reports to load
Report.nrofReports = 8
# length of the warm up period (simulated seconds)
Report.warmup = 0
# default directory of reports (can be overridden per Report with output setting)
Report.reportDir = reports/
# Report classes to load
Report.report1 = MessageStatsReport
Report.report2 = PhoneSensingConnectionListener
Report.report3 = PhoneSensingMessageListener
Report.report4 = ContactTimesReport
Report.report5 = CreatedMessagesReport
Report.report6 = DeliveredMessagesReport
Report.report7 = EncountersVSUniqueEncountersReport
Report.report8 = PhoneSensingAplicationListener

##########################################

## Optimization settings -- these affect the speed of the simulation
## see World class for details.
Optimization.cellSizeMult = 5
Optimization.randomizeUpdateOrder = true

##########################################

## GUI settings
# GUI underlay image settings
GUI.UnderlayImage.fileName = data/helsinki_underlay.png
# Image offset in pixels (x, y)
GUI.UnderlayImage.offset = 64, 20
# Scaling factor for the image
GUI.UnderlayImage.scale = 4.75
# Image rotation (radians)
GUI.UnderlayImage.rotate = -0.015

# how many events to show in the log panel (default = 30)
GUI.EventLogPanel.nrofEvents = 100
# Regular Expression log filter (see Pattern-class from the Java API for RE-matching details)
#GUI.EventLogPanel.REfilter = .*p[1-9]<->p[1-9]$

"""
    return input
```

B.2 Capability File Generator

This python application generates the capability file for the event generator.

```
'''
Created on May 4, 2011
```

```
@author: nadirturkman
'''

import random
import capacity_func



#static values
caps = ["GPS","3Axis-Accel","LightSensor"]
file_name = "../output/"

#Query user for necessary
nodes = int(input("number of nodes: "))
file_name += input("output file name: ")

#Initialization line
init = '#,' + str(nodes-1) + ',0\n'

print "output_file: " + file_name

file = open(file_name, "w")
file.write(init)

i = 0
rand = 0
while i < nodes:
rand = random.randint(1,3)
j = 0
aux = list()
while j < rand:
c = random.choice(caps)
if c in aux:
continue
else:
aux.append(c)
```

```
j += 1

line = capacity_func.split_to_string(aux).rstrip(',')
file.write(str(i) + ',' + line + '\n')
i += 1

file.close()

'''
Created on May 4, 2011

@author: nadirturkman
'''

def split_to_string(aux):
    line = ''
    for x in aux:
        line += x + ','

    return line
```

B.3 Required Sensors File Generator

This python application generates the required sensor file for the event generator.

```
'''
Created on May 4, 2011

@author: nadirturkman
'''

import random
import event_func

#static values
caps = ["GPS","3Axis-Accel","LightSensor"]
```

```
file_output_name = "../output/"
file_input_name = "../../cap_gen/output/"

#Query user for necessary values
nodes = int(input("number of nodes: "))
file_output_name += input("output file name: ")
file_input_name += input("input file name: ")

#Initialization line
init = '#,' + str(nodes-1) + ',0\n'

print "output_file: " + file_output_name
print "input_file: " + file_input_name

#output file
file_output = open(file_output_name, "w")
file_output.write(init)

#input file
file_input = open(file_input_name, "r")

#readfirst line
line = file_input.readline()
if not line:
    print "error while reading file.."
    exit()
aux = line.split(',')
if aux[0] != '#':
    print "error, invalid input file"
if int(aux[1])+1 != nodes:
    print "error, input file has " + aux[1] + " nodes"

i = 0
while i < nodes:
    line = file_input.readline()
    if line:
        aux = line.split(',')
        if len(aux) < 4:
            file_output.write(str(i) + ',' + event_func.select_event(aux, caps) + '\n')
    i +=1

file_input.close()
file_output.close()

'''
Created on May 4, 2011

@author: nadirturkman
'''
import random

def select_event(aux, caps):
    i = 0
    c = list()

    while i < len(aux):
        c.append(aux[i])
        i += 1

    while 1:
        event = random.choice(caps)
        if event not in c:
            return event
```

C

User Interface Screenshots

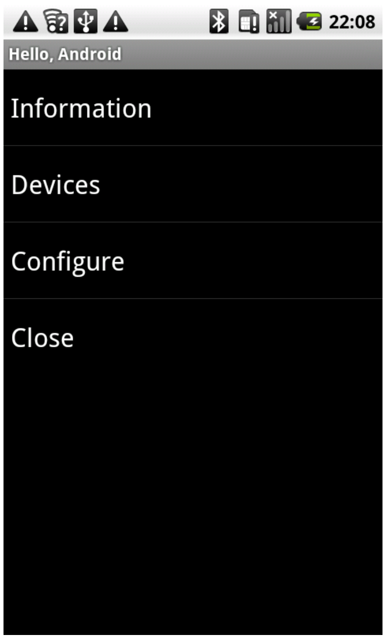
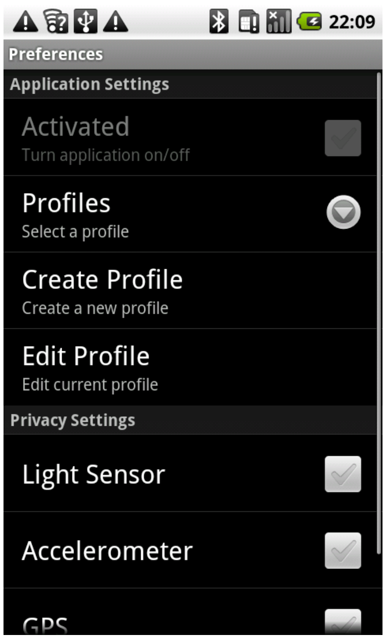


Figure C.1: Main Menu


Preferences

Application Settings

Activated
Turn application on/off

Profiles
Select a profile

Create Profile
Create a new profile

Edit Profile
Edit current profile

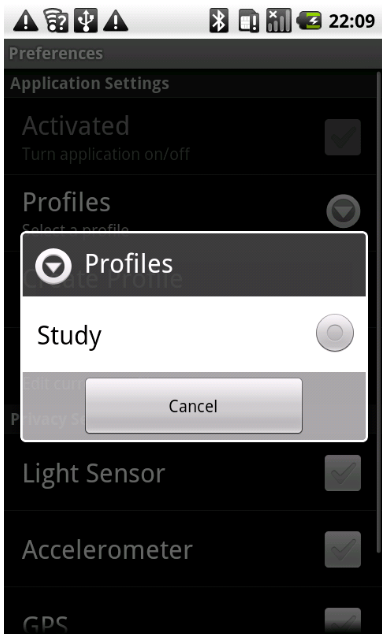
Privacy Settings

Light Sensor

Accelerometer

GPS

Figure C.2: Preferences Menu


22:09

Preferences

Application Settings

Activated

Turn application on/off

Profiles

Profiles

Study

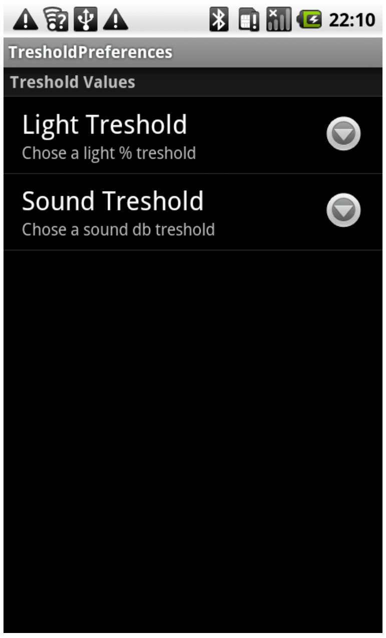
Cancel

Light Sensor

Accelerometer

GPS

Figure C.3: Select Profile


22:10

TresholdPreferences

Treshold Values

Light Treshold

Chose a light % treshold

Sound Treshold

Chose a sound db treshold

Figure C.4: Edit Profile

D

Result Tables

Contents

D.1 Simulator

D.1.1 Test Scenario 2

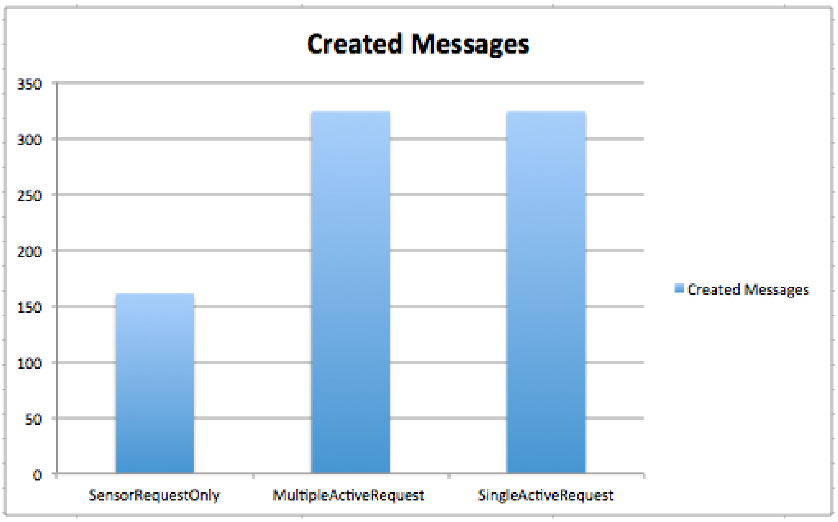


Figure D.1: Created Messages Test 2

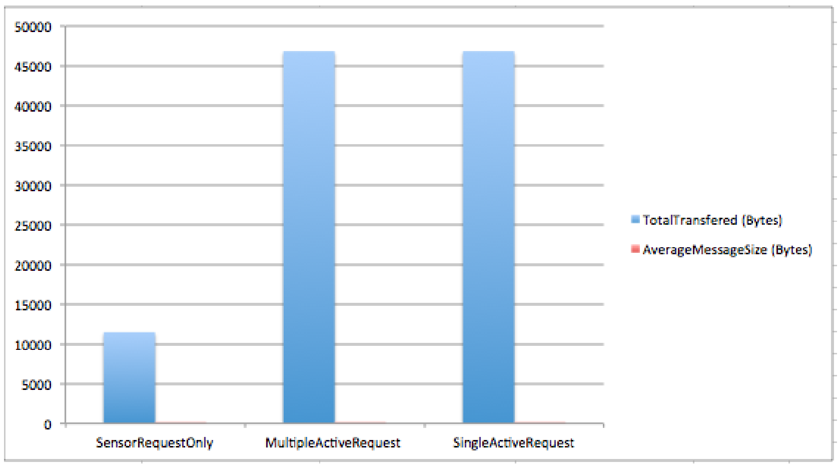


Figure D.2: Total Sent Test 2

D.1.2 Test Scenario 3

D.1.3 Test Scenario 4

D.1.4 Test Scenario 5

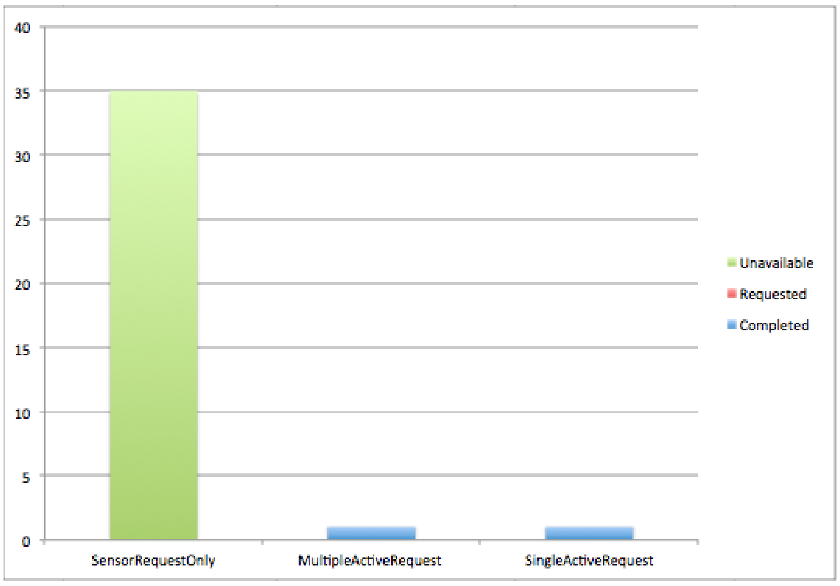


Figure D.3: Ignored Messages Test 2

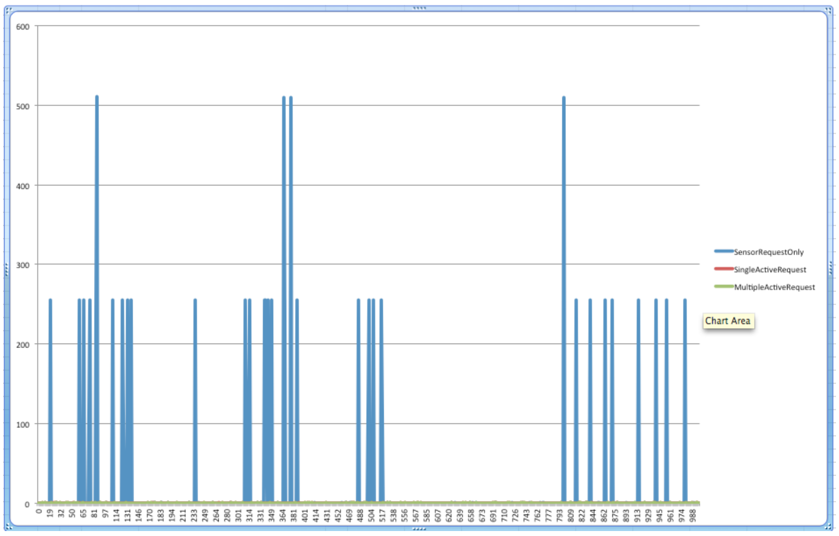


Figure D.4: SOR Response Times Test 3

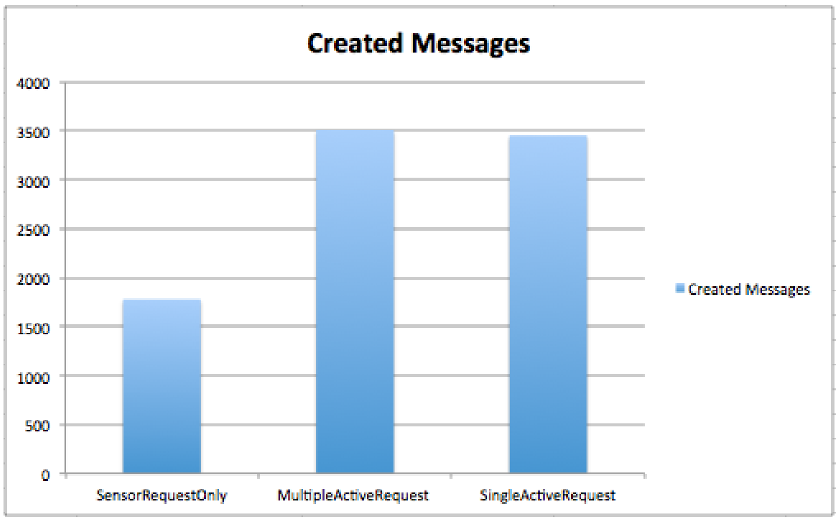


Figure D.5: Created Messages Test 3

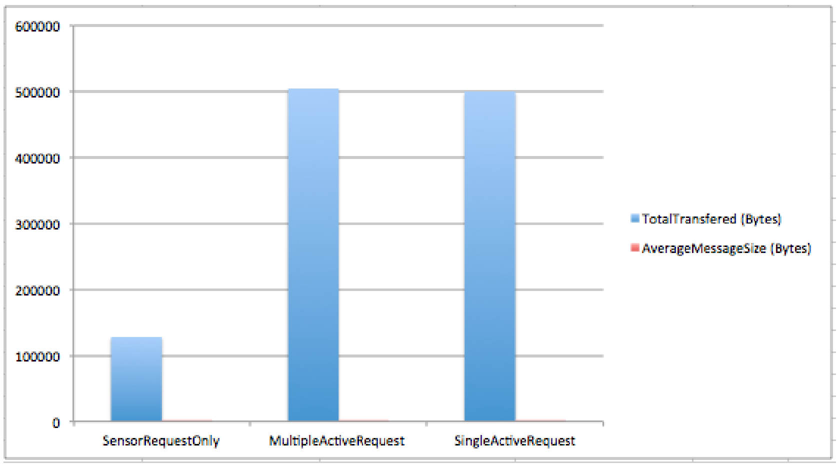


Figure D.6: Total Sent Test 3

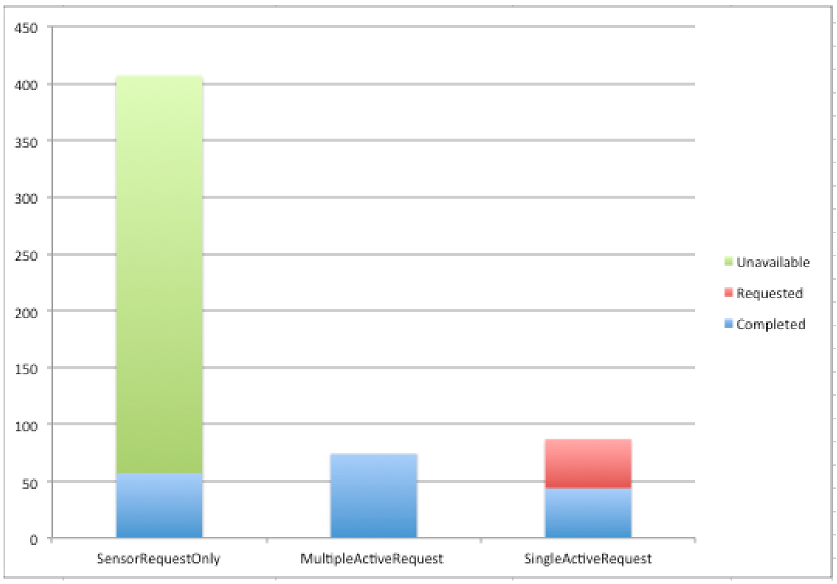


Figure D.7: Ignored Messages Test 3

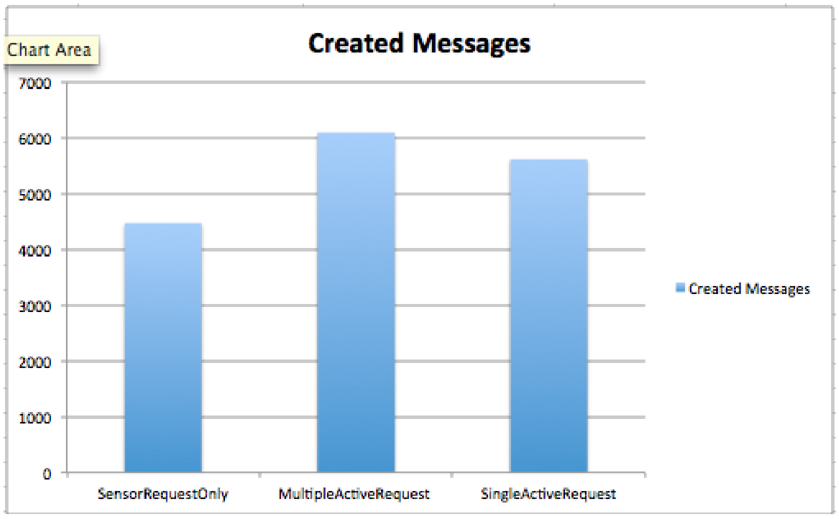


Figure D.8: Created Messages Test 4

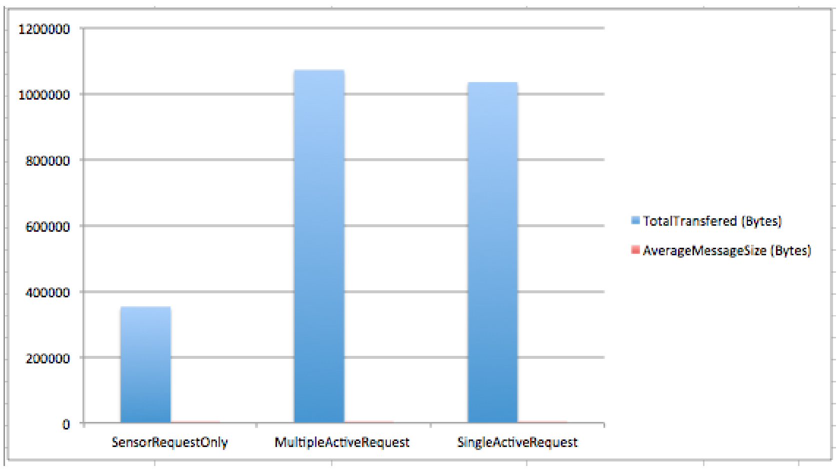


Figure D.9: Total Sent Test 4

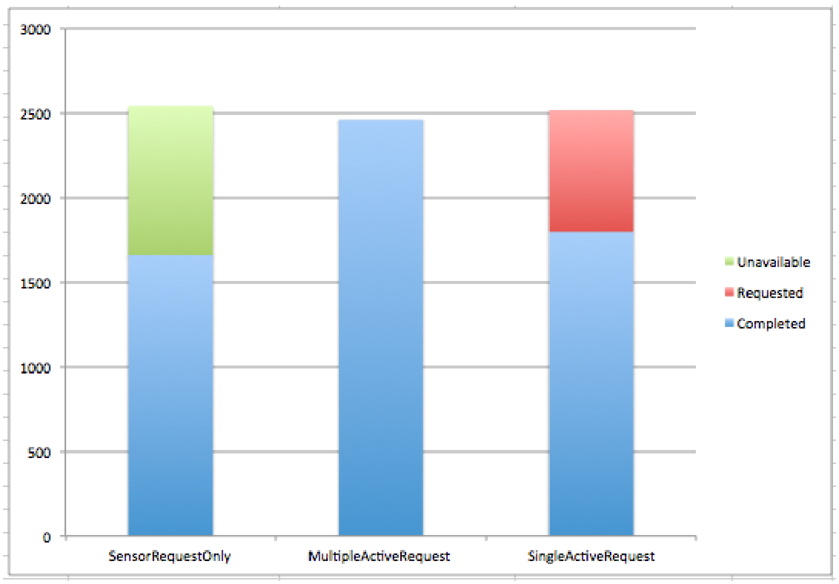


Figure D.10: Ignored Messages Test 4

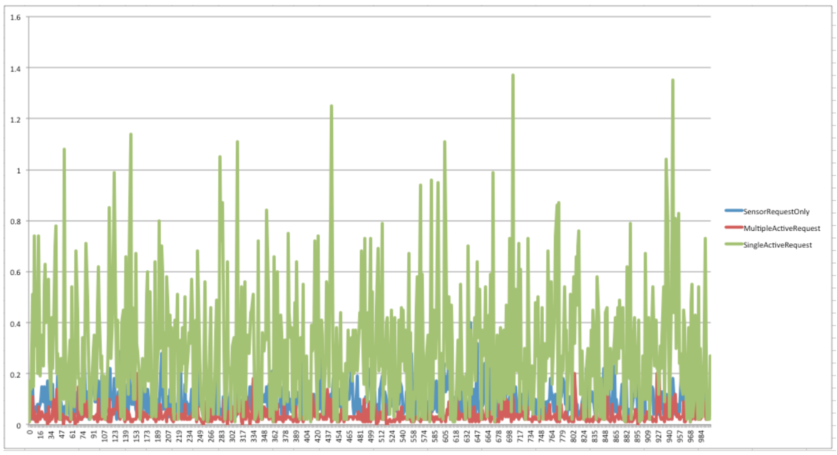


Figure D.11: Response Times Test 5

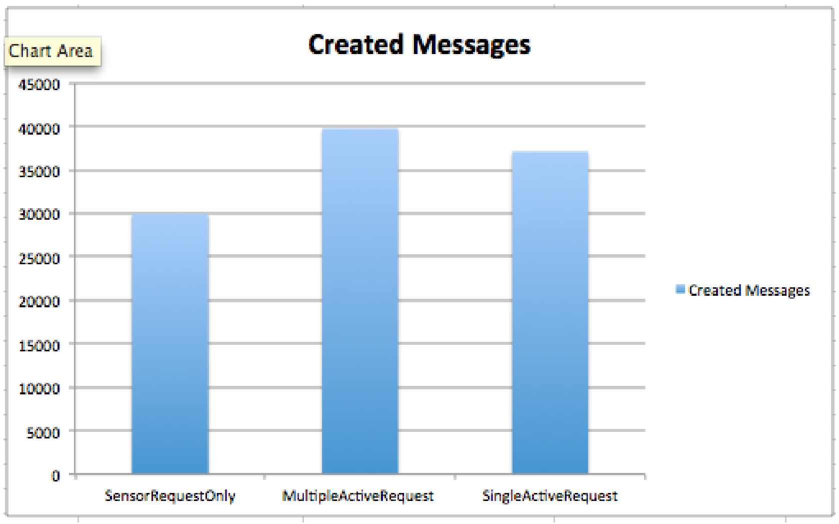


Figure D.12: Created Messages Test 5

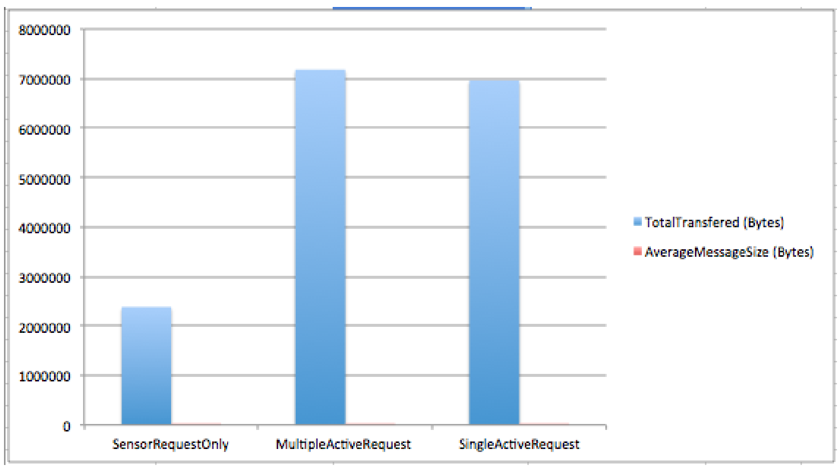


Figure D.13: Total Sent Test 5

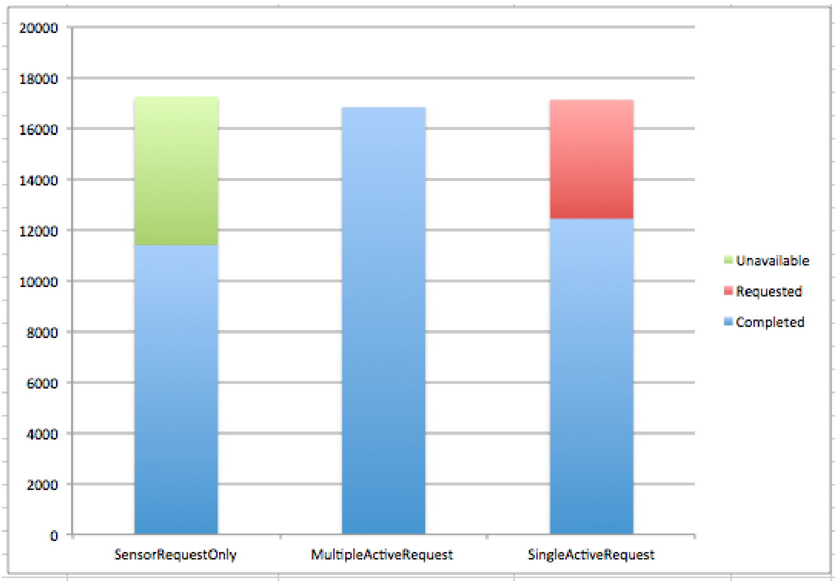


Figure D.14: Ignored Messages Test 5