Eindhoven University of Technology

MASTER

Service discovery, monitoring and management in smart spaces composed of low capacity devices

Uzun, H.Ö.

Award date:
2009

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

Master's Thesis

SERVICE DISCOVERY, MONITORING AND MANAGEMENT IN SMART SPACES COMPOSED OF LOW CAPACITY DEVICES

by

H. Önder Uzun

Assessment Committee:

Dr. T. Özçelebi (Supervisor)
Prof. Dr. J.J. Lukkien
Prof. Dr. P.M.E. De Bra

Abstract

Contemporary developments in mobile and distributed computing combined with miniaturization and wireless networking technologies enable vision of smart spaces, i.e. seamless, user-oriented technology embedded within a living space which dynamically adapts to its environment through automated contextual information gathering and processing.

The latest smart space solutions presume the smart space network is composed of high capacity nodes (e.g. laptop computers, mobile phones, PDAs). Their layered architectural design gives rise to abstraction and modularization, which eases application development, however, results in extra energy costs and messaging overheads. This solution approach is inapplicable to smart spaces of low capacity nodes (e.g. wireless sensors) due to resource constraints, e.g. energy, computational power, network bandwidth limitations.

This master's thesis introduces an architectural solution approach for smart spaces composed of very low capacity nodes. Two main targets of this work are *i)* to develop a smart space architecture with basic capabilities to discover, monitor and manage nodes, and their services and resources; and *ii)* to provide lightweightness, for low capacity nodes to participate. For this purpose, a system architecture for sensor networks is used as the baseline architecture and is extended to the proposed lightweight scalable smart space architecture. Discovery, monitoring and management tasks are realized by means of a central node called the Resource Manager (RM). Conclusively, the new architecture is deployed on physical wireless sensor nodes and the footprint of the implementation on the nodes is obtained.

Experimental results show that the proposed smart space architecture is indeed lightweight and suitable for very low capacity nodes.

Table of Contents

1. Introduction 7

1.1 Overview of Smart Spaces *8*

1.1.1 Pervasive Computing *8*

1.1.2 Context-Aware Computing *9*

1.1.3 Smart Spaces *10*

1.2 Concerns/Requirements of Smart Spaces *11*

1.3 Wireless Sensor Networks *15*

1.3.1 Relevance to This Master Project *17*

2. Problem Description *19*

2.1 Architecture for Smart Spaces *19*

2.1.1 Need for a Dedicated Architecture *19*

2.1.2 Fundamental Aspects of a Smart Space Architecture *20*

2.2 Lightweightness *21*

2.2.1 Need for a Lightweight Solution *21*

2.2.2 Low Capacity Devices *21*

2.3 Related Work *22*

2.4 Objective: An Architectural Solution for Low Capacity Devices *23*

3. Proposed Solution *25*

3.1 OSAS Framework *25*

3.1.1 Programming Model *26*

3.1.2 System Architecture *27*

3.1.3 Network Programming Language *28*

3.1.4 Messaging and Communication *30*
3.1.5 OSAS Toolchain *31*

3.2 Implementation Alternatives *32*
3.2.1 Our Choice: Centralized Solution *33*

3.3 Service Discovery *34*
3.3.1 Discovery of Nodes: The Heartbeat Protocol *35*
3.3.2 Proposed Enhancements to Heartbeat Protocol *37*
3.3.3. Discovery of Services and Resources *40*

3.4 Service/Resource Monitoring *41*
3.4.1 Handling of RM Node Entries *42*
3.4.2 Reporting from Nodes Perspective *43*
3.4.3 Reporting Intervals & Triggers *44*

3.5 Management *46*
3.5.1 Service Mapping *46*
3.5.2 Path Based Routing *48*

3.6 Putting It Altogether *51*

4. Experimental Results *53*
4.1 Simulation Environment *53*
4.1.1 Example Scenario *54*

4.2 Deployment on Physical Nodes *58*

5. Conclusions and Future Work *61*

Appendix *63*

References *65*

1. Introduction

Recent developments in miniaturization, wireless networking and sensor technologies combined with profound contemporary research reveal the vision of future physical environments, i.e. *smart spaces*. The smart space concept foresees complete merger of technology and living spaces.

Smart spaces idea is inspired by *pervasive computing*[1] and *especially context aware computing*[2] and extends them, envisioning seamless integration of smart devices into users' everyday life where the central focus is shifted from devices and technicalities to user and user's tasks. Overall, a smart space is a technology embedded physical space, exhibiting pervasive and context-aware capabilities. Pervasiveness refers to seamless integration of technology. Context-awareness refers to providing services improved by automated contextual information gathering and processing.

Nodes within the smart space network communicate and collaborate in order to provide advanced services to its users. The system is entirely user oriented which aims minimal user interaction and maximal autonomous operation. An important feature is that a smart space network is highly dynamic, i.e. physical nodes can get in and out of the smart space arbitrarily. Therefore, the system should automatically adapt to changes in network composition and topology as well as context.

At the node level, such a system requires the functionality for the nodes to expose their service and resource capabilities so that the smart space can utilize the node for enriched user experience. These requirements can only be fulfilled by an architectural design that supports easy and fast service/resource discovery in addition to real-time service quality management.

The smart spaces vision has attracted many researchers as well as industry and a number of solutions have been proposed upon extensive research and development. The majority of contemporary solutions in the field presume the smart space network is composed of rather high capacity nodes (e.g. laptop computers), proposing abstraction through decoupling of the system layers of designed architecture. This solution, however, results in extra energy costs and messaging overheads and would be inapplicable to low capacity nodes with limited resources, e.g. energy, memory, computational power, network bandwidth.

In this project, our aim is to facilitate low capacity nodes, such as wireless sensor nodes, to form smart spaces. Wireless sensor nodes are small (ranging from a shoe size to a small coin) devices with very limited capacity of resources such as battery power, processing power and memory. These spatially distributed autonomous devices use sensors to cooperatively monitor physical or environmental conditions such as temperature and vibration. An example would be to collect temperature readings from the environment and raise an alarm if a possible fire incident is deduced. A wireless sensor network (WSN) is a ubiquitous distributed network of wireless sensor nodes, whose aim is mainly to collect contextual data and communicate it.

In this work, we extend an existing WSN architectural solution to realize a smart space architecture that is composed of very low capacity nodes. We develop lightweight methods for

---

[1] Please refer to Section *1.1.1 Pervasive Computing*
[2] Please refer to Section *1.1.2 Context-Aware Computing*

automatic discovery of smart space components and monitoring their services and resources at run-time. This functionality is extended to map service requests to capable devices by selecting a provider and a subscriber most suitable for satisfying the request. The main contribution of this work could be summarized as to leverage a low capacity network into a smart space network by implementing and realizing the fundamental functionalities and aspects such as service discovery and monitoring, automatic operation, autonomous adaptation, and service mapping.

This chapter follows with an overview of smart spaces concept, giving highlights of the foundational ideas and discussion of technical concerns and requirements of smart spaces. A brief introduction to WSN is also presented. In Chapter 2, the problem description is put forward stating requirements and objectives, followed by a discussion of relevant literature research on the specific problem domain. In Chapter 3, our proposed solution architecture is presented. This includes selection of a WSN framework to build on, the design decisions taken and the solution approaches in every detail. In Chapter 4, experimental results are presented and discussed. A scenario is described and the system is tested both on a simulator, which is improved for smart space testing, and on deployed physical nodes. Moreover, the footprint values (i.e. the measured numbers on the usage of hardware resources) of the added extension to the underlying WSN architecture is put forward, along with communication load. Finally, in Chapter 5, conclusions are drawn along with a discussion of future work.

1.1 Overview of Smart Spaces

Smart spaces idea is built upon two prior foundational concepts: *pervasive computing* and *context-aware* computing. In order to conceive the smart space notion accurately, firstly, these concepts are introduced in the following two sub-sections. A more detailed description of the smart spaces concept is given afterwards.

1.1.1 Pervasive Computing

Nowadays, computing is moving towards ubiquitous environments, in which devices, software agents, and services are all expected to integrate and cooperate in support of human users [1]. This paradigm of the 21st century computing foresees systems to anticipate needs, negotiate for services, act on users' behalf and deliver services in an anywhere, any-time fashion.

In this vein, pervasive computing basically refers to seamless integration of devices into users' everyday life. This phenomenon is projected to be the new era of mobile and distributed systems in the contemporary world where the systems will be unnoticeable and the central focus will be shifted from devices and technicalities to user and user's tasks. The ultimate aim is to serve users not by alienating them from technological devices, but in such ways they will find refreshing and pleasant.

Pervasive computing differs from conventional notion of computing in three main aspects [2]:

- A device is a portal into an application/data space, not a repository of custom software managed by the user.
- An application is a means by which a user performs a task, not a piece of software that is written to exploit a device's capabilities.
- The computing environment is the user's information-enhanced physical surroundings, not a virtual space that exists to store and run software.

There are two distinct earlier steps of evolution enabling pervasive computing: distributed computing and mobile computing [3] [4], as depicted in *Figure 1*. *Distributed computing* opened the way for computers to connect, form a network and seamlessly access each other remotely. For example, the World Wide Web has been a huge success by introducing ubiquitous information and communications infrastructure. *Mobile computing* brought cellular technology into users' everyday life. The contribution of mobile computing is to provide services to user independent of their location and their end devices to access the services. A user with a SIM card can make calls from any cell phone, and this separation carries the importance of accessing the services above the end devices.

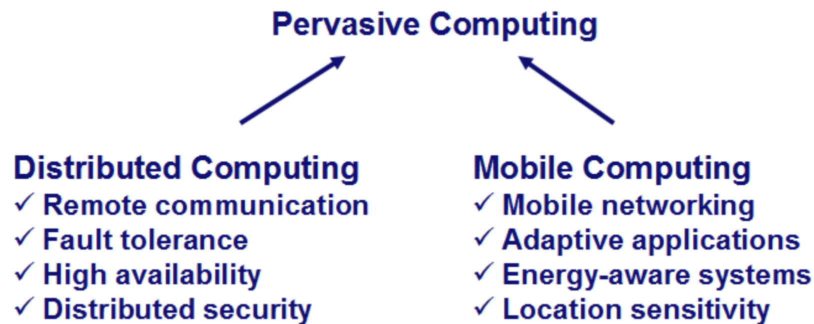

Figure 1: Prior concepts to pervasive computing

It seems that we can expect a future where the user attention and time is the most valuable resource rather than devices or underlying technologies. It won't be the humans to enter the virtual world of computers to accomplish tasks but the computers to come and integrate into physical world of humans. Moreover, this transformation will make the technology disappear as well as being, or appearing to be, every-time and everywhere. Improving on the experience and principles of mobile and distributed computing, pervasive computing becomes the vision of future.

Latest proceedings in miniaturization, wireless networking, and sensor technologies as well as profound contemporary research enable computer systems to exhibit pervasive behavior. However, the picture is not impressive, yet, in terms of market acceptance and commercial availability and the realizations are mostly prototypes so far. The reason is that designing, implementing and maintaining such systems is tremendously challenging, let alone financial infeasibilities. The technical challenges are mainly to achieve scalability, heterogeneity, integration, seamlessness and context-awareness [3].

1.1.2 Context-Aware Computing

Context awareness has always been a relevant issue for humans as to comprehend the world and act accordingly. Whether consciously or not, humans are able to pick situational information and use it to understand daily situations and adapt their behavior and they are very good at it [5]. Most importantly, humans use contextual information to enrich their communication which brings success and increased bandwidth of human-to-human communication. For example, when we're holding a conversation in a noisy place, we talk louder so that the other person can hear. But when we're in a meeting, we whisper so as not to disturb other people.

Contrary to human species, today's traditional interactive computing services are far behind capturing and using contextual information. An example regarding today's technology could be

the login services that notify the computers of user identity and thus the provided services are specific to the user. Nonetheless, future systems are projected to be context-aware in a way that would revolutionize our interaction with them. As an example, the mobile phone of the future would detect if a user is at a meeting by sensing sound levels, gathering location information and using the user's digital agenda; and it will rather not ring and give a notification after the meeting.

A context-aware system is defined as a system that adapts to its location of use, the collection of nearby people and objects, as well as the changes to these entities over time [6]. In this respect, the word "context" can be defined as any information that can be used to characterize the situation of an entity, where the entity could be a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves [7].

Context-aware computing is actually a field of pervasive computing; being specialized on context-awareness, yet following pervasive computing goals and principles. The key feature of context-aware systems is to be able to provide automated services through contextual information gathering and processing. As contextual information is gathered and processed, the systems will adapt and behave accordingly. This is the point where technology starts to feel no longer 'dumb' but rather 'smart'.

1.1.3 Smart Spaces

A smart space is basically a defined physical space (e.g. home, office) with pervasive and context-aware capabilities and thus providing superior services to its users. A smart space consists of distributed technological devices that sense, communicate and collaborate, where the computational intelligence is integrated into users' physical environment [7].

The core of "smartness" idea is to manage and reason about contextual information and services/resources at hand and adapt behavior automatically. Sensed contextual information can be used to automatically control the physical space tailored to user preferences. Moreover, the contextual data can be stored and used for future reasoning and for many other advanced functionalities.

Services provided by smart spaces should possess a certain degree of intelligence (i.e. reasoning, learning) and a social ability (i.e. communication, co-operation). We can define four key activities to support these characteristics [8]:

- Learn from previous events and adapt behavior
- Be aware of other services, applications and resources
- Advertise functionality to other services
- Interact with other services, applications and resources

The smart space could also be perceived as an extensive pool of resources and services. The devices can come into or get out of the smart space and they should be automatically recognized and their services seized. As well, the absence/unavailability of the devices should be recognized and the services should be dropped. A smart management mechanism monitors and manages the resources in a service oriented approach where the overall system aims to fulfill user requests in an optimized way. A requested service will be provided as long as the resources and service capabilities are present, regardless of individual devices and underlying technologies.

The requirements, challenges and other technical concerns regarding smart spaces concept are discussed in the next section.

1.2 Concerns/Requirements of Smart Spaces

In this section, we state relevant concerns and requirements specific for smart spaces domain. Not all of the issues addressed in this section are within the scope of this work. However, it is necessary to clarify the concept of smart spaces at the outset, by proposing a substantial, well-defined set of requirements and considerations.

Automatic Operation

The system to be designed will perform its duty with minimal or no human intervention. There is an option for the users to define so called "user defined" policies or preferences and the system would adapt. However, except for voluntary user involvement, the system has to carry out its operations at a fully-automatic level. This kind of operation is also known as *invisibility*.

Autonomous Real Time Adaptation: Dynamicity

Autonomous real time adaptation refers to run-time flexibility and self-adjustment of the system with respect to ever changing conditions. Such swift adaptation is also referred as *dynamicity*.

In a smart space, the availability of resources and services may change during a typical period of system operation. Such a system should assume that any arbitrary distributed element can exhibit unexpected behavior or go out of order, even while the devices are providing service. So, the system has to be sensitive to tolerating faults in run-time. Dynamicity also implies the system to support reasoning about absent nodes and incoming nodes. When a device arrives or leaves the space, the space should handle this physical transition in terms of registering/dropping the device and its services and adapting its current behavior to the new conditions.

Figure 2 illustrates a node coming in and getting out of the smart space. In the first image, Node A is out of the smart space. In the second image, Node A has moved into the smart space zone and now it is discovered and registered into the system. The system, now, can utilize Node A for providing services to its users. In the third image, Node A has left the smart space zone and now it is unregistered from the system. All this adaptation process is done automatically.

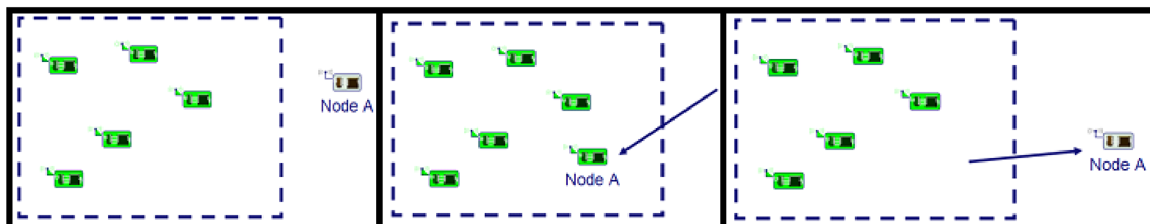

Figure 2: Dynamic Discovery of Devices

Service Mapping and Arbitration

A smart space system should be able to respond to requests from its user(s), as to provide appropriate services to match the requests. This management operation basically consists of finding appropriate service provider devices, selecting one (or more depending on the request) and executing the requested service. In this vein, there are two important concepts: service mapping and arbitration.

Service mapping is the process of deciding which resource handles which request. With proper service mapping, a system decides to which device a specific type of service request will be directed. This, of course, depends on the availability of devices in the system and their services and resources. Figure 3 illustrates a simple example, where the system receives a *Service B* request and among available devices, trivially, the node which is capable of providing that service is selected to fulfill the request. The request is then redirected to that node to be executed.

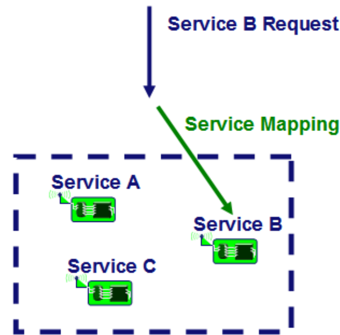

Figure 3: Service Mapping Example

Service arbitration is the appropriate allocation of resources and optimal use of them. At a minimum, arbitration mechanism should make sure that the devices are not used beyond their capacity. Moreover, it ideally should make optimal or nearly optimal use of scarce resources via appropriate allocation. Figure 4 illustrates an example case: the system receives a *Service A* request. Among available nodes, there are two that can provide the service. The system selects the appropriate one by applying a cost function. The cost function is a function which takes the resource usage and availability of the nodes into account, e.g. the average energy usage of a node. In this case, the system decides which node to select according to their energy usage since they are battery powered. One node uses 3 units of energy to provide the service and the other one uses 5 units comparably. Then, the arbitration mechanism selects the one with the least energy cost.

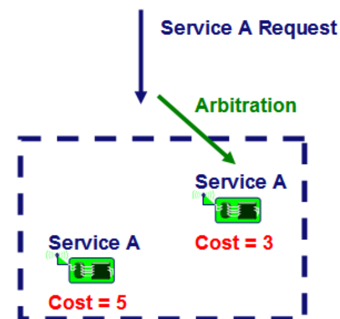

Figure 4: Arbitration Example

Complex Services and Service Chaining

In order to facilitate full leverage of the services the system can offer, it is necessary to support provisioning of not only simple individual services, but also more *complex* (or aggregated) *services*. For instance, the system should be able to automatically distribute a video call service to an available beamer, microphone and speakers and provide this complex service through more than one device (see Figure 5).

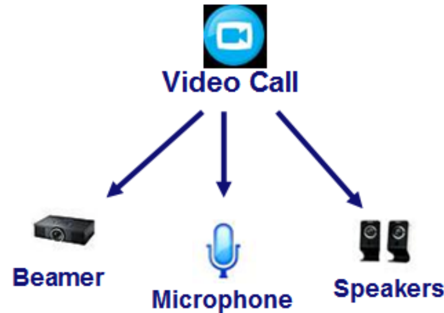

Figure 5: Complex Service Example

A powerful concept with different service capabilities among devices is *service chaining*. Chaining of services is relevant when a device requires some other service to provide its particular service(s). Figure 6 demonstrates an example: A smart space consists of a laptop computer, a portable music player and loudspeakers and the user wants to listen to his/her favorite music from the music player on the loudspeakers. However, loudspeakers cannot play that specific format of music data. In this case, the laptop comes into play by converting the music data into compatible format so that it could be played on the loudspeakers. The challenge with this behavior is that it should be automatic and invisible to the user as though the user just plays his music on the loudspeakers by selecting it on the music player.

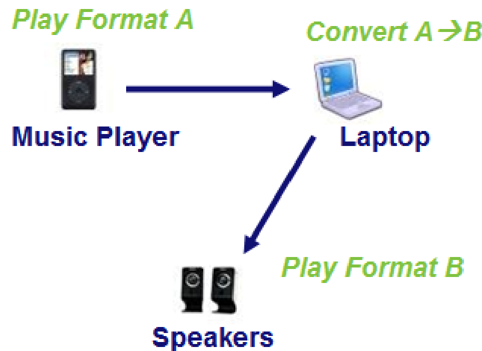

Figure 6: Service Chaining

Reliability & Robustness

The services provided by the system must be reliable, i.e. they must perform their required functions under stated conditions. In addition, the system must be robust, i.e. it must be capable of coping well with variations in its operating environment with minimal damage, alteration of services or functionality loss.

To preserve reliable and robust behavior, the system has to be *flexible*, and *fault tolerant*. Flexibility can be achieved through well defined standards. Fault tolerance is the ability of the system to cope with undesired behavior. In this case, the system will be a distributed one, and a fundamental design practice is to keep the system as *loosely coupled* as possible to prevent single entities affect the overall system behavior. For instance, an asynchronous messaging/communication model, such as publish-subscribe model, would be preferable for a smart-space system over a synchronous one just because synchronous communication increases dependency among entities.

Lightweightness

Lightweightness could be expressed as the necessary burden on a node's resources in order to join the smart space network and be a "smart node". This burden could be expressed as the size of the program code to be loaded on the node, run-time hardware requirements, communication bandwidth need and energy usage.

The variety of components in smart spaces restrains us from making assumptions about the individual devices' capabilities. The devices in a smart environment may range from resource limited simple devices to resource rich server computers. Thus, the smart space environment shall support *heterogeneity,* allowing also resource poor devices to take part in the smart space. Such integration is only possible if the agents can contribute by running lightweight operations. Therefore, lightweightness as a support for heterogeneity is of crucial importance to smart spaces.

Scalability

Scalability is a general measure of how good a system handles large sets of inputs. In smart spaces context, it is the number of services and devices in the environment. In this study, we define scalability within the smart space, not across a number of smart spaces; and this is named *localized scalability*. A scalable smart space should tackle no practical problems when the physical smart space environment is populated by many nodes.

Since the system is totally oriented to its human users, the response time of the system should not exceed a couple of seconds in most cases and maybe even more constrained in some. No matter how large the group of devices is present in a smart space, it should provide acceptable service quality whenever possible.

Data Model Standards and Knowledge Sharing

Smart spaces collect, store and act upon context data. The biggest drawback of today's smart space implementations is the lack of common standards on how to store and communicate data. This results in prototype implementations which probably never go out to the market and become commercially available. The reason is that these implementations are not *interoperable*, meaning that they can not work with other smart systems or any other system which does not execute the exact same data format and communication methods.

The generic solution to interoperability lies on taking an infrastructure approach. The advantages of an infrastructure solution are *i)* being independent from hardware and software configuration, *ii)* easy maintenance and evolution, and *iii)* sharing of resources, data and services in between entities. As a model to capture contextual data, a promising solution appears to be the Semantic Web technologies. Semantic Web aims to extend current Web into a common platform where

data can be shared and reused across virtually anything that exists in cyberspace. Today's Internet is composed of "islands in the sea" where the information is encapsulated in application specific areas. Semantic Web's aim is to overcome that by introducing standards where the information can be queried and processed automatically at the semantic level. This way, independently developed systems can share knowledge, minimizing cost and redundancy in gathering and accessing information.

A point to note is smart space systems have mostly imperfect contextual information. So, a probabilistic data storing and processing approach is necessary for this domain. At this point, an enormous advantage of use of ontologies comes into play: automatic reasoning about data. Using ontology data modeling, contextual information can be stored and gathered for reasoning about missing, inconsistent or uncertain contextual information. Logic inference engines can be implemented to achieve this behavior.

Security & Privacy

As in most service oriented systems, *security and privacy* requirements are highly relevant for smart spaces as the system is in charge of highly private and personal data. So, the access of the system to private data and, more importantly, the access of others to one's private data must be strictly controlled. On the other hand, the users should also be given the freedom to reveal their personal information selectively. Note that the smart space security service itself has to be ubiquitous and invisible to comply with the system.
To accomplish ensuring security, an Internet solution may come handy: security certificates. As long as certificates are approved by institutes that we (and the rest of the world) trust, the security would be guaranteed. As an example, a user would disallow access to his/her mobile phone by any third party except for trusted domains with valid security certificate, such as a cinema hall that would put the mobile phone in silent mode during a movie.

Network Access & Handover Control

In order to achieve full potential, the smart spaces such as home, car and office should be able to communicate and coordinate themselves for seamless intersystem roaming. One question is how to implement systems to automatically redirect users across and handover the services. H*andover control* aims to provide uninterrupted services to smart system users while the user is on the move in across many smart spaces. Handover control is an important consideration for overall network stability and quality-of-service provisioning. One another issue to address is how to control and regulate access of devices and users to individual smart spaces. *Network access control* aims to grant access to authenticated users or devices to join and make use of the network. Contemporary mobile communication technologies (e.g. GSM cellular networks) provide a solid basis to achieve desired behavior. Policies are applied to determine the rights of users and devices of what they can do on the network.

1.3 Wireless Sensor Networks

Today's information processing technology resides on large, general-purpose computational devices ranging from mainframes to modern laptops. These devices are centered around a human user of a system and has no or little relation to the surrounding physical environment. However, there are application areas where physical environment is the focus of attention. In such

application areas, embedded systems do control tasks, such as temperature monitoring and correction, in their physical environment. Their job is mostly tied to a larger scale context. Embedded systems technology has a profound place in everyday life: e.g. cars, dishwashers, mobile phones, video players.

The view of pervasive computing takes embedded control one step further as the intelligence can be implanted into the physical ambience, i.e. *ambient intelligence*. In order to realize technologies which control physical processes and interacts with human users, a crucial aspect is needed in addition to computation and control: communication. Wired communication is mostly and obstacle to success in creating as it may end up being costly in regard to the large number of devices in the target space and maintenance. Moreover, wired technology takes away mobility and it is more difficult to integrate wired devices into the space. Therefore, this new class of sensor networks has emerged: Wireless Sensor Network (WSN).

There are diverse application areas of WSN use. Mostly, the information is gathered at *source* points where the nodes make measurements of physical conditions, i.e. sensing. Sensed information may go through some simple processing and is sent to *sink* points. Sinks collect data and they are either part of the sensor network or an external entity (Figure 7). The interaction in between source and sink are usually of event detection or periodic measurements.

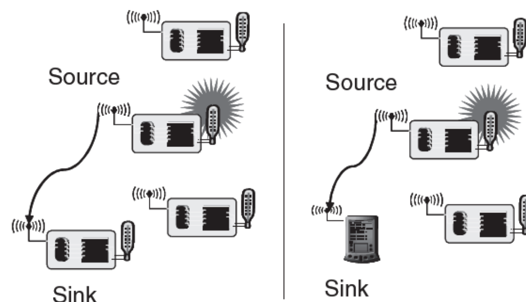

Figure 7: Two different types of sinks [9]

WSNs are intended to be used in various application areas: disaster relief operations, environment control, intelligent buildings, agriculture, healthcare, etc. In this vein, WSN phenomenon brings large flexibility and opportunities; however it introduces many technical challenges as well. First of all, WSN concept visions a *data centric* approach where arbitrary nodes are not significant, contrary to conventional address centric architectures. The importance is rather on providing meaningful answers (such as scoping information according to time and location) instead of communicating data and most importantly the specific source of data. Moreover, service quality control and fault tolerance issues are quite peculiar since (battery) energy expenditure and lifetime of nodes should be taken into account in designing an architecture. The application area is usually the determining factor of how many nodes will form a network, ranging from a couple to hundreds or thousands. Therefore, scalability and variety in density of network are significant features. Lastly, programming and maintaining of such systems pose a challenge as these nodes should be flexible enough to be programmed during their operation cycle in the field.

At the node level, a wireless sensor node is typically rather a simple and small size electronic device compared to general purpose computers. It is equipped with one or more sensors to make measurements. It has limited processing and memory resources as well as energy capacity. Since the wireless sensor nodes are not connected to unlimited energy sources (i.e. an electricity plug) it typically runs on batteries, though optionally some nodes do possess energy scavenging

capabilities (i.e. facilitating environmental energy, such as light or vibration, to charge battery) due to their field of work and specialization. The node is equipped with a radio transceiver with an antenna to be able to receive and send messages over air. Optionally, the node may have an actuator, such as an alarm to raise. In Figure 8-a these hardware elements are depicted along with a photo of a real sensor node, the size of two AA batteries.

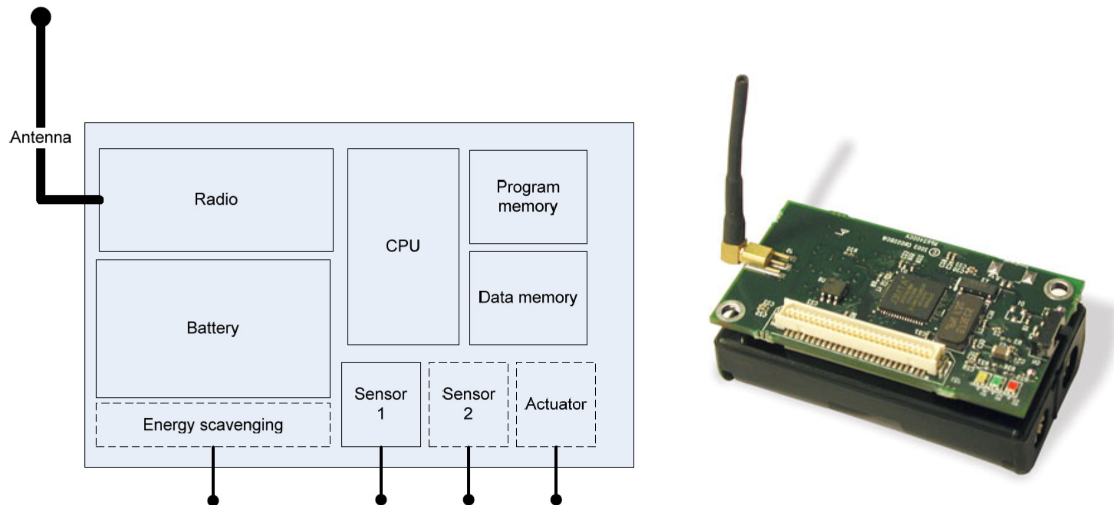

Figure 8: a) Hardware Elements of a Wireless Sensor Node b) A Wireless Sensor Node Photo

1.3.1 Relevance to This Master Project

At its core, WSN concept poses low-cost and easy to deploy solutions to meet application specific needs. The focus of such systems lies on long-term monitoring and operation especially in difficult environmental conditions where it would be too costly or practically infeasible to bring an infrastructure and implement conventional solutions. A significant part of added value lies in collaboration among nodes on the field, so that the joint contribution of the nodes sums up to an accomplished and operational application.

WSN idea gives the impression of being just the right concept to utilize in order to realize smart spaces to begin with. It provides a solid foundation for utilization of wireless communication and collaboration on small and rather simple devices which would be quite feasible to exploit for home electronics. In addition to being naturally pervasive, it provides a foundational backbone for automatic operation and dynamicity. Best of all, it compensates for financial infeasibilities that current smart space implementations suffer because WSN nodes are simple and cheap.

Nevertheless, one has to consider both sides of the coin about utilizing WSN networks. At one hand, WSNs have many extensive features; but on the other hand, these networks have limitations and considerations to take into account, especially in terms of performance trade-offs. First of all, the sensor nodes are very limited on computing capacity, making it difficult to program and design applications light, yet sufficiently proficient to match service needs. The processing power can not go beyond simple data processing (e.g. averaging, aggregation of data) and low memory limits the node from keeping loads of data or doing computations which require sizeable memory space. Secondly, the sensor nodes run on a limited and scarce energy source: battery. This brings huge flexibility to a node and the formed network particularly, however the energy expenditure is extremely crucial to take into account, if we consider operational lifetime of the nodes. Especially

communication, i.e. sending or receiving radio messages, consumes substantial energy compared to other tasks. For that reason, the nodes need a lightweight communication protocol where they can collaborate effectively to fulfill their job.

All in all, leveraging wireless sensor nodes to form or participate in smart space networks is a promising idea. As expected, smart spaces notion aims to exploit a wide range of electronic devices to join the network, and not all of them are as highly capable of resources, just as wireless sensor nodes. The difficulty is to exploit the lower end of the range; consequently a proper smart space solution should be able to support low capacity devices. This support is created by complying with requirements of lightweightness and low energy considerations. This could be done by implementing a solution architecture which requires low resource contribution costs from the participants of the network.

In the following chapter, we describe the problem of designing an architecture solution for lightweight devices (e.g. WSN nodes). Referring to related work in the field, we address the lack of lightweight architectural solutions and what benefits it could bring. Behavioral requirements of the intended system are specified.

2. Problem Description

This project primarily aims for an architectural solution for smart spaces composed of low-capacity nodes. From the perspective of this specific project, requirements regarding the system composition and networking architecture are highly relevant and important whereas requirements regarding -gathering, storing and processing- contextual information and security/privacy are beyond the project scope.

There are two specific focus points:

- Architecture of Smart Spaces (the fundamental aspects being discovery, monitoring and management)
- Lightweightness (for low capacity nodes to participate)

2.1 Architecture for Smart Spaces

2.1.1 Need for a Dedicated Architecture

In this project, the focus is on the architectural structure of smart space networks and essentially how to realize one suitable for low capacity devices. In that regard, a smart space network is basically a distributed network of electronic devices, which may be highly mobile or rather stationary. From a scientific perspective, the knowledge of mobile and distributed networks is deep and thorough, also for low capacity devices (e.g. wireless sensors). Nevertheless, how to realize an effective architecture to meet the wide variety of special requirements of smart space phenomenon, as stated in the previous chapter, is yet to be answered.

The network architecture constitutes the skeletal backbone of the smart space system, regardless of how the system functionality is implemented. Architecture model provides a foundational infrastructure as how the network is set-up and maintained as well as how the components communicate and collaborate in providing the desired functionality.

From an architectural perspective, there are specific requirements to bear in mind. Real time adaptation, as mentioned in the previous chapter, is a core requirement of a smart space architecture which refers to the dynamicity of the system as self adjustment to ever changing conditions (i.e. failing of nodes, topology changes, nodes getting in and out of the smart space network) in run-time. Moreover, from the user perspective, the services provided by the system should exhibit reliability (i.e. to avoid interruption of services) and stability (i.e. to preserve sufficient service quality). Note that, all this functionality should take place seamlessly, i.e. full automatic operation with no or minimal human intervention.

The distinctive characteristic of smart spaces is that the provided services are totally user oriented. This very characteristic requires the system to be highly responsive, i.e. quick in responding to a user input/request in run time. A smart space should ideally keep its response time as short as a few seconds, because users should not be alienated by excessive delays. Ideally, the response time should not change or become unacceptable when a large number of nodes join the network (i.e. scalability).

2.1.2 Fundamental Aspects of a Smart Space Architecture

The underlying behavioral elements of a smart space architecture solution are automatic handling of nodes and autonomous real-time adaptation and ultimately service provisioning, either upon service requests or through inference of contextual information or pre-specified rules. In this vein, we define three gradual steps which compose the core foundation of smart space architecture: **discovery**, **monitoring** and **management** of nodes and their resources/services. These steps narrow down the scope of our study, and are vital for smart spaces in general; the solution should exhibit these features regardless of design decisions made and directions taken.

These three fundamental aspects are shortly introduced as follows:

- *Service discovery* is automatic detection of nodes on a network, their offered services and resources. This is done by, so called, service discovery protocols which standardize the communication across devices in a network. In this domain, service discovery mechanism should be able to function properly under highly dynamic environments and be fault tolerant (e.g. failure of a node, lost communication messages).

- *Service/resource monitoring* is the art of maintaining consistent information regarding the network, over time. Monitoring is vital in order to establish control over the system. In this domain, monitoring resolves to keeping track of ever-changing information such as dependencies between nodes and utilization of services and resources. This information is necessary to determine the availability of the nodes to provide reliable services when requested.

- *Management* simply means to manage services and the utilization of resources in order to satisfy service requests by users. In this domain, management functionality revolves around mapping service requests to nodes at its core. Service requests could either be inferred from contextual information or performed by users of the system externally. The basic service mapping functionality could be extended with precise arbitration and providing complex or chain services.

It is worth to mention that these three fundamental aspects of implementing a smart space architecture are gradual in their functionality. They are wrapped in a hierarchical structure such that the subsequent functionality is only possible through successful implementation of the preceding one. To be more specific, monitoring capability is built upon service discovery as nodes, and information regarding them, need to be recognized and registered first in order to keep track of them. Similarly, by intuition, only monitored entities can be managed. This dependency is depicted in Figure 9.

Figure 9: Hierarchical Dependency of Fundamental Aspects of Smart Space Architecture

Above defined fundamental aspects of smart space architecture constitute the focus points of this project.

2.2 Lightweightness

2.2.1 Need for a Lightweight Solution

Ideally, an architectural solution for a smart space should address wide variety of nodes (e.g. home electronic devices) in terms of their resource capacity. It is desirable to also allow low capacity devices to join the network in order to make use of their services and thus to augment the system. This is possible only through a simple and lightweight solution which minimizes obligations of participating nodes in the network. In other words, in order to join the smart space, nodes should spend minimal extra resources; such as processing power, network bandwidth, and most importantly energy.

Joining the network requires a node to be discovered and registered in the network, and be monitored during run-time via remote calling of introspection (i.e. self-examination) operations. Compliance with this functionality is a minimum for any node, assuming that nodes only play a participant role in the system. Apart from that, the functionalities at the higher system level (e.g. monitoring the nodes) could also be implemented in a distributed way, and that means a heavier burden fall on the participating nodes.

This project aims to enable low capacity nodes to participate in a smart space. Therefore, the issue of lightweightness is, for all intents and purposes, the main point of attention in this research. Participation of low capacity nodes will enrich and extend the capabilities of smart spaces as well as the user experience. More importantly, if the system could support simpler and cheaper devices, such as wireless sensor nodes, the overall system implementation in general would become feasible in financial terms.

2.2.2 Low Capacity Devices

In this sub-section, we provide a definition of low capacity devices by declaring their hardware configuration and setting boundaries on their capabilities. Figure 10 illustrates hardware components for a low capacity node in basic terms.

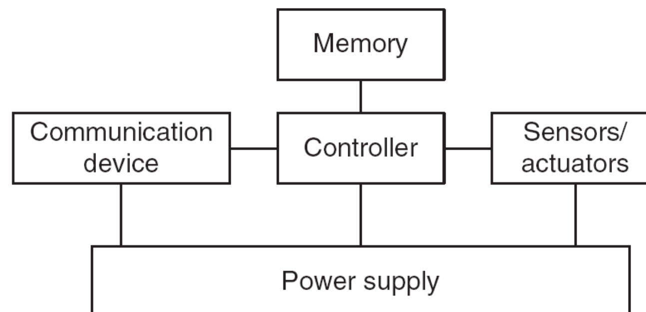

Figure 10: Overview of Low-Capacity Node Hardware Components [9]

The controller of the node is the processor unit. In design of low capacity nodes, controllers are referred as microcontrollers, which are flexible in connecting to other devices (e.g. sensors) and consume little power. A microcontroller usually has 8 or 16 bit Reduced Instruction Set

Computer core with 1-25 MHz clock frequency and consumes energy in 1-20 milliWatt range when active.

Node memory consists of Random Access Memory (RAM) to store intermediate data (e.g. sensor readings, incoming packets) and Read Only Memory (ROM) to store program code. Nodes typically have 1-10 kilobytes of RAM and 1-128 kilobytes of programmable Flash memory.

The communication devices are radio frequency transceivers that utilize communication frequency band of range between 433 MHz and 2.4 GHz, with a data transmission rate up to 250 kilobits per second. Radio transceivers consume in range of 100 to 400 milliWatts when they are receiving or transmitting data.

Node power supply is a rechargeable battery, mostly realized as a lithium-ion polymer battery which has an energy density of around 1000 Joules/cm^3. The capacity is about 2.0-2.5 Ampere-hours for size of an AA-battery. Scavenging energy from the environment is available (e.g. solar cells).

The node usually possesses passive sensors which make measurements by probing the environment (e.g. thermometer, light sensor, vibration sensor). Sensors' power consumption can be ignored in comparison to other devices (e.g. radio transceiver). Actuators are limited to simple operations on node side, such as opening a switch to raise an alarm in case of fire detection.

2.3 Related Work

The idea of smart spaces has received significant attention from both industry and academia. Roughly covering the last decade, there has been a profound research in this field. The concept was realized in different ways using different approaches, each of which proposes a diverse definition of smart spaces. Therefore, design solutions reflect on different trade-off points and eventually end up being suitable for specific purposes. To the best of our knowledge, there are no lightweight smart space architectural frameworks designed for very low capacity nodes in the literature. There is a profound research on smart spaces; nevertheless solution approaches are designed mostly for high capacity nodes (e.g. PDA, tablet PC, laptop) and with lacking support for dynamicity.

In [10], an approach oriented towards high-mobility smart space environments is presented. For discovery of devices and services, Bluetooth service discovery protocol is used. JCAF[11] and [20] implements an event based subscription model and uses JINI [12] for service discovery and JMS[13] for messaging. These approaches mainly lack in supporting highly dynamic environments with high availability. Due to burdensome requirements, such as the Java Virtual Machine and IP based addressing, cost of communication is increased. This results in hindering low capacity nodes to join the network.

In [14], a centralized gateway solution is proposed, possessing an asynchronous event-subscription model. It uses Service Locating Service [15] for dynamic service discovery and binding of service requests to nodes. In addition to [14], solutions proposed in [18] and [21] focus on decoupling application layer from middleware layer providing integration of management functionalities into applications. This is a useful approach as it brings ease for application developers through high level of abstraction introduced though layered architecture. However, the

proposed framework limits further optimization in energy expenditures and resource utilization as cross layer optimization option is naturally sacrificed.

Architectural solutions such as [16], [17] and [18] propose XML based data models for messaging. Many other solutions introduce even more complex data formats for better structure and more expressive power. Nevertheless this practice brings an extra burden on the nodes and boosts up energy consumption with message format overheads.

Solutions which focus on contextual information gathering and management, such as CoBrA[22], have poor support for resource discovery or none at all. Resource discovery mechanisms are currently rarely used in contemporary frameworks [23]; this implies that these systems assume nodes are stable and permanently available. Therefore, these solutions disregard high mobility needs, as discussed in [19], and dynamicity is mostly not taken into account.

2.4 Objective: An Architectural Solution for Low Capacity Devices

To come to the point, this project aims for an architectural solution which enables implementing a smart space system out of low capacity devices, which is not specifically addressed by any other solutions in the literature.

We redefine and narrow down the scope of this project to realizing an architectural solution of service discovery, service/resource management and monitoring functionalities for a distributed wireless network system composed of WSN nodes.

Conditions and boundaries regarding the required system behavior are stated below. We aim to come up with a solution satisfying the following conditions:

- A node entering the vicinity of the network is discovered within 5 seconds. This means detection and proper registration of the node to the system, such that the node is considered to be actively taking part in the network. Discovery mechanism requires the node to report its presence with the relevant definitive information such as node description and the services possessed. [1]

- A node not responding to the smart space system is unregistered and dropped out of the network within 5 seconds. This might be caused by the physical departure of the node out of network vicinity, failure of the node or simply lack of communication due to message loss. Therefore, recent information regarding the node’s healthy presence in the network is no older than 5 seconds. [1]

- Miscommunication due to dynamic network topology change is not tolerable. When a node physically moves within the smart space network, it shall remain in the network

[1] We assume the stated boundary conditions proposed are acceptable for satisfactory user experience. Stated time duration values are intended to be the average for the declared behavior to happen; the tolerable worst case value is assumed to be double the average value, assuming no packet loss.

without being dropped. Its connectivity shall remain operational regardless of the network topology.

- The system maps an external service request to nodes accordingly. It is done by resolving the service request to selecting a ‘best’ provider-subscriber pair and matching them up so that the provider provides its services to the subscriber. The smart space system is responsible for guaranteeing that the provider and subscriber can communicate to each other, i.e. there exists a valid communication path from the provider to the subscriber so that the service messages can be sent.

- Individual nodes as single entities can not fail the system or other nodes. The entities are decoupled both in functionality and communication. When a node exhibits unexpected behavior for any reason, the overall system is not affected any more than as if the node is dropped.

- Fast changing information regarding a node’s run-time resources is monitored such that it is no older than 30 seconds. This requirement is made on the assumption that a critical resource, such as remaining battery energy, could change considerably within that interval. This assures that resource use information is acceptably recent so that the system can use it to make decisions of dynamic adaptation and management, e.g. assigning a service to another node which has more remaining energy.

- Run-time information regarding the use of services (i.e. which node provides what service to whom) is monitored by the system with the same precision of node presence in the network, i.e. 5 seconds interval. This requirement applies only for the services which are initiated and run under the supervision the smart space management mechanism.

- The message size used for communication among the smart space network nodes is below 100 bytes per message.

- The smart space participation cost to a node is less than 1 kilobyte of RAM and ROM in total.

- An idling node registered in the system does not send/receive more than 100 messages per minute. This condition disregards redirected or ignored messages.

- If any two arbitrary nodes need to communicate to each other for any reason, they know how to reach to each other, i.e. they have a network route in between. The smart space system is responsible to assure proper communication in general.

- All functionality stated in the previous bullets takes place in an absolute total automated way. There is no human intervention as the system exhibits the required behavior.

3. Proposed Solution

Our target is to develop an efficient hardware and software architecture for service and resource discovery, monitoring and management in a smart space composed of very low capacity nodes. The proposed solution is an alternative to the computationally heavier architectures of similar kind that can be found in the literature (see Chapter 2). In order to achieve this target, we have chosen to take an existing WSN solution, namely the Open Service Architecture for Sensors (OSAS), as the base architecture. This has enabled us to speed up the implementation process, providing a good starting point, since WSNs are pervasive in nature and they have wireless communication and collaboration abilities which could be tailored to exhibit smart space behavior.

3.1 OSAS Framework

OSAS has been developed as a framework for programming sensor networks [24]. Using OSAS as the base programming architecture brings the following advantages:

- Lightweight and scalable
- Publish-subscribe model
- Event based
- Realistic simulation environment
- Developed by System Architecture and Networking group at TU/e (where this project took place)

Publish-subscribe model allows for decoupling of publishers and subscribers, constituting a firm support for scalability and dynamicity. Publishers are loosely coupled to subscribers, and needn't even know of their existence; each can continue to operate normally regardless of the other.

Event based operation; hence event-driven architecture enables the components and services of the system to be loosely coupled and well-distributed. This way, more responsiveness of the overall system is ensured, because the system is more normalized to unpredictable and asynchronous environment.

The OSAS model proposes programming the network as a whole instead of programming individual nodes one by one. This approach conceives the WSN in three distinct views: the application, the network and the nodes. At the application level a program is written that specifies the behavior of the network entirely. At the network level, the application program is compiled into a series of small configuration messages which are flooded through the network and conditionally installed into the nodes. Finally, at the node level, a virtual machine is run, which abstracts the node hardware specification and executes the bytecode. This way, a single program can be sent over the network and the nodes translate the corresponding parts of the program into a set of configuration messages and bytecode. This practice is also known as macroprogramming.

The programming platform is realized using a toolchain which consists of: a compiler (to compile application program code to bytecode and configuration messages), a loader (to upload these

configuration messages to sensor nodes, either physical or simulated nodes), a simulator (to test and rapid prototype), and an interpreter (to execute bytecode on the nodes).

We introduce OSAS framework more thoroughly in the rest of this section.

3.1.1 Programming Model

In the programming level, the nodes run services. A service has a state, generated *events* and event *handlers*. The flow of the program is determined by the events and a service can generate events or/and consume (handle) events in principle.

Services can generate events through event generators. The process of linking the event generator of one service to the event-handler of another is called a *subscription*. When a subscriber subscribes to an event, the event generator is triggered periodically. The triggering period is determined by the subscriber. There could be many subscriptions on a single service and subscription period may differ per subscription. When an event generator is triggered, it tests its condition to fire an event.

Example: A gateway node subscribes to a temperature node's *temp* service, which is a service that reports sensed temperature values. The gateway node needs temperature values in periods of 5 seconds and it is stated in the subscription. Say, its firing condition is that it only sends temperature values if it measures an unexpectedly high value, i.e. greater than 40 degrees Centigrade. All this information results in the event generator on the provider side (the temperature node) to trigger every 5 seconds, do a measurement, and check if it is over 40 degrees. If so, the provider reports the temperature value to the subscriber.

The interaction between the services is realized by a message exchange. When an event is generated, it is sent to the subscriber node to be received and handled by a particular event handler. In general, a typical message is of the following format:

HandlerID	**arg_0**	**...**	**arg_n**

When the message is received, the corresponding handler (with the handler id as stated in the message) is called with the parameter list provided in the message. In this vein, event handlers could be conceived as callback functions. Note that binding of event generators and event handlers, i.e. subscriptions, is not part of the services and is externally implemented.

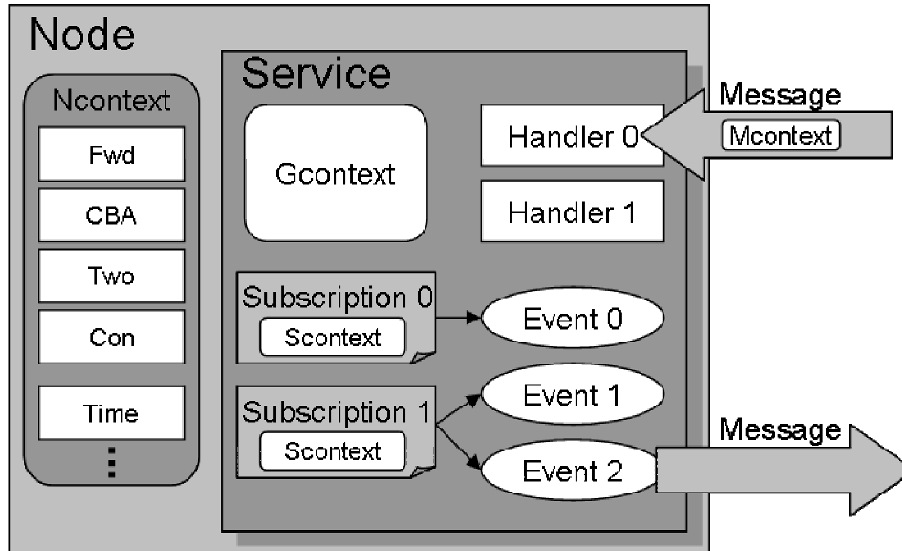

Figure 11: Execution Contexts of a Node

There are four contexts in which the services are executed (Figure 11):

- **Ncontext** (Node context) is a list of globally available, node-wide constants and values. It consists of a collection of system calls (e.g. a function to read sensor values) and system handlers. System handlers are a part of the bootstrap mechanism which is required for fundamental system tasks, such as installing new service to a node.
- **Gcontext** (Global context) is the global state of a service which is shared between all its event generators and event handlers.
- **Scontext** (Subscriber context) is the state of the subscriber which contains values determined by the subscriber side for that specific subscription. Furthermore, it contains the standard subscription specific information such as triggering period of the event generator and the address of the subscriber.
- **Mcontext** (Message context) is the state of a message. It contains the body of a message which is basically the handler id followed by function parameters.

3.1.2 System Architecture

In Figure 12, the architectural view of OSAS is presented. A node has services which are composed of event generators and event handlers, along with the service state (GContext). Note that system service, the infrastructural service to start with, is also a service itself. Services have subscriptions, with subscription state context (SContext). Subscriptions generate schedulers which trigger event generators.

Both event generators and handlers can make system calls, which basically specify the node's type and capabilities and have a context (NContext) accessible throughout the node. For instance, a temperature sensor node would have a system call to make temperature measurements, such as *Temp()*. Again, both event generators and handlers are bytecode fragments which execute on a virtual machine running on the node. Lastly, event generators generate events and create messages with their own context (MContext), whereas event handlers consume and process messages.

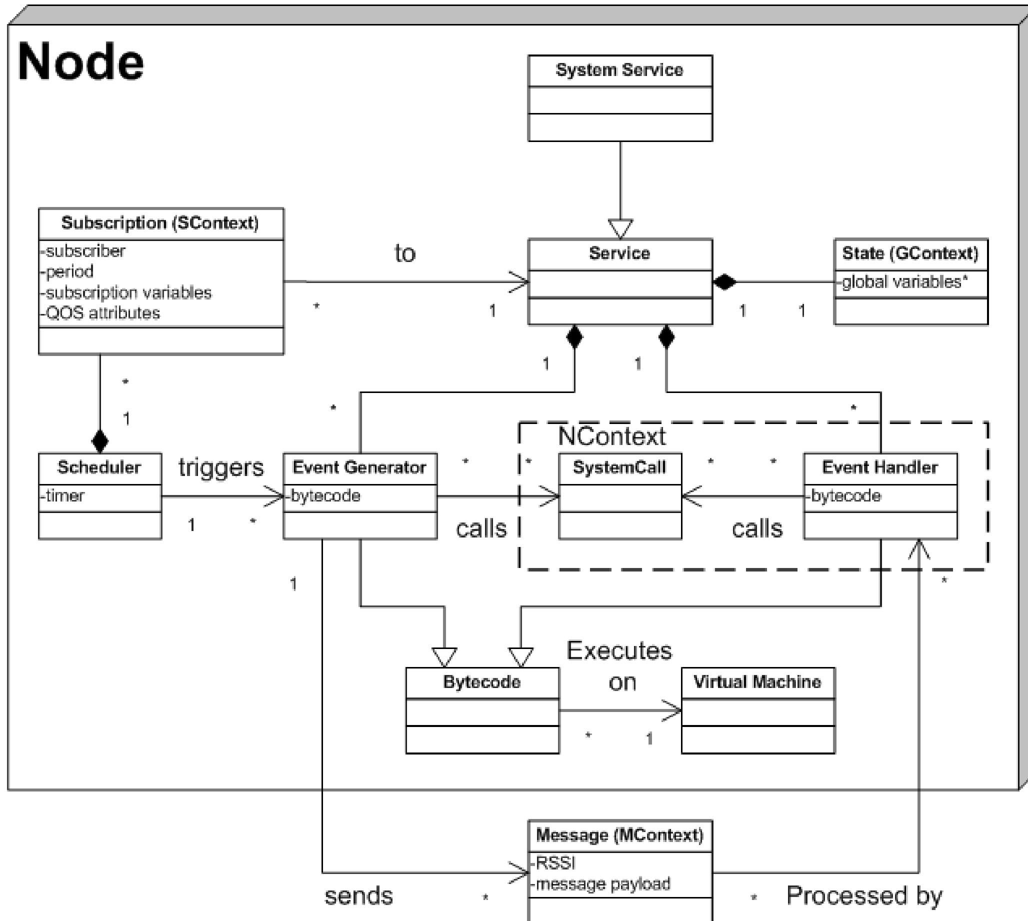

Figure 12: Architectural view of OSAS

3.1.3 Network Programming Language

The network programming language of OSAS is called WASP [29] Service Composition Language (WaSCoL). It lets the programmer to write one single program, compile the code and automatically deploy it to various nodes.

The language mainly has two main functions:

- Specification of services with event-condition-action rules
- Composition between these services (subscriptions)

The main concepts in the language are system calls, services (i.e. implementations of event generators and handlers) and subscriptions. System calls can be predefined and they depend on the code flushed into the nodes. In the scope of this project, let's assume that stated system calls exist on the intended nodes.

Exemplar Event Generation:

```
1 service HelloWorld($Handler)
2 for [Network|*]
3 on event e0 when True do
4   SendToSubscribers($Handler, "Hello World!")
```

The above lines of code is an exemplar declaration of a service with one event generator, named *HelloWorld*. The service has a parameter named *$Handler*, which is basically the handler id to be passed on when a subscription is made to this service. The handler id is used in line 4 to be added into the message sent to the subscriber.

The second line shows a for clause. The for clause declares an address predicate which identifies to which nodes this service declaration applies. Its general form is:

```
for [<scope>|<selector>|<predicate>]
```

The scope identifies a subset of nodes in the network, or it could be set to *Network* to address the whole network. The selector is used to select some candidates from the subgroup; it could be set to '*' to select all nodes of the scope. The predicate is an expression which yields either true or false. It defines an address predicate which is sent to and stored on all nodes in the network. Whenever a node receives a message, the message can refer to an address predicate to test, in order to verify if the message should be processed or ignored.

The third and fourth lines show the definition of a very simple event generator. Event generators are event-condition-action rules. The condition is a predicate which results in *True* or *False* boolean values. The action will typically contain at least one call to *SendToSubscribers* which notifies all subscribers that an event has occurred.

The call to *SendToSubscribers* above can be interpreted as the following remote method invocation (without any return value) on all subscribers:

```
$Handler("Hello World!")
```

Exemplar Event Handling:

```
1 service Gateway()
2 for [Network|*|HasFunction(print) && NodeType()=="Gateway"]
3 action PrintValue(value) do
4   print(value)
```

The above lines of code declare a service with one event handler, named *Gateway*. In the addressing (line 2), note that the predicate indicates that this service is defined on nodes which possess a function named *print* and whose type is *Gateway*. Note that this addressing scheme is different from conventional point-to-point addressing. This way of addressing network elements by their data is called content based addressing.

Lines 3 and 4 specify an event handler, named *PrintValue*. An event handler declaration could be conceived as a function declaration which has a name, parameter list and a sequence of statements.

Exemplar Subscription:

```
1 subscription HelloWorldSubscription
2 for [Network|*|HasService(Gateway)]
3 to HelloWorld($Handler=PrintValue)
4 on [Network|*|HasService(HelloWorld)]
5 with (period=5s..30s, deadline=2m, send="Normal", exec="Normal")
```

The above lines of code declare a subscription, named *HelloWorldSubscription*. This subscription specifies that all nodes with a gateway service (line two) must subscribe to all nodes with a HelloWorld service (line four). Subscribing to a service will cause all event generators of that service to send their events to the subscriber. In the third line, which service to be subscribed (*HelloWorld*) is stated along with a subscription variable (the handler being *PrintValue*). The last line specifies the period of the subscription (along with other non-functional properties of subscription such as quality of service parameters), as how frequent the event generator would trigger.

Note that, the frequency of event generator trigger and the values of quality of service parameters are determined by the subscriber, not the provider. The consequence of this is that the eventing interval and the subscription variables can differ per subscriber. For this reason event generators execute within the context of a specific subscriber. A side note: The non-functional properties of subscriptions other than the period are not practically in use yet, since OSAS framework is a work in progress.

In the language, more advanced constructs can be defined. Basic control flow constructs are supported using *if <cond> then <stats> [else <stats>] fi* and *while <cond> do <stats> od* syntax. Services can have state information declared in their description code. Nested addressing in subscriptions is also possible.

3.1.4 Messaging and Communication

OSAS aims for encoding messages in an effective way to minimize the message size. There are three encoding schemes introduced in the system:

- Eight-bit encoding: Every value is encoded in one byte.
- Variable length encoding: Every value is encoded one, two or three bytes according to its integer value range.
- Sixteen-bit encoding: Every value is encoded in two bytes.

All messages are sent as a sequence of integers and packing of messages is done accordingly, minimizing the message size. In other words, the encoding scheme resulting in the least message size is selected automatically. The encoding used for a message payload is put into the message header so that the receiver side can properly decode it.

The decision of selection of an encoding mechanism is done as follows: if all values in the payload are within range [0-255] then eight-bit encoding is selected. This way, encoded message length in bytes is equal to the number of integers encoded. If at least one integer value in the message exceeds the upper limit of 255, the message will be encoded in either of the remaining two options (i.e. variable length encoding and sixteen bit encoding). At this point, a quick computation is made to see which encoding would result in the least message size to make a selection. Indeed, this computation checks whether variable length encoding would result in a

message length in bytes more than two times the number of integers encoded, as it would result in sixteen-bit encoding.

The above selection mechanism determines lower and upper bounds for encoding message payloads. The encoded message payload size in bytes is between one and two times the number of the integers to be encoded.

For communication, WaSCoL programming language provides two alternatives: point-to-point and content based addressing. Point-to-point addressing is conventional addressing scheme for most contemporary systems (e.g. IP addressing), whereas content based addressing is data-driven and is useful to address many points which ensure a content predicate (e.g. possessing a service or a system call). Content based addressing constructs a backbone structure for OSAS platform and WSNs in general because groups of nodes can be communicated with ease where their features are important but not their identities. This way, the entire network could be programmed with ease, as implemented in OSAS framework.

There are two high level constructs for communication in OSAS. *SendMessage* is used to flood a packet to the whole network whereas *SendToSubscribers* is used to route the packet to subscribers for a service. A route is set from the service provider to the subscriber either via a subscription request (automatically) or flooding of a *SetRoute* message (manually). The internal mechanics of communication in OSAS are not discussed any further since it is out of the scope of this work.

3.1.5 OSAS Toolchain

The OSAS toolchain consists of three main programs: simulator, compiler and loader. How these modules are connected is depicted on Figure 13.

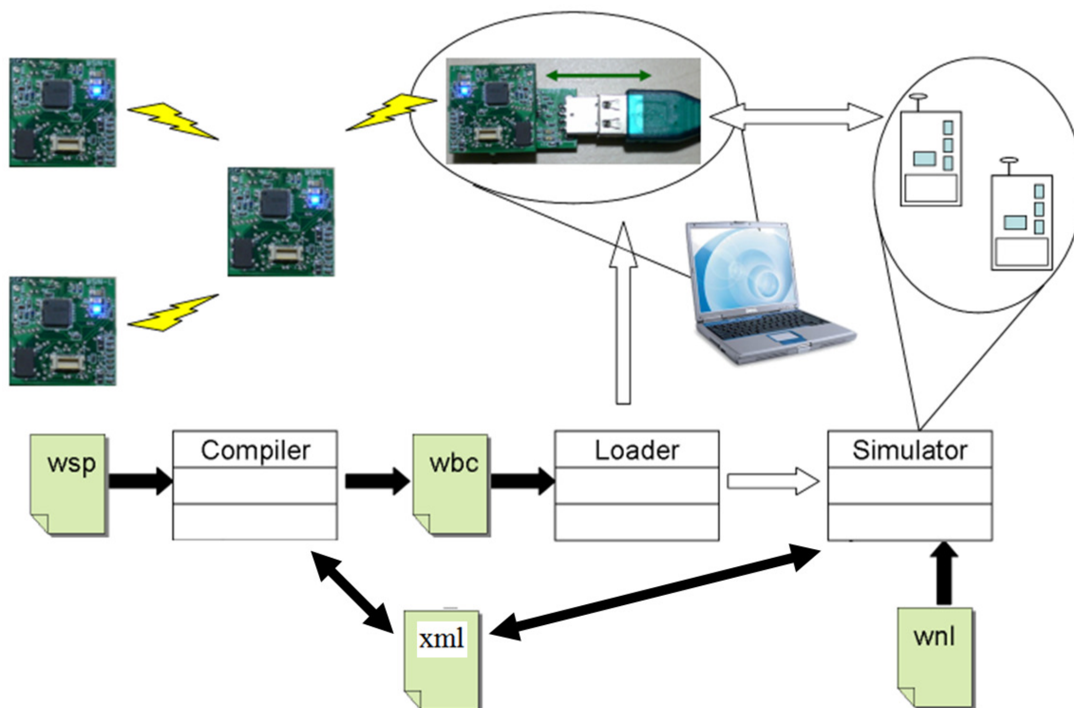

Figure 13: The OSAS Toolchain

Programming a sensor network with OSAS programming model is done in three stages:

1. A program is written to define services and compose them (subscriptions). This file is a *.WSP* file written in WaSCoL language.
2. Program file is passed into the compiler and the compiler generates bytecode (*.WBC* file). This step also requires the definition of a proper symbol table file (*.XML* file) for the compiler. This symbol table maps names of system calls, services, subscriptions, handlers, events and variables onto short IDs such that communicated messages remain small. These IDs are static and globally defined all over the network. The compiler has two outputs. First it creates a set of configuration messages which can be loaded into the network (bytecode or *.WBC* file). Second it writes an updated version of the symbol table to disk. This updated symbol table can be used to extend or reconfigure an already running network.
3. The loader only takes the configuration message bytecode (*.WBC* file) as input. It loads the bytecode to either a physical or a simulated network of nodes. The network at node level is programmed this way.

The simulator is used for testing and rapid prototyping. To compensate for the missing physical nodes, the simulator reads in a scenario description file (*.WNL* file) where the simulated nodes (i.e. their types and physical locations on a 2D bird-sight view) are declared. Simulator reads and updates the symbol table file with introduced systems calls, handlers and node types defined in scenario description file.

3.2 Implementation Alternatives

OSAS programming platform is specifically intended to program WSNs and well tested on real nodes. Therefore, an extension to OSAS which runs on physical nodes would practically ensure lightweightness requirement. Considering this, the scope of this project boils down to leveraging OSAS to realize a smart space network out of a WSN.

However, at this point, we are faced with a major decision point concerning apportioning of responsibilities. The question is how to divide and carry out smart space functionality in the network, and basically who takes what responsibility for the system to sound and exhibit smart space behavior.

There are basically two extremes on the scale:

- The solution architecture is totally distributed. No mediator is existent and the responsibilities of smart space system is divided and assigned to participating nodes.
- The solution architecture is totally centralized. There exists a mediator who takes on the responsibility of smart space functionality. The clients are simple and self interested, i.e. they do their tasks when they are asked and they do not actively engage in making decisions to manage the network in order to exhibit smart space behavior.

Main advantages and disadvantages of these two ends are listed below:

Distributed Solution	*Centralized Solution*
Pros:	Pros:
- Better fault tolerance	- Nodes are self interested
- Localized operations	- Practically easy to implement
Cons:	Cons:
- Burden on nodes	- Single point of failure

A distributed solution offers better fault tolerance since that there is no single point of management. The responsibility of failed nodes can be replaced and reassigned to operational nodes. Faults can be tolerated by dynamic revising of the responsibility division. The implemented system is not dependent on any other mechanism and it becomes truly pervasive. Moreover, decisions and operations concerning a localized portion of the network will be carried out in a localized way. This brings much efficiency in communication since remote messaging will be minimized. On the contrary, a pure distributed solution lays a heavy burden on the nodes since the smart space functionality will be shared among the nodes. This is something we want to avoid beyond doubt.

A centralized solution offers no extra burden on the nodes, allowing the nodes to be self interested, i.e. the nodes do their individual tasks only and they have no managerial contribution in implementing smart space functionality. The nodes are assumed to be truthful in their reporting of any requested data, and cooperative so that they do what they are asked to do. Rather than the nodes, a central manager is responsible for the system to exhibit smart space behavior. This implies all functional tasks are carried out by the central manager. A weak point is that the central manager becomes the single point of failure. If the central manager fails, the whole system fails. A major advantage is that central solution architecture is relatively easy to conceive and to implement. From a simplified point of view: Required data will be collected on the central manager and it will decide on how to keep consistent state and manage the whole network from one point.

As a third option, a hierarchically distributed approach could be considered. This approach is a hybrid between centralized and distributed solutions which tries to benefit the advantages of both approaches with minimal side effects. However, our aim is to provide a proof of concept; therefore we leave the hierarchical approach as future research.

3.2.1 Our Choice: Centralized Solution

From this project's point of view, the one crucial constraint of solution approach is lightweightness. Therefore, we absolutely aim to avoid laying unnecessary burden on the low capacity nodes. For that reason, our solution decision is implementing a central server to administrate and manage the network in order to exhibit smart space functionality.

Moreover, even though we define the smart space nodes to be highly dynamic (i.e. nodes move in and out of the space), one thing to note is that the physical space (i.e. living space) is not. This gives us the room to establish any infrastructural necessities, such as the central manager itself, into the space in a static way. Consequently, we can assume that the central manager is not dynamic (it stays at a specified point in a smart space) and it is a persistent component within the physical space. For that reason, it can be designed as a powerful node within the network, such as

a simulated node running on a laptop computer. This way, the central manager can draw all managerial burdens on itself because it can disregard resource considerations, such as energy use and processing power.

We name the centralized server of the smart space solution as *Resource Manager*. The resource manager is practically a simulated node running on a PC so it is free of limitations of its capabilities, compared to physical low capacity (i.e. WSN) nodes. The smart space functionalities, namely service discovery, service/resource monitoring and management, are responsibility of the resource manager to carry out.

3.3 Service Discovery

We have put Resource Manager (RM) into practice as a simulated node running on a PC. Thus, the specification of RM is implemented in the simulation development environment and the program code specifying RM is written in Python high level programming language [26]. What we need to do is to create RM as a specialized node extended with dedicated system calls, event generators and event handlers so that it administers the network to exhibit smart space behavior. At architecture level, we call this dedicated functionality as *Smart Space Administration* (Figure 14).

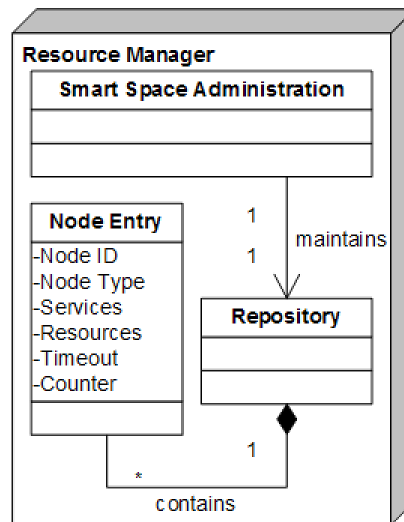

Figure 14: Architectural View of Resource Manager

RM specifically needs to contain relevant information regarding the nodes present in the network. All this information is stored in a data structure unit named *Repository*. The Repository contains node entry objects, every one of which represents a node in the network. This node entry objects are replicas of the real nodes, representing what RM knows about any particular node. The particular contents of a node entry and the consistency of the stored information as to what degree it reflects the real node state will be discussed in the next section *Service/Resource Monitoring*.

In this view, service discovery functionality resolves to four steps:

1. Detection of a node.
2. Gathering descriptive information (i.e. node type, services and resources).
3. Storing gathered information in the repository as a node entry object.

4. Keeping the repository up to date regarding nodes' presence, so that node entry objects could be relied on to infer that actual nodes are present in the network physically.

Discovery of the nodes and discovery of their services and resources are discussed separately in the upcoming sub-sections.

3.3.1 Discovery of Nodes: The Heartbeat Protocol

In our implementation, all stated service discovery functionality is accomplished by a soft-state service discovery protocol. The binary state protocol, as used in this project, is also named the *heartbeat protocol*. In essence, the nodes send their 'heartbeat' signal to RM periodically. As long as RM receives heartbeat signals, it infers that the node is existent in the network and it is operational. Figure 15 illustrates an exemplar case. Node A successfully transmits its heartbeat signal to RM, therefore RM creates and keeps a node entry for Node A indicating that Node A is existent in the smart space network. On the other hand, Node B's heartbeat signal is not received by RM, for example, due to message loss. Consequently, RM does not know that Node B is still in the network. As a result, the entry for Node B is deleted from RM repository and this node is considered out of the smart space network.

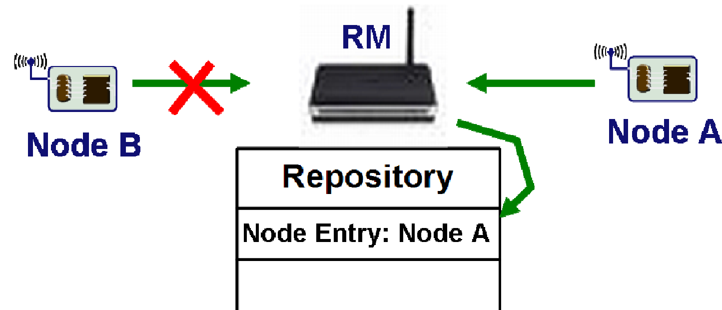

Figure 15: Heartbeat Signaling Example

In the heartbeat protocol, the presence of a network node has to be maintained by periodic refreshments of its state and not by explicit declaration (as compared to a hard state protocol), i.e. RM receives the heartbeat signals periodically. If RM misses more than a specified number of heartbeat signals, it infers that the node has either failed or got out of the physical vicinity of the network. Thus, the node is unregistered, i.e. the node entry of that particular node on the repository is deleted. Heartbeat protocol provides an asynchronous messaging mechanism which is desirable for dynamic wireless communication systems with unreliable communication channels because it results in decoupling and fault tolerance.

The constructs of periodic refreshment is implemented on the node entry object itself. The *timeout* and *counter* values are used for keeping track of the state of a node. The *timeout* value is the absolute time duration after which a node is dropped out of the network if heartbeat signals are missed. This value is typically a factor of the heartbeat reporting period and it could be optimized with respect to the reporting period and channel latency [25]. *Counter* value is the dynamic tracker of time remaining for a node to be dropped if RM does not successfully receive its heartbeat signal for that duration. Expiration of *counter* value results in removal of the node entry from the registry. The update of the *counter* value (and consequential action of deleting a node entry) is done for all the nodes registered in RM. For doing this, RM subscribes to its own *RefreshRegistry* service which deals with updating the registry. This service periodically measures the time elapsed since the last refreshment of the registry and updates the *counter* times

for the registered nodes by decreasing them. If a *counter* value gets below zero value, the corresponding node entry object is deleted from the registry.

When RM receives a heart beat signal, there are two cases:

- If the node is not registered, RM creates a node entry in its registry and prompts for relevant node information. The reasoning and the mechanism behind gathering node information are discussed later in section *3.3.3 Discovery of Services and Resources*. Our focus at this point is node discovery.
- If the node is already registered, RM updates the state of the node by refreshing the *counter* value of the corresponding node entry. This is done by simply setting the *counter* value to the *timeout* value.

Below is an example sequence diagram demonstrating registration of a node in the system, update of its counter value and dropping of the smart space network due to timeout caused by missed heartbeat signals (Figure 16).

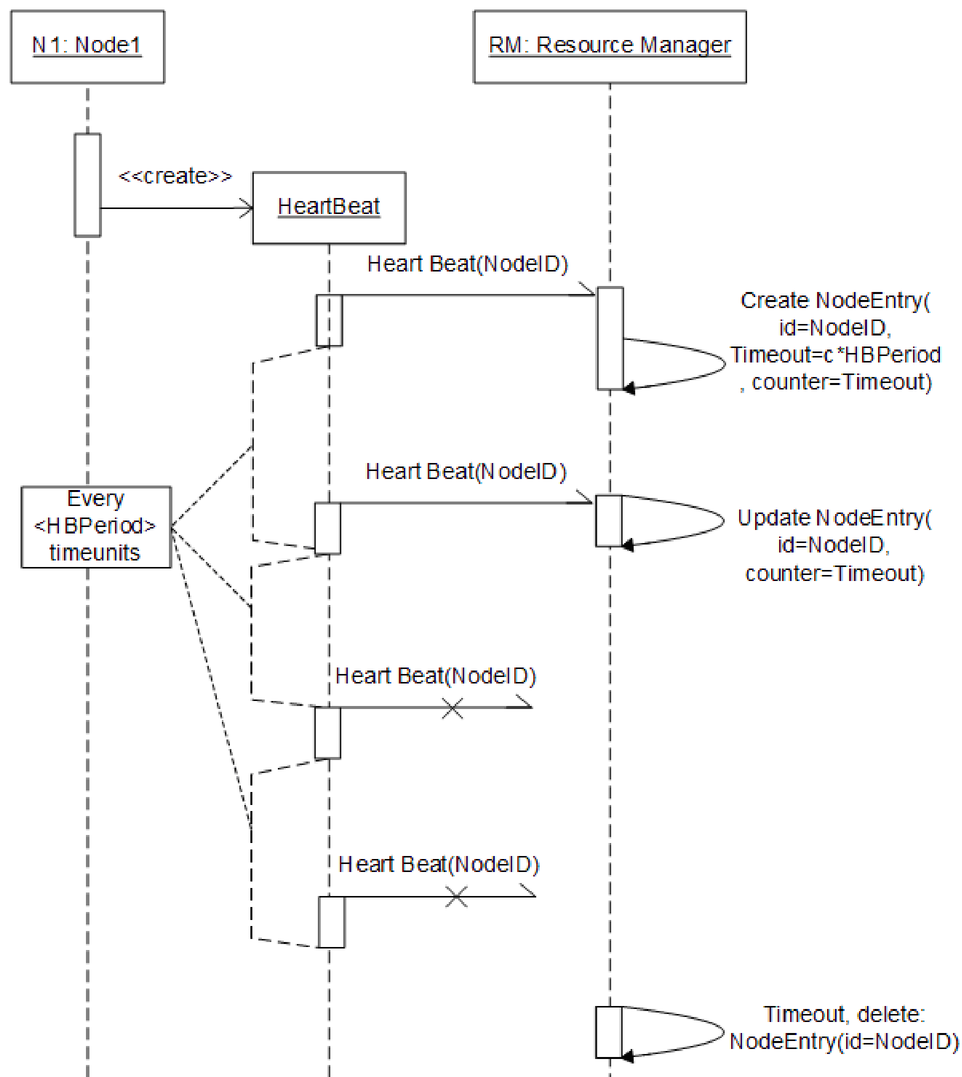

Figure 16: Sequence Diagram Demonstrating a Node Registration, Counter Update and Dropping

The heartbeat signal is basically a message created by the event generator on the node side. It contains two values in its payload, the *HandlerID* of the receiver side and the *node id* of the sender. We assume that *NodeID* is a static value which uniquely identifies a node on a network. Any arbitrary node that can join the network is called a *HeartBeat Node* (or shortly *HB Node*) because it possesses *HeartBeat Service* and sends heartbeat messages. When the message is received at RM side, its payload is handled by the corresponding handler (*HeartBeatHandler*). The *NodeID* is passed as a parameter to the handler. The relationship in between RM and a node is basically a subscription as depicted in Figure 17.

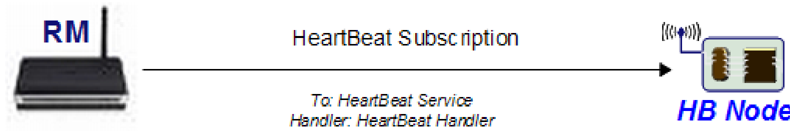

Figure 17: HeartBeat Subscription from RM to HB Node

RM subscribes to the *HeartBeat Service* of the *HB Node*. In return, the *HB Node* automatically constructs a route to RM (i.e. it knows how to communicate to RM), and reports its presence through periodic firing of its *HeartBeat Service* event generator. The frequency of this is determined by RM and this issue is discussed in section *3.4.3 Reporting Intervals & Triggers*. Two questions arise: i) How is this subscription mechanism initiated, and on what condition? ii) What if the route to RM is lost due to dynamicity of the network? These two issues regarding bootstrapping and support for dynamicity are discussed in the following sub-section.

3.3.2 Proposed Enhancements to Heartbeat Protocol

The heartbeat protocol is enhanced in two ways:

- Periodic advertisement
- Conditional route reconstruction

The initiation of discovery (bootstrapping) takes place in RM side. Periodic advertisement is announcement of RM of its presence by flooding a subscription message to all *HeartBeat Nodes* (which have *HeartBeat Service*) throughout the whole network. When a node comes within the vicinity of the smart space network, it eventually receives the subscription request and will create the subscription and start reporting its presence (i.e. heartbeat messages) periodically. This advertisement is done actively on RM side so that a new node entering the network can discover RM. If a node is already in the network and receives the advertisement message, it overwrites the subscription it already has. Yet, the mechanics of subscription do not change and there is no behavioral or functional change. Periodic advertisement makes sure that any new node entering the system is discovered: i) it constructs a route to reach RM, ii) it starts reporting its presence. This enhancement provides discovery mechanism to be carried as an automatic operation (invisibility).

Conditional route reconstruction occurs when a node moves within the smart space resulting in a change in network topology such that the node can not reach RM via the same route. As a result, the heartbeat messages do not get delivered to RM, as intended. This is where conditional route reconstruction takes place: when a node is about to be dropped due to timeout. In order to keep the relocated node in the system (without dropping it), a rerouting message is flooded in the system and nodes reconstruct their communication paths to RM. Therefore, a relocating node is not dropped out of the network. For this specific purpose, a high level messaging construct of

WaSCoL programming language is used: *SetRoute* (Please refer to section *3.1.4 Messaging and Communication*). This enhancement introduces a firm support for dynamicity.

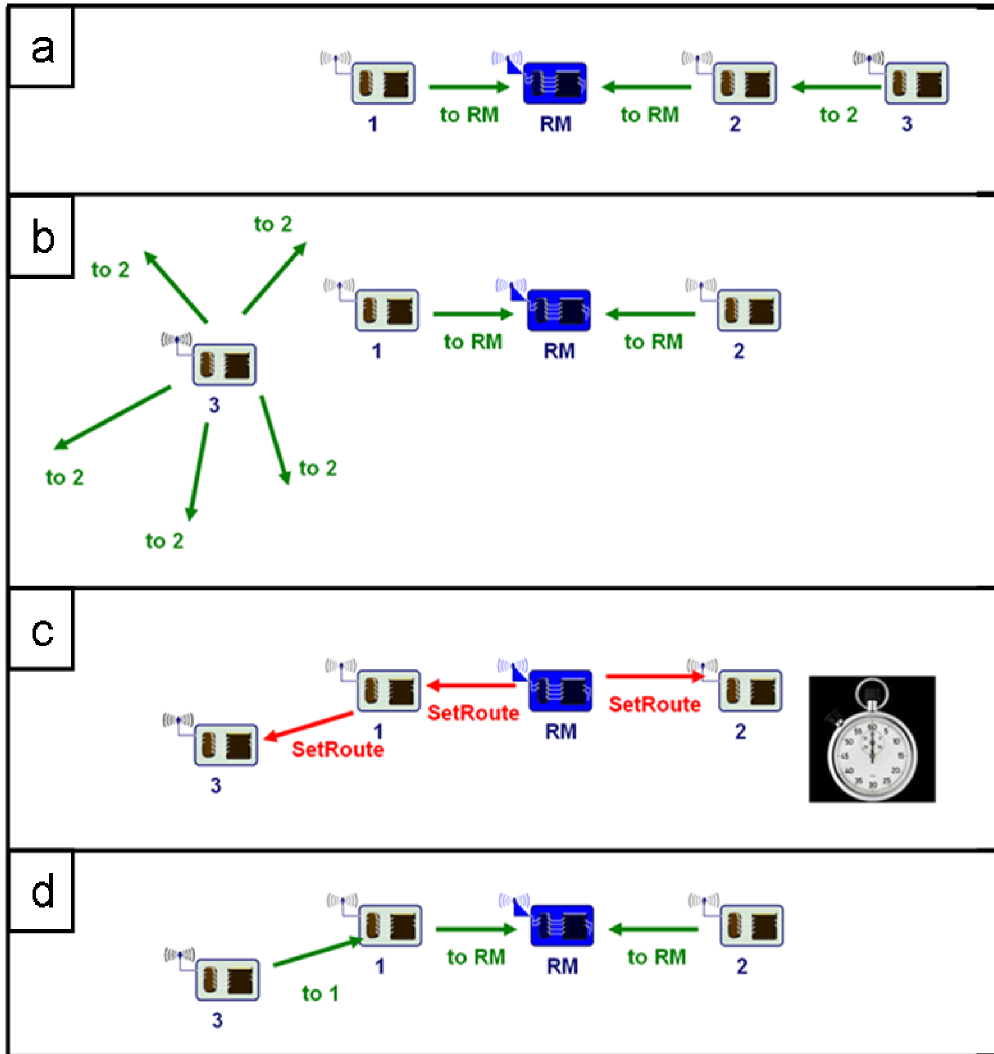

Figure 18: Conditional Route Reconstruction Enhancement Demonstration

In the above figure, the initial state of the smart space network can be seen in Figure 18.a. Three nodes report their presence: Node 1 and 2 send their messages directly to RM whereas Node 3 sends its message to Node 2 and it is redirected to RM since Node 2 already knows how to reach to RM. In Figure 18.b, Node 3 is moved within the vicinity of the smart space network; however it can not reach to Node 2 anymore. Node 3's heartbeat messages stamped to be picked up by Node 2 get lost. Eventually the node entry for Node 3 in RM times out and RM triggers route reconstruction, by flooding *SetRoute* messages (Figure 18.c). Every node reconstructs their route to the sender of the *SetRoute* message, which is RM. As a result, Nodes 1 and 2 construct their route such that they directly send their messages to RM. The reason is that they receive the *SetRoute* message directly from RM, so RM is directly accessible. Node 3 receives route reconstruction message from Node 1, so it knows which node to send its messages if it wants to communicate to RM, which is Node 1. As the route reconstruction finishes, it is guaranteed that if a node is accessible (i.e. physically in the vicinity of the network), it has a route to RM. Therefore, Node 3 healthily reports to RM again and it is not dropped, as shown in Figure 18.d.

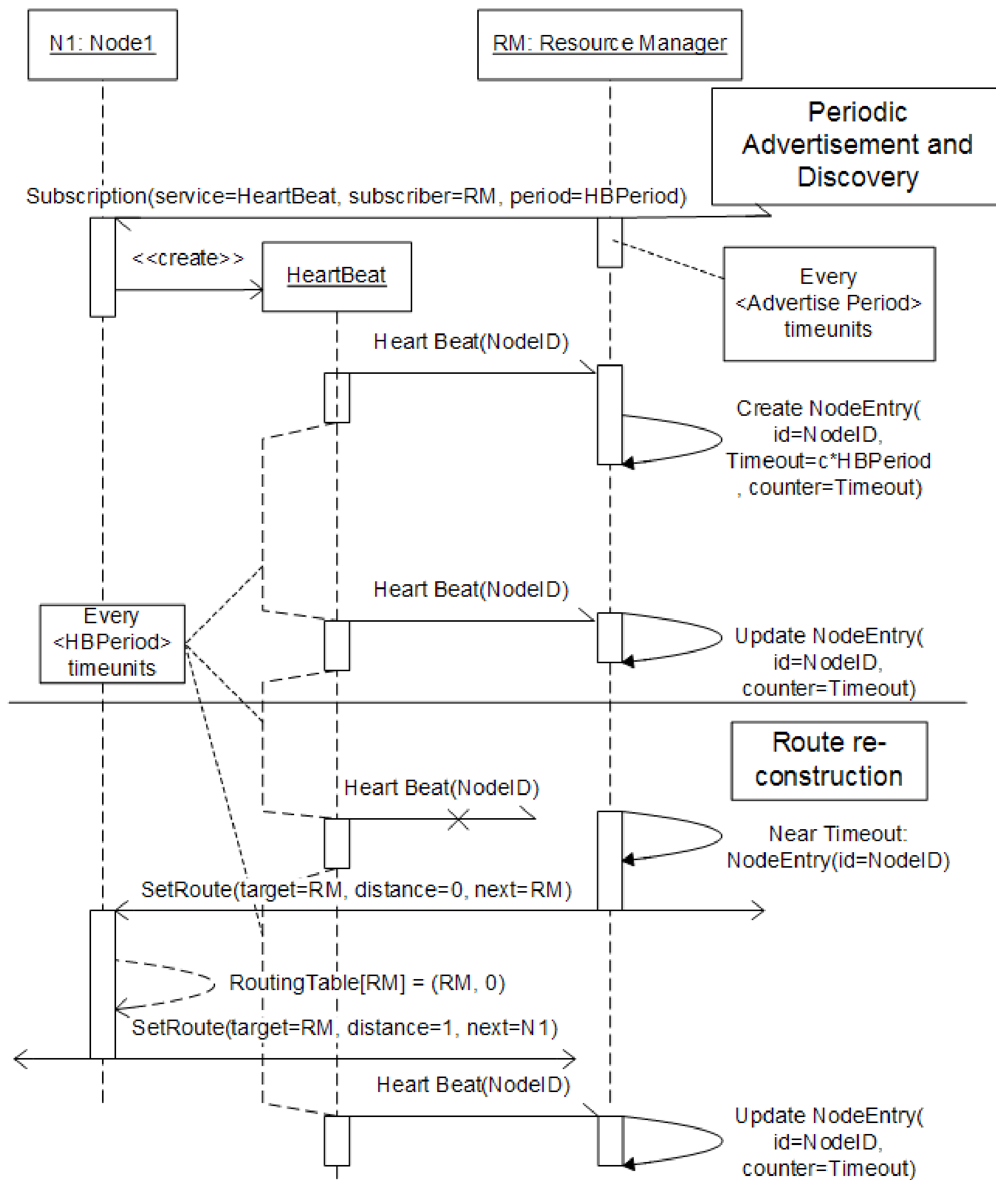

Figure 19: Sequence Diagram Demonstrating Periodic Advertisement and Route Reconstruction Enhancements

An example sequence diagram is given in Figure 19, demonstrating the functionality of the enhancements made. First, the heartbeat node is discovered upon periodic advertisement: it receives the subscription request message and creates the subscription instance and consequently starts reporting its heartbeat. When a certain number of heartbeat signals are missed, the node entry in RM hits near timeout state. This triggers RM to flood *Set Route* message and the node reconstructs its route to RM. In the final state, RM healthily receives heartbeat signals back again and the node is not dropped out of the network. With the enhancements made, the system exhibits true automatic discovery of nodes and support for dynamicity.

3.3.3. Discovery of Services and Resources

So far, we have discussed node discovery, i.e. detection and enrollment of a node to the smart space network. Nevertheless, service discovery covers the discovery of services and resources as well. In the process, just the detection of a node is not much of a discovery if the system does not know anything about it except that it is there and it can send heartbeat messages. Therefore, the descriptive and relevant information regarding the discovered node should be gathered as soon as a node is discovered. We draw the line between the process of detecting the capabilities of a node (in terms of its services and resources) for the first time and observing the changes to these capabilities over time (monitoring). *Service and Resource Monitoring* is examined in the next section.

Discovery of services and resources is a one-time process: As long as the node is not dropped and rediscovered, the discovery of services and resources occurs once only and it is triggered upon detection and registration of a node in the smart space network. This results in energy savings since node information is not reported periodically like the heartbeat signal. There is a special subscription mechanism for one-time-reporting: setting the subscription period to zero. This is a special mechanism implemented internally in OSAS; if nodes receive subscription requests with period indicated as zero, they execute the corresponding event generators of the service for one time only.

Ideally, RM should prompt for node information specifically from the newly joined node (unicast). However, this requires an existing route from RM to the node. So far, all heartbeat nodes have a route to RM, which is constructed automatically when RM subscribes to their heartbeat service. Nonetheless, there exists no route the other way around, i.e. from RM to the nodes, since no subscription is made from nodes to RM. For that reason, in our implementation we take a different approach: Whenever a new node joins the network (i.e. by sending heartbeat messages), RM requests node information from all nodes by flooding. Although flooding is costly and it limits scalability, in this way, there is no need to construct routes from nodes to RM. One advantage of the flooding approach is that, this enables RM to update its entries for nodes other than the newly joined node and to maintain the most recent information for all nodes. Note that, the alternative approach, i.e. reverse constructing routes, is not cheap either since the routes need to be periodically updated (i.e. reconstructed) due to dynamicity. Therefore, at this stage of development, we use flooding as a compensation solution for en-to-end messaging needs.

The sequence diagram in Figure 20 depicts the flow of the proposed smart space discovery mechanism. First, the node is discovered as it reports its heartbeat for the first time. Upon creation of a node entry in RM, the node information is prompted as a one-time subscription. Heartbeat node reports its node information to RM and the related node entry is updated. Note that we abstract what is being reported as “node information”. The details regarding what specifically is reported and what is stored in RM node entry objects, is the subject of the following section 3.4 *Service/Resource Monitoring*.

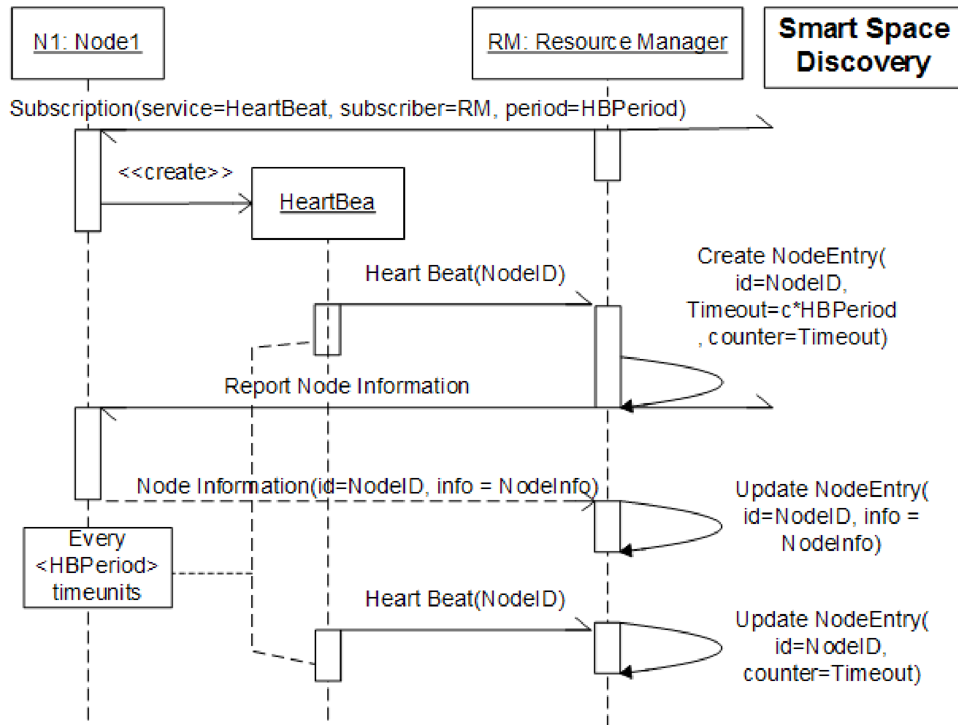

Figure 20: Sequence Diagram Demonstrating Discovery of Node and Node Information

3.4 Service/Resource Monitoring

Service and resource monitoring is used for maintaining consistent information concerning the utilization of services/resources during run-time. This capability is built upon service discovery as services/resources need to be recognized and registered first, in order to keep track of their utilization. In this project, the goal of monitoring is keeping RM 'aware' of the states of the network nodes, their services and their resources over time.

Regarding smart space nodes, we define three distinct types of information that a node holds which are relevant and necessary for monitoring:

1. *Node Type and Services:* This group of information refers to unchanging or slowly changing information over time. From a smart space perspective, a node is of a permanent type and possesses well defined services which are not likely to change in due course.
2. *Subscriptions to Services:* This group of information refers to dynamically changing service utilization on a node, i.e. subscriptions. In the network, nodes interact with each other or with RM and all these interactions are in forms of subscriptions to services.
3. *Resources:* This group of information refers to dynamically changing resource utilization on a node. As the services are seized, this reflects on the resource usage. Utilization of resources makes a key decision factor on assigning service requests to nodes.

3.4.1 Handling of RM Node Entries

Defining node information types, we have a clear picture of what precisely needs to be stored in a node entry object in RM. Let's have a closer look to a node entry in Figure 21.

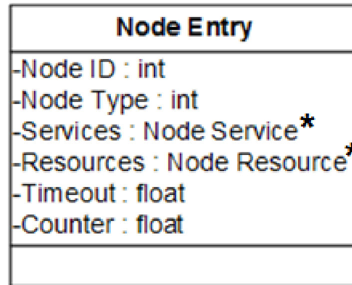

Figure 21: Node Entry Object Overview [1]

In the above figure, we see the data composition elements of a node entry object with their data types. *Node ID* is an integer value which uniquely identifies a node in the whole network. *Timeout* and *Counter* values are floating point numbers since they keep precise timing values. These three values are maintained by the *Heartbeat Protocol* which is discussed in the previous section, *Service Discovery*.

The information regarding monitoring of the nodes are stored in the remaining fields: *Node Type, Services* and *Resources*. *Node Type* is an integer value which is uniquely identified throughout the network. *Services* and *Resources* are collections of defined data structures: *Node Service* and *Node Resource*.

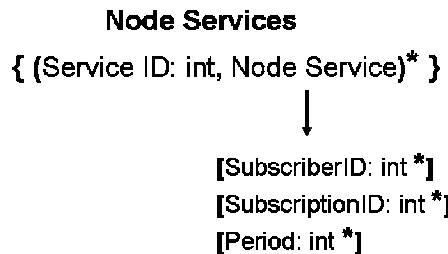

Figure 22: Node Services Data Structure [1 2 3]

Node services collection (Figure 22) is a hash map data structure containing key-value pairs of (*Service ID*, *Node Service*). *Service ID* is an integer and it is a unique identifier for a service in the network, due to OSAS implementation of symbol table file, which is updated and used by the compiler (Please refer to *3.1.5. OSAS Toolchain*). *Node Service* is a wrapper class formed of three lists of integers denoting the ids of the subscribers, the ids of the subscriptions and the periods of the subscriptions. The lists are the same length, and the same index value for all three lists refers to the same *subscription*. Together with the *Service ID* (key of the hash map), values retrieved from these lists at the same index form a *schedule*.

---

[1] ***** The asterisk quantifier indicates there are *zero or more* of the preceding element, the same as in regular expressions.

[2] **{ }** The curly braces represent hash map data structure.

[3] **[ ]** The square brackets represent list data structure.

Schedule

Service ID, Subscription ID, Subscriber ID, Period

Figure 23: Schedule Data Structure

We define a schedule as an information unit regarding a subscription. A schedule represents a single subscription with all the relevant information (Figure 23). This abstract data structure is critical for management of services.

Node Resources

{ (Resource ID: int, Resource Value: int)* **}**

Figure 24: Node Resources Data Structure [1] [2]

Node resources collection (Figure 24) is a hash map data structure containing key-value pairs of (*Resource ID*, *Resource Value*). The *Resource ID* uniquely identifies a resource type over the network. This mechanism is not implemented by default in the symbol table file of the OSAS platform, so we manually drafted a convention for numbering and assume that nodes are aware of this convention. It is also worth mentioning that different type of nodes may possess different kinds and number of resources.

3.4.2 Reporting from Nodes Perspective

Nodes, unfortunately, do not have high level constructs to store and manage information. We only assume they have basic introspection (i.e. self-examination) capabilities so that the information is accessible. The question is: How to push information in OSAS network message packets so that the nodes can communicate to RM properly?

According to OSAS framework's "go-as-you-pay" principle, it is desirable and advised to avoid delivering any bit of information which does not need to be delivered. Therefore, messaging overhead must be reduced, ensuring lightweightness. OSAS have room for manually applying this principle in messaging, and we follow it in reporting information on the nodes side.

Note that in all three following message formats, *NodeID* is included. This is essential for RM to recognize from which node the message is sent so that it can update the corresponding node entry. It is also worth to mention that all values sent in the messages are basically integers (Please refer to section *3.1.4 Messaging and Communication*). Therefore, messages can be treated simply as a sequence of integers.

The message format of reporting *node type and services* is as follows:

HandlerID	NodeID	NodeType	ServiceID1	ServiceID2	…	ServiceID n

The message starts with the *handler id* of the receiver side (RM) which denotes the event handler to process the message (*ServicesReportHandler). NodeID* and *NodeType* follow. The message is concluded with the *Service ID*s of numerous services the node possesses. Since the number of services id not known beforehand, they are at the end of the message. The receiver side unpacks

[1] ***** The asterisk quantifier indicates there are *zero or more* of the preceding element, the same as in regular expressions.

[2] **{ }** The curly braces represent hash map data structure.

the message and reads integer values one by one till the very end.

The message format of reporting *Subscriptions to Services* is as follows:

HandlerID	NodeID	Schedule1	Schedule2	…	Schedule n

The message starts with *Handler ID*, of *SubscriptionsReportHandler*, and *node id* of the sender node. In this case, the *schedule* items are at the end of the message since their number is unknown beforehand. Schedules are basically formatted as sequences of four integers. Their order is: *SubscriptionID, ServiceID, SubscriberID* and *Period*. As a result, the receiver side (RM) unpacks the schedule items as reading four integers at a time, till it reaches the end of the message.

The message format of reporting *Resources* is as follows:

HandlerID	NodeID	Resource1	Resource2	…	Resource n

At the start reside the *HandlerID,* of *ResourcesReportHandler*, and *NodeID* of the sender node by default. As in the previous case, this time *resource* items are appended to the end of the message because their number is variable. Every resource item is a sequence of two integers: *ResourceID*, *ResourceValue*. Therefore, resource items are read as two integers at a time.

3.4.3 Reporting Intervals & Triggers

What is essential for monitoring is to decide on what kind of information to report in what frequency or conditions. If the information on RM is outdated, user experience will be diminished due to wrong or late information and hence wrong decisions in management. On the other hand, if the information is reported too frequently, precious energy is wasted. The aim is to balance this tradeoff and achieve a system spending the least energy while providing required functionality.

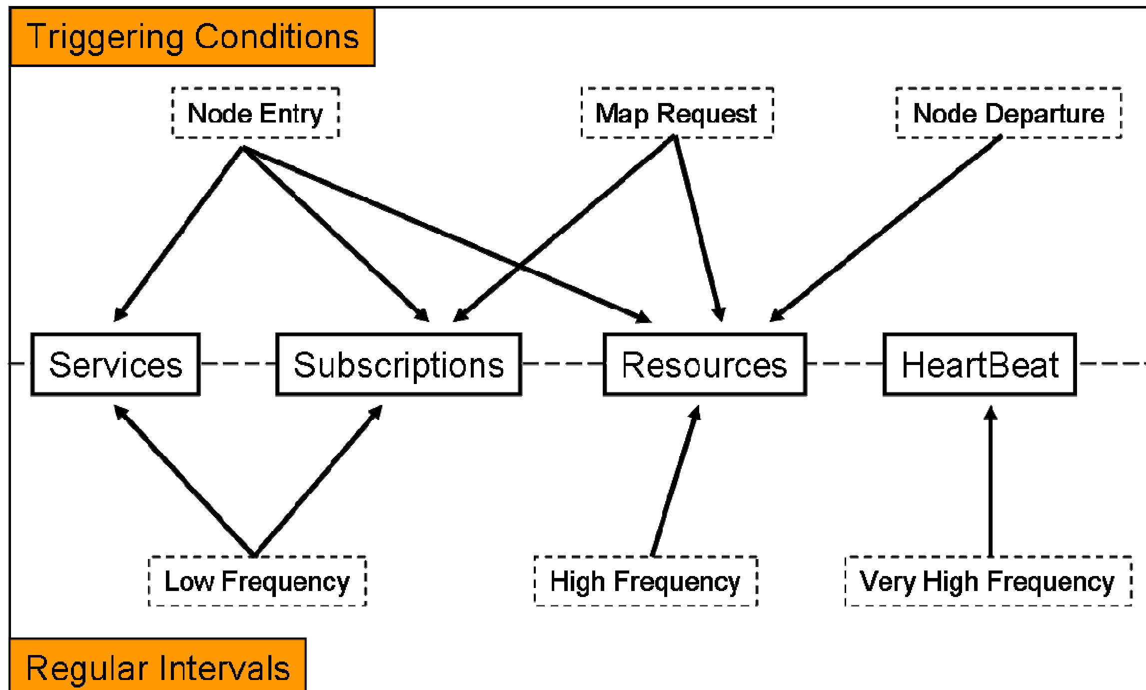

Figure 25: Reporting Conditions and Periods for Reporting Information

In fact, all three types of information for reporting as we describe (i.e. *Node Type and Services, Subscriptions to Services,* and *Resources*) along with *HeartBeat* signal have their distinct characteristic and they need to be reported on different periods and on different triggering conditions. Figure 25 illustrates how to classify these periods and conditions, and how to match them to types of information for reporting.

We see that, except for *HeartBeat*, all the information regarding a node should be reported once as the node joins the smart space network. This is done for discovery purposes.

For *Services*, reporting is done on a low frequency interval as well. This is because we assume node type and its possessed services are very unlikely to change over time: A smart space node is a dedicated node designed for a specific purpose (e.g. play mp3 audio). This assumption does not hold for pure WSN concept, since WSN nodes are more likely to be reprogrammed during their operational lifecycle.

The *Subscriptions* information is as well reported on low frequency because we assume that subscriptions on a node rarely change beyond control of RM. We have a new concept of *Map Request* depicted in the figure and it actually represents mapping of external requests to provide services to selected nodes which is carried out by RM. This subject is covered in detail, in the next section of *Management*. What we need to know at this point is whenever a service is utilized on a node thorough mapping of RM, the subscription information changes and it has to be reported for update purposes.

Node *Resources* are reported at high frequency intervals because this information is very dynamic. As an example, the remaining energy of a node battery is ever changing and it is crucial information in order to take management decisions. Moreover, resource utilization rates change as subscriptions are made or removed. This condition occurs on mapping of service requests on nodes, therefore resources are reported on *Map Request* events as well. One last condition to report resources is node departure. This is important because whenever a node leaves the smart space network its duty will be redirected to other nodes. This decision is best made with fresh information regarding resources.

The *HeartBeat* signal has to be reported with very high frequency and on a regular interval, because the information regarding the presence of nodes needs to be fresh. As an example, from a user perspective, it is intolerable if a node seemingly registered in the repository does not respond a service request because it has already left the smart space network.

Note that Figure 25 explains the reporting mechanism but not exactly how it is implemented. The conditional triggers are one time subscriptions, by their nature. On the regular intervals side, HeartBeat subscription is generated upon advertisement of RM and it goes periodic with very high frequency. However, the low and high frequency reports are not periodic in implementation; they are one time subscriptions. Rather, RM subscribes to its own services (one for low and one for high frequency) for periodically sending one-time subscription messages to nodes. This way, more energy is consumed as the nodes receive this one-time subscription message every time and report once instead of just reporting periodically. This is a decision we make, for supporting high dynamicity at the cost of energy; with every subscription, the routes from the nodes to RM are reconstructed. Moreover, the system will be less affected from message losses in the long run because subscription messages are re-sent periodically in case of a loss.

3.5 Management

Within the scope of this project, management stands as the final destination which is built on top of discovery and monitoring. In the very simple form: These two prior functionalities aim to collect information; management aims to use that information for the system to exhibit smart space behavior externally as it interacts with its users.

In this work, management functionality is aimed to be implemented at a basic level (i.e. *service mapping*), as a proof of concept. Theoretically, the management system could be further extended to a much more complex degree, since the information about the nodes are already stored in RM in a structured and consistent way. As long as the information is there, using it to achieve any complex functionality is a matter of time and programming effort.

3.5.1 Service Mapping

We model an external service mapping request as a one-time subscription to RM's *SmartSpaceSubscriber* service. Therefore, a service request could be loaded into the network (by the loader) and be routed to RM (by content based addressing). As the subscription message is received, the event generator of *SmartSpaceSubscriber* service will be triggered: executing the internal service mapping mechanism of RM.

There are essentially four parameters needed in the subscription context of the *SmartSpaceSubscriber* service:

- The service requested
- The subscriber node type
- The provider node type
- Period of subscription

The job of RM is to search for two nodes which would fit to satisfy the service request: a provider and a subscriber. These nodes have to be of the corresponding type as stated in the parameter list. In addition, the provider node has to possess the specific service type requested. Subsequently, RM executes two system calls with relevant parameters in order to select 'best' candidates for a provider and a subscriber. These system calls are:
MatchProvider(provider_type, service_id) and *MatchSubscriber(subscriber_type)*.

These system calls first filter the nodes registered in the repository which fit the requirements. Afterward, they select the 'best' candidate among these. This selection procedure can be implemented using the already existing information regarding the number subscriptions (utilization of service) and the utilization of resources. In our implementation, as a proof of concept, the selection is made according to the remaining energy levels of the nodes; the node with the most remaining energy is selected. This is simple and effective mechanism which prolongs overall network lifetime if there are many candidates for providing requested services.

As the best candidates for a provider and a subscriber are retrieved, RM needs to actually map the service request to these two nodes. Since there does not exist any route from RM to other nodes, the only way to contact with a node is to flood a message with a content based address that would fit the target node(s). Therefore, RM floods a message specifying the provider id in the content based address, to the selected provider node and invokes its *SubscriptionHandler*, with parameters being the selected subscriber node, the requested service and the requested time

period. This action mimics a straightforward subscription request, as if the subscriber node sends a subscription request to the provider node itself. Flooding of the mapping message is costly; however, without constructing reverse routes this is the only way to do the job using the default routing mechanism implemented in OSAS. Including this case, the general problem of unicast messaging is overcome with the introduction of *Path Based Routing* which is presented in the next section *3.5.2 Path Based Routing*).

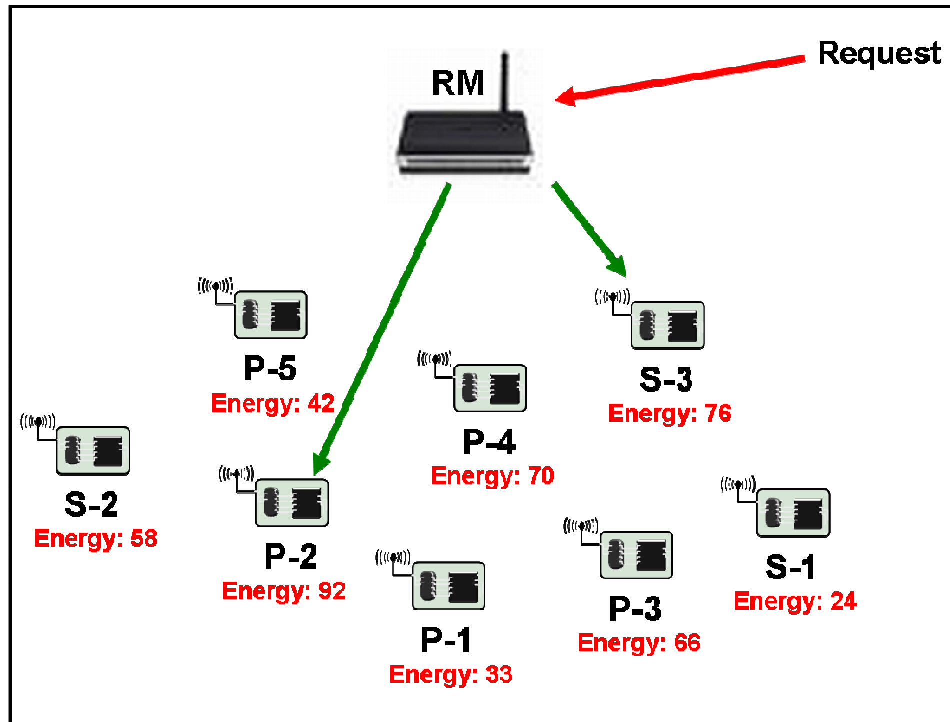

Figure 26: Service Mapping Example

An example scenario of service mapping is depicted in Figure 26. An external request is loaded to the network with parameters as node type *P* as provider and node type *S* as subscriber. The period and specific service are not mentioned in order not to loose focus. RM filters nodes for a provider and comes up with a set of all P-type nodes. Among them, it selects *P-2* as it has the most energy remaining compared to the others. The same filtering and selection is made and *S-3* is selected as it has the most energy among the candidates (all S-type nodes). Subsequently, RM matches these two nodes by subscribing *S-3* to *P-2* via invoking *P-2*'s subscription handler.

Implementing service mapping this way, however, assumes that there is an existing route in between the provider and the subscriber. This is because invoking *SubscriptionHandler* does not automatically build a route as *SubscriptionRequestHandler* does. *SubscriptionRequestHandler* is a low level construct and it takes content based address field as a parameter, which is in bytecode format. This bytecode is generated on compilation of a WaSCoL program file (.wsp) with a subscription definition. The generated bytecode is then wrapped in a message, which is fed to the *SubscriptionRequestHandler*. This way, a route is constructed on the information provided in the bytecode. The rest of the message is actually executed on *SubscriptionHandler* with non-bytecode (integer) parameters. In order to take advantage of route construction, we need a mechanism to generate bytecode on the fly. This is a technical issue yet to be resolved. Thus, we can not establish routes with *SubscriptionHandler* even though we can generate a subscription.

In the implementation with the default routing mechanism, the nodes have a route to RM (through periodic advertisement and route reconstruction) but not to one another. RM does not have a route to any node either. As a result, two-way communication is not possible without flooding messages or implementing dummy subscriptions among nodes to auto-generate routes. In order to overcome the routing problem, flooding routing messages can be thought of as an alternative, so that every node has a route to every other node. This idea sounds firm and practical, however it is neither effective nor feasible. First of all, a wireless sensor node can not store more than a few addresses in its routing table, due to its limited memory. Moreover, even if that would be possible, bombarding the whole network with routing messages is too much costly and hinders scalability drastically. Besides, the routing bombardment shall occur frequently in order to catch up with the dynamic behavior of the smart space network. Therefore, flooding network with routing messages is not an option.

At this point, the provider node has to be able to contact (have a route) to the subscriber node, because the subscription is simply ignored if subscriber is not enlisted in the routing table of the provider. In order to overcome this problem of routing, we applied a prototype version of a new routing protocol: *path-based routing*.

3.5.2 Path Based Routing

Path based routing protocol is implemented in WaSCoL programming language by OSAS developer team. It is rather an advanced routing protocol which we integrate into our solution to overcome the routing problems we encounter. This protocol overcomes the general problem of routing when we encounter the need of end-to-end messaging. The technicalities of the routing protocol is out of the scope of this work, therefore our introduction to path-based routing in this section is be limited to conceptual description and functionality of the protocol. We rather focus on how it solves our problems in an effective way.

The routing protocol defines a root node being a central point for collecting route information about the whole network. The goal is to create a hierarchical tree representation of the network on the root node. The initiation process starts with the root node broadcasting (not flooding) a message stating that it is the parent of the receiver nodes. This message contains a phase number, stating the recentness of the initiation operation. The nodes which receive the message select the root node as their parent; as the next step, they broadcast themselves as being parent nodes. The interesting point is if a node has already selected a parent (in the same phase) it ignores the message. Thus, the selection process dissipates through the network like a wave driving away from the root node. If a node has not selected a parent with the current phase (i.e. newer phase compared to which it has selected a parent for the last time), it selects its parent as the first node from which it receives the message. This process is repeated till the whole network is covered. Afterwards, every node reports its presence to its parent and forwards reporting of their children (or grandchildren from deeper levels of hierarchy) to its parent by appending its node id in front of the report message stack. Consequently, the root node collects these messages and constructs a tree like structure of nodes reporting to it. In a new initiation process, the phase number is increased so that the nodes know they have to reselect a parent for the most recent phase.

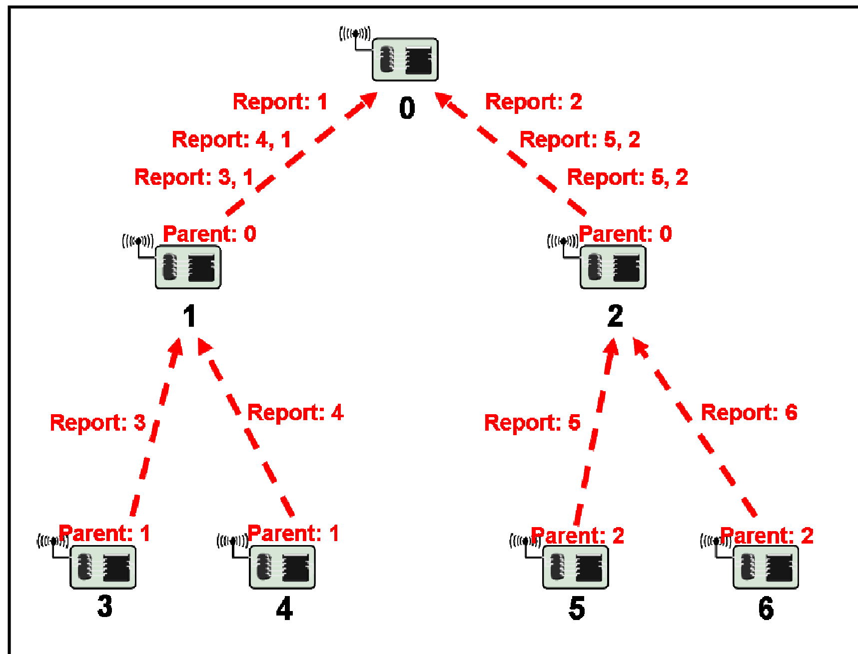

Figure 27: Path Based Routing Initiation Example

Figure 27 presents an exemplar scenario of path based routing. The root node is node 0 and it broadcasts path-based routing initiation message (with, say, phase number 1). Message arrives at nodes 1 and 2 and they select node 0 as their parent. Following that, nodes 1 and 2 broadcast a message stating their parental status. This time nodes 3 and 4 receive the message from node 1 and select it as their parent. Similarly, nodes 5 and 6 select node 2 as their parent. Eventually, the initiation ends after a determined time period and nodes report to their parents. Nodes 3, 4, 5 and 6 only report their presence as they are leaf nodes in the tree. Yet, parent nodes 1 and 2 forward their children's report to their parent, additionally. The root node 0 receives all these messages and constructs a tree out of all the branches reported. Node 0 constructs a routing table as follows: {3:1, 1:0, 4:1, 5:2, 6:2, 2:0}. The pairs represent destination and next hop, e.g. to go to node 4, the next hop is 1. The routing table presents the whole network topology. In the next path construction phase, the root node will initiate the process with an increased phase number (in this case: 2) so that the nodes will know that it is a new initiation process and reselect their parents.

Path based routing introduces a new high level messaging mechanism *SendToTarget* which is used for point-to-point communication. This mechanism exploits the routing information collected at the root node, because root node knows through which path to reach any other node in the network. If a node uses *SendToTarget* to communicate to any other node, the message created is forwarded to its parent (unless it is the target node). The parent node forwards it to its parent node (again, unless it is the target node). This goes all the way up to the root node if the target node is not one of the ancestor nodes up the branch. The root node receives the message, makes a look up in its routing table and extracts a route. It adds the complete route to the message and broadcasts it. The message is picked up by the matching node as it is on the top of route stack in the message. It removes itself from the stack and broadcasts it for the next node in the stack. The message is propagated all the way to the destination. The destination node knows that it is the destination because it is the last node on the routing stack of the message.

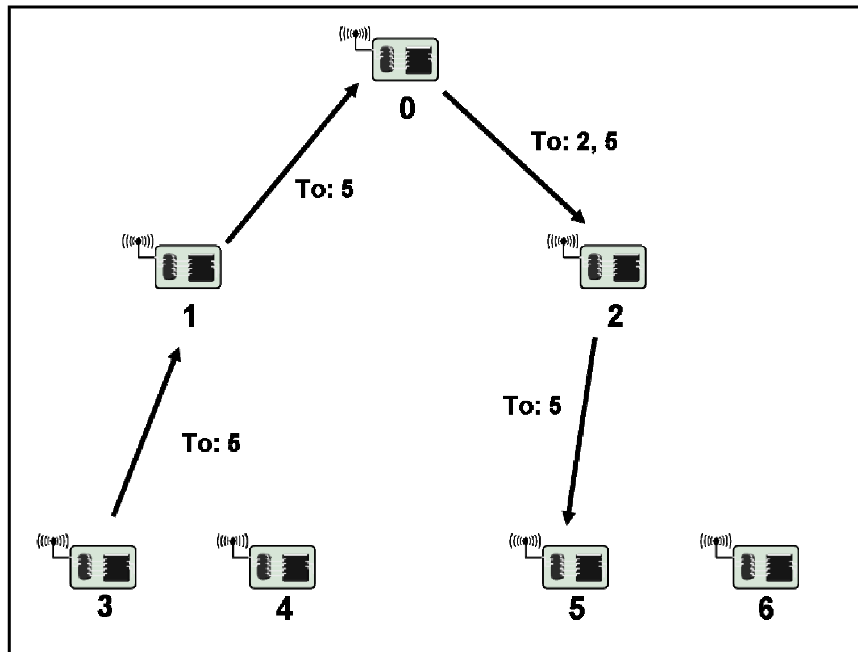

Figure 28: Path Based Routing Messaging Example

Figure 28 presents an example of messaging among nodes. Suppose that we are using the same network in the previous figure and path based routing is has been initiated as explained. Say, node 3 wants to send a message to node 5. It sends it to its parent, node 1. Node 1 checks if it is the destination, and concludes that it is not. So, it forwards the message to its parent, node 0. Node 0 is not the destination, neither. Since it is the root node, it looks up how to get to the destination and creates a route stack as being (2, 5). This is the path to the destination. It broadcasts the message and node 2 picks it up. It removes itself from the path list and broadcasts the message with route stack (5). The message is picked up by node 5 and since there is no other address on the route stack it infers that it is the destination and processes the message.

Path based routing ensures communication from and node to any other node in a sub-optimal way. A message being sent is guaranteed to travel number of hops at most two times the routing tree height. Therefore, unicast communication mechanism is quite efficient and scalable (as the travel path grows logarithmic with the node number). One more advantage is that all the nodes, except the root node, only store their parent in their routing table, which is very important for supporting lightweightness. On the other hand, there’s much burden and responsibility on the root node because it stores all the routing information and so many messages are forwarded through it.

We implemented path-based routing, RM being the root node. RM has no limitations; therefore it is desirable to draw on as much burden as possible from the nodes to RM. Moreover, it is more reliable with communication (having stronger transmission power with messaging) and persistence in the network. In this vein, path based routing perfectly fits to our aim. Beyond being the central point of node information, RM becomes the central point of routing as well.

In our implementation, we have upgraded RM’s periodic advertisement with path-based routing initiation. In its new form, advertisement service both reconstructs the routing tree and sends subscription message to all nodes at the same time by sending a single broadcast message.

Most importantly, path-based routing solves the general problem of end-to-end messaging we encounter. We replaced previous inadequate solution of flooding with path-based routing solution in corresponding points of implementation. With the new solution, only the node which enters the system is prompted for information instead of all the nodes. Service mapping is also taken care of, since any two nodes in the network can effectively communicate to each other through *SendToTarget* messaging. As a result, the provider properly provides its service to the subscriber upon service assignment is made by RM. In fact, all *SendToSubscribers* messaging is replaced by unicast *SendToTarget* messaging in the implementation, as the service assignment and hence providing are carried out on 1-to-1 basis (1 subscriber subscribes to 1 provider). As a downside, content based addressing capability is sacrificed.

3.6 Putting It Altogether

The top level architecture diagram of developed lightweight smart space solution with service, node and resource discovery/monitoring/management capabilities is depicted below in Figure 29.

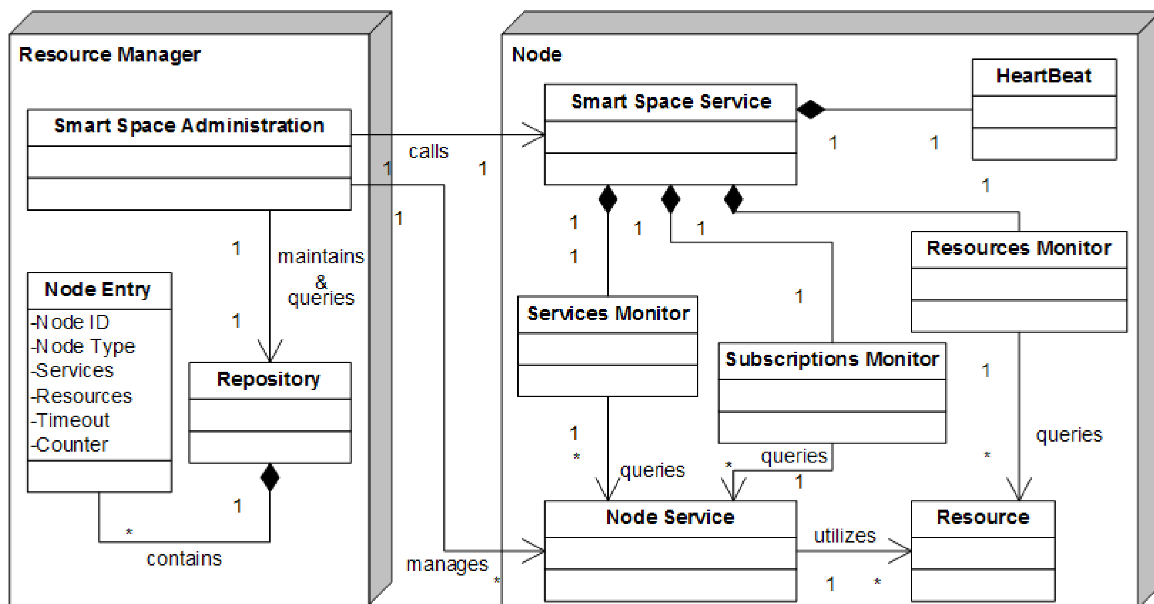

Figure 29: Lightweight Smart Space System Architecture Diagram

RM is composed of a smart space administration unit and a repository. The administration unit deals with all tasks of discovery, monitoring and management. The repository is a data structure which is responsible for keeping relevant information regarding the nodes in the smart space network. Repository is composed of node entry objects, each representing information about a node, i.e. node id, node type, the node's resources and services. Smart space administration unit has two types of interactions with nodes: *i)* it calls nodes' smart space services and hence, maintains the repository by updating information, *ii)* it queries information from the repository in order to take management decisions and manages the node services remotely.

A smart space node has a smart space service unit which is composed of heartbeat, services monitor, subscriptions monitor and resources monitor. The heartbeat service is used for discovery purposes. Introspection services query and report information regarding node services, subscriptions to these services and resources.

On the nodes side, the lightweight smart space solution architecture is basically an implementation using the OSAS framework. The proposed solution does not modify or extend OSAS architecture; however it utilizes OSAS platform to program smart space nodes. Smart space node functionality is fully implemented in WaSCoL and can be deployed on physical nodes (i.e. WSN nodes) for testing, assuming that nodes have system calls for introspection into their services and resources.

On the other hand, RM software is implemented in the development environment of the simulator unit of the OSAS toolchain, for the reasons of hardware and software development infeasibilities[1]. Numerous system calls and handlers are introduced which are artificially developed in a high level programming language (Python), not in WaSCoL language. This means that RM functionality can not be deployed on a physical node, but can be simulated on a virtual node.

Nevertheless, the interaction dynamics of the system, namely all communication in between RM and smart space nodes, is implemented in WaSCoL. This means that, as a simulated node, RM can communicate to physical nodes and carry out it duty. This makes the system eligible for testing as a whole on the field: deploying smart space node functionality on real nodes and RM on a simulated node running on a computer. This is discussed in more detail in Chapter 4.

[1] Please refer to section *4.2. Deployment on Physical Nodes*

4. Experimental Results

In this chapter, we give details about testing our implementation both in OSAS simulator and on physical nodes. We portray a simple scenario to test our implementation with adjusted parameters. We present a footprint of our implementation in terms of memory allocation (i.e. RAM and ROM usage) and messaging (i.e. message size and number).

4.1 Simulation Environment

We used the simulation environment of OSAS for rapid prototyping and testing our implementation during development phase. The simulator essentially mimics a given network scenario with defined number and placement of simulated nodes in a virtual space. Moreover, simulator provides a high level language (Python) environment to define nodes of different types and functionalities, for testing. System calls and event handler functions can be implemented in Python in a relatively fast and easy way. This is because Python provides high level abstract data types (e.g. hash maps) and object oriented programming (example use: node entry objects). Nodes specified in Python are fully compatible with the system architecture as they are programmed by compiled bytecode of a network programming file (.WSP file written in WaSCoL). They mimic real nodes in behavior (e.g. by using the same messaging protocol) and it is even possible to program a network consisting of both simulated and physical nodes interacting with one another. Overall, simulated nodes are indistinguishable from the physical nodes in terms of their behavior.

In order to work with simulated nodes properly, we need a user interface where we can observe the behavior of the network when we run the simulator. The latest version of OSAS simulator provides a graphical user interface with many capabilities. It has a tabbed view pane with three tabs for different purposes:

- *Console Tab* provides a Python interpreter which can be used for diagnosis of the network in run-time. There are numerous pre-defined commands for a quick introspection of nodes in the network (e.g. listing node services). Run-time errors are redirected to console, as well.
- *Message Log Tab* displays detailed view of messages sent over the network, in run-time. The messages can be filtered to focus on and analyze a specific node's behavior.
- *Canvas Tab* provides a canvas of 2-D bird-sight view of the simulation space. Simulated nodes appear on the canvas; their type and location in the space are denoted by the scenario description file (.WNL). The canvas is interactive: the nodes can be dragged within the canvas space by mouse, reflecting on the network topology. There is a textbox in this tab which is used for printing data by invoking *PrintHandler* of a node. *PrintHandler* is an event handler introduced for testing purposes in the simulation environment; only simulated nodes can have this handler for use. For visual feedback, the key tab is the canvas tab in the current simulator, since the other two tabs are mostly for diagnosis and debugging.

Screenshots of simulator tabs can be found in the Appendix.

The simulator GUI is useful for general purpose testing of applications. Nevertheless, we look for a more advanced GUI which is tailored for our smart space implementation. Actually, we would like an interface to RM: both to observe the network state and to request for services. We seek to implement the following features:

- Visualization of the network in such a way that RM and smart space nodes can be identified. The nodes participating in the network can be distinguished from the ones which are not.
- Display of information regarding dynamic structure of the network. This includes depicting real-time composition of the network as well as notifying which nodes join or depart.
- Display of repository of RM.

For the reasons stated, the simulator GUI is enhanced with an additional tab, the *Smart Space Tab*, as a part of this thesis project. The demonstration of this enhancement with a screenshot is combined with an exemplar scenario presentation in the next section.

4.1.1 Example Scenario

Figure 30 demonstrates a screenshot of the Smart Space Tab, running a example scenario. The Smart Space Tab has a grid pane to the left, which visualizes the events printed on the canvas tab. In this example, there are six nodes simulated and displayed on the grid, with randomly assigned node ids. In the simulator, the nodes can be clearly distinguished by their colors as they are color coded: RM node is shown in blue color, whereas the nodes which are presently in the network and nodes which are out of the network appear in light green and gray colors, respectively. In Figure 30, the nodes in the screenshot image are also shape coded for readability purposes. RM node (node 45) is drawn as the blue circular shape. The nodes 35, 38, 41 and 43 are in the smart space network as they are light green and of triangular shape. The signaling distance is up to four squares on the grid; consequently node 32 is out of the network range and it is shown as the gray colored square shape. On the right side of the view, there are two text boxes. Both of these boxes are an interface to RM, where the top box prints changes to the smart space network over time and the bottom box prints the present state of the repository of RM.

Let's look into the top text box in Figure 30 by investigating its printed lines from top to bottom regarding the scenario run. Firstly, all five nodes join the network, as the simulator starts. Following that, the content nodes of the repository are printed. Afterwards, node 32 moves out of the vicinity of the network and becomes unreachable; therefore it is dropped out of the smart space network. The contents of the repository after the drop are printed, representing the latest state. Then, we see that a service request is made on the system. This is done manually by the smart space user, by subscribing to RM's "smart space subscription request service" via the loader. In the loader display, subscriptions defined are listed and they can be loaded to the network on demand. We defined *RequestTemperatureSubscription* to do the job for this scenario. Upon the service request, RM matches a provider (node 38) and a subscriber node (node 41) and maps the service accordingly. At this point, node 41 starts receiving temperature measurements and prints received values on the canvas tab's textbox (not visible in the figure). Lastly, latest repository contents are printed again.

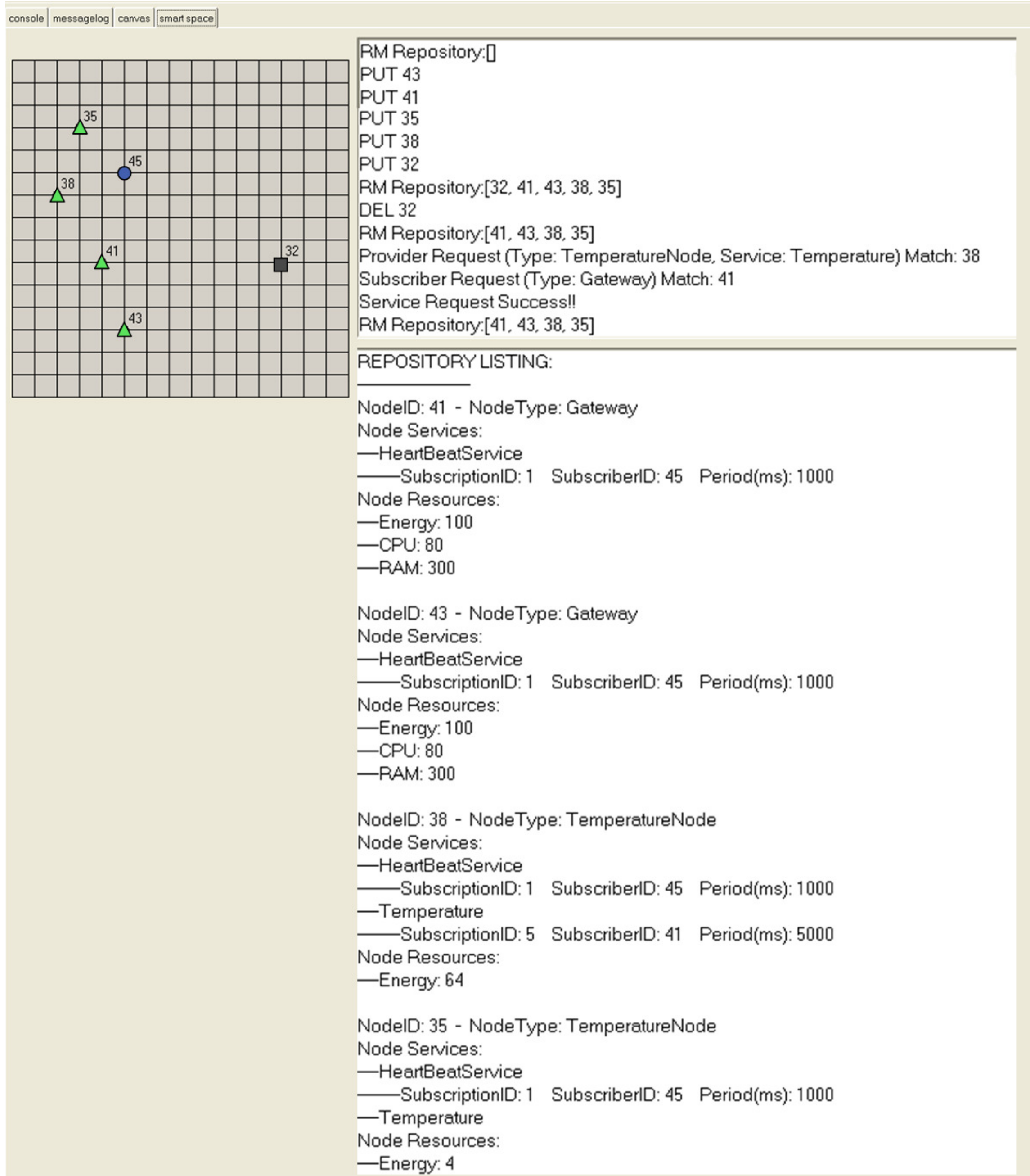

Figure 30: Smart Space Tab Extension Running a Sample Scenario

The bottom text box in Figure 30 displays detailed information about nodes participating in the network. Basically, repository contents are queried and information for each node is displayed in a specific format. The display format for a node is shown in Figure 31.

The stored values, as they are communicated by the nodes, in the repository are actually all in integers. However, for informative display purposes, applicable values are converted to strings using the symbol table file. For example, *HeartBeatService* has a unique Service ID 1 which is registered in the symbol table with its string match; so, its string value can be retrieved with a simple lookup.

NodeID: <*node id*> - NodeType: <*node type*>

Node Services:
<*Service 1*>
SubscriptionID: <*Subscription ID 1*> SubscriberID: <*Subscriber ID 1*> Period(ms): <*Period 1*>
SubscriptionID: <*Subscription ID 2*> SubscriberID: <*Subscriber ID 2*> Period(ms): <*Period 2*>
…
SubscriptionID: <*Subscription ID n*> SubscriberID: <*Subscriber ID n*> Period(ms): <*Period n*>
<*Service 2*>
....
<*Service m*>

Node Resources:
<*Resource Name 1*>: <*Resource Value 1*>
<*Resource Name 2*>: <*Resource Value 2*>
….
<*Resource Name k*>: <*Resource Value k*>

Figure 31: Node Information Display on Smart Space Tab

As seen from the bottom text box in Figure 30, all the nodes in the network, of course, have *HeartBeatService* installed and RM (node 45) has subscribed to this service on all nodes with 1 second period. In this scenario, there are two types of heartbeat nodes: *gateway* and *temperature node*. Nodes 32, 35 and 38 are temperature nodes, and nodes 41 and 43 are gateway nodes. Temperature nodes have temperature sensing capability and consequently a *Temperature* service installed on them which reports the temperature measurements to the subscribers. Temperature nodes only have "energy" resource which indicates the energy remaining on node battery. Gateway nodes have no specialized services defined and their job is to collect temperature readings from temperature nodes. They have "energy", "CPU" and "RAM" resources. The defined resources are not really measured in the simulation environment, but their values are changed over time for proving the concept. Therefore, the displayed measurement values are emulated, i.e. the energy resource decreases in a steady fashion (different per node) and the other resources (i.e. CPU and RAM) have constant values. In this example, a service mapping is made in between nodes 38 and 41, which can be seen in the top textbox. We can see that node 38's *Temperature* service is subscribed by node 41 with a period of 5 seconds.

Assigned timing parameters in the simulation run are as in Table 1.

Parameter	**Value**
HeartBeat Period	1 second
Timeout Period	6 seconds
Route Fix Trigger	3 seconds to Timeout
Max. Route Fix Frequency	Every 3 seconds
Refresh Registry Period	1 second
Advertisement Period	10 seconds
LowFrequencySubscription Period	60 minutes
HighFrequencySubscription Period	30 seconds
Triggering Conditions Check/Response Period	1 second

Table 1: Assigned Timing Parameters in Example Scenario

This scenario runs stably in the simulator with the above stated parameters. From experimental simulation runs with these settings, it can be inferred that the smart space architecture implementation meets the timing requirements stated in Chapter 2.

Number of messages sent & received per node per minute, at regular intervals, is shown in Table 2.

Sent Messages	Received Messages
HeartBeat: 60	Advertisement: 6
Report Resources: 2	Resource Report Subscription: 2
Report Services: 1/60	Services Report Subscription: 1/60
Report Subscriptions: 1/60	Subscriptions Report Subscription: 1/60
Total: ~ *62*	*Total:* ~ 8

Table 2: Message Count of Regular Interval Messages [1]

Disregarding triggering conditions, every node in the smart space network sends and receives a total of about 70 messages every minute.

There are also messages sent and received for every triggering condition. The values are shown in Table 3.

	Messages Sent	Messages Received
Node Entry	3	3
Set Route	1	1
Node Departure	1	1
Service Mapping	2	2

Table 3: Message Count of Triggering Conditions Messages [2]

We make an estimation of 3 node arrivals and departures, 2 service mapping requests and 5 set routes to take place per minute, representing an active smart space. An arbitrary node would send and receive messages occurring due to set route and node departure, which sums up to 16. Messages occurring due to node entry and service mapping are sent and received only by the corresponding nodes, thanks to the use of path-based routing mechanism.

With the assumptions we make, an idling node sends and receives roughly 86 messages per minute in total. This result is a good match to our previously stated goal, which is to keep message count less than 100 messages per minute.

With the parameter settings of Table 1, the majority of message count is constituted by heartbeat messages only; especially if the smart space network is not active (i.e. no or few triggering conditions happen every minute). In Figure 31, resulting total message numbers per minute are shown according to different parameterization of heartbeat messaging period.

1 The numbers presented in this table map to the parameter values presented in Table 1, using Figure 25 in section 3.4.3 Reporting Intervals & Triggers.

[2] The numbers presented in this table map to the parameter values presented in Table 1, using Figure 25 in section 3.4.3 Reporting Intervals & Triggers.

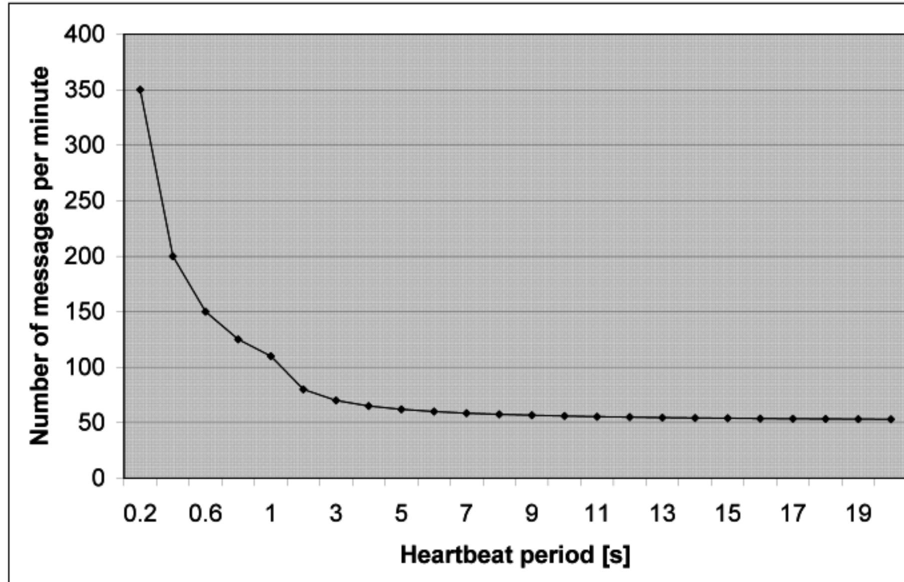

Figure 31: Heartbeat period vs. Num

Figure 31 shows, as heartbeat reporting period increases, the total number of messages sent and received by the node decreases asymptotically. Note that heartbeat period value is very decisive by itself, keeping the other parameters constant. Fortunately, along with other parameters, heartbeat period can be optimized for specific needs and uses.

4.2 Deployment on Physical Nodes

We have successfully tested our implementation on physical nodes as well. We have implemented our system on Body Sensor Network (BSN) nodes [28] (Figure 32). BSN nodes are WSN nodes which are designed for developing applications for healthcare. Focus points in BSN research are system integration, sensor miniaturization, low-power sensor interface circuitry design, wireless telemetric links and signal processing. Technical specifications of BSN nodes can be found in the Appendix.

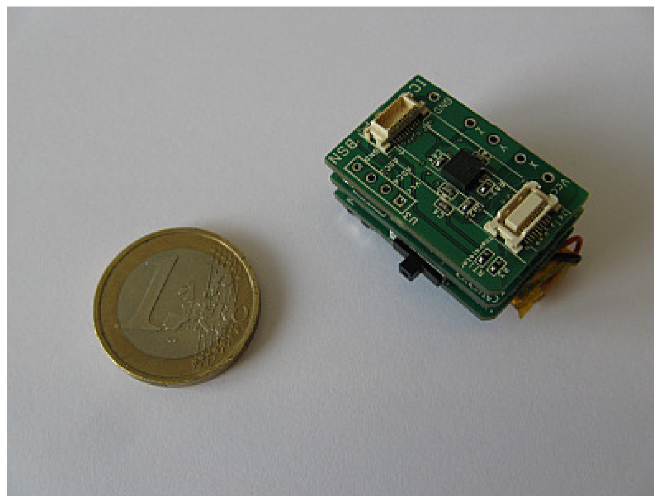

Figure 32: BSN Node Photo

Physical nodes are in fact limited in capacity and they simply do not operate properly when overloaded. In this section, memory allocation and message size calculations will be presented.

First, we present OSAS footprint on nodes without the smart space implementation, on Table 4. The current implementation of OSAS framework runs on TinyOS [27] operating system on node hardware.

	ROM (in bytes)	**RAM** (in bytes)
TinyOS (w/o OSAS)	15912	947
TinyOS (w/ OSAS)	28068	1289

Table 4: Footprint of TinyOS and OSAS

OSAS also dynamically uses RAM: 160 bytes for execution stack and 132 bytes for message buffers.

At earlier stages of development, where only service discovery was implemented, we mapped RM functionality from Python code to WaSCoL code. The idea was to be able to implement RM as an application program so that it could be programmed onto any physical node, i.e. one node in the system is assigned to be RM. The code conversion revealed two issues. Firstly, it resulted in a footprint of approximately half a kilobyte, increasing rapidly in size depending on the number of nodes and their services. This was a clear indication that RM could not be implemented on physical nodes, especially with much functionality yet to be added and implemented in WaSCoL. Secondly, mapping of code process in between two languages is neither trivial nor efficient, in practical terms; mostly due to lack of high level data constructs. Thirdly and finally, OSAS have size restrictions for event generators and handlers that do not fit to relatively complex counterparts in our implementation. For the stated reasons, it turned out that the best way to put RM into practice is to implement it as a simulated node in the smart space environment. Clearly, it is a significant disadvantage due to loss of support for RM mobility and inability to program an arbitrary node to be RM in the space. However, if we perceive the smart space to be an immobile and persistent physical environment, like an office room, running RM on a PC in the space makes it quite acceptable. This way, we bundle the physical space and RM together.

The footprint for RM is not presented because much of its functionality is implemented as system calls and handlers in Python language. As a result, we do not have the bytecode to assess footprint values.

On the smart space nodes side, the required functionality is much lighter since complexity is moved to RM node in our centralized design. This results in small footprint values, which is a core aim of this project. The extra memory allocation for a WSN node to be a *smart space node* can be expressed as the total memory allocation of the individual services stated in Table 5.

Service	**Size** (in bytes)
HeartBeatService	28 (14 ROM + 14 RAM)
ReportServicesService	30 (16 ROM + 14 RAM)
ReportResourcesService	30 (16 ROM + 14 RAM)
ReportSubscriptionsService	30 (16 ROM + 14 RAM)
Total	*118 (62 ROM + 56 RAM)*

Table 5: Memory Allocation of Smart Space Application

The smart space application is quite lightweight, even when compared to other applications programmed on WSNs: A considerably complicated healthcare application results in approximately 250 bytes of code for services. In above calculation, path-based routing is excluded because it is not a core requirement of this project and rather it is used as an enabler. We take it as a solution routing mechanism provided by the infrastructure framework, as it is not specifically designed for this project. Keep in mind that footprint calculations are made with the assumption that nodes do have system calls for introspection of information implemented. For instance, *ReportResourceService* would make use of a system call which basically returns values of the resources possessed. After retrieval of introspection values, the service can actually pack a message in a specific format and send the message. In the physical nodes which we deployed our implementation, the system calls for services and subscriptions are implemented however resource introspection is missing. Therefore, we have written a service in WaSCoL which mimics resource reporting and it is loaded into physical nodes. The point is to see RM operating properly with the resource information reported, as a proof of concept.

Messages sent for communication among the nodes are encoded as described in Chapter 3, in section *3.1.4 Messaging and Communication*. The MAC protocol and OSAS messaging protocol occupy 10 and 5 bytes, respectively, in the message header.

Message	**Size** (in bytes)
HeartBeat Report	2
HeartBeat Subscription	17
Services Report	3 + (number of services)
Resources Report	2 + 2 * (number of resources)
Subscriptions Report	2 + 4 * (number of subscriptions)
Services Report Subscription	17
Resources Report Subscription	17
Subscriptions Report Subscription	17
Service Mapping	9
Service Request Subscription	17

Table 6: Message Payload Sizes

The resulting message payload sizes for the smart space application are presented in Table 6. Payload sizes are calculated assuming eight-bit encoding. The upper bound values result with sixteen bit encoding, and are two times the corresponding values presented. For information on encoding of messages in OSAS, please refer to *3.1.4 Messaging and Communication* section in Chapter 3.

5. Conclusions and Future Work

In this work, our goal was to facilitate low capacity nodes, to form a smart space network with means to discover, monitor and manage nodes, and their services and resources. For that purpose, we extend a Wireless Sensor Network (WSN) architecture into a lightweight smart space architecture.

We implement a centralized solution, where a resource manager (RM) node is proposed to discover nodes which enter into the space, monitor node services and resources, and map service requests to selected best candidates. We use OSAS platform as a programming framework to design a smart space architecture for WSN nodes. For service discovery, we implement binary soft-state protocol, i.e. heartbeat protocol, which offers fault tolerance and decoupling through monitoring and negotiation. We propose extensions of periodic advertisements and conditional route reconstruction as enhancements for discovery. For service/resource monitoring, we define three information types for introspection: *i)* node type and services, *ii)* subscriptions to services and *iii)* resources. We classify regular intervals and triggering conditions of reporting, separately for each information type. For management, we implement service mapping that makes use of information gathered through discovery and monitoring operations, as a proof of concept. We utilize path-based routing protocol, which proposes a root node that knows the topology of the entire network and forwards communication messages in between nodes. We integrate this functionality to our RM node, providing support for lightweightness since responsibility of routing in the network is drawn onto RM, rather than the low capacity nodes.

It turned out to be infeasible to deploy RM on a physical node due to several reasons. Firstly, high capacity requirements on hardware make it practically infeasible. Secondly, mapping from the high level programming language (i.e. Python) environment of the simulator to lower level programming language (i.e. WaSCoL) of application development is non-trivial and requires inefficient use of unit constructs in order to implement high level data structures. Thirdly and finally, OSAS framework has size restrictions for event generators and handlers that do not comply with relatively complex counterparts in our implementation.

Consequently, RM was implemented in OSAS simulator environment, which redeems it from limitations. We extended the simulator GUI by implementing a user interface to RM, reflecting network state. Furthermore, we successfully deployed our ‘smart space node’ solution on Body Sensor Network (BSN) nodes. Credits to our design decisions, the required capabilities of the nodes to participate in the smart space are kept minimal. A sample scenario is run on the network composed of physical nodes and RM running on a PC. We present measured footprint values and investigate into message size and frequency.

The novelty of this work resides in having implemented a smart space architectural solution on lightweight devices. The solution is implemented purely as an application program on participating nodes; however RM needs a more advanced software and hardware platform to be implemented. The implementation of this architectural solution exhibits smart space functionalities with complete automatic operation and a firm support for dynamicity.

This project has much room for further research and development. First of all, the solution proposed is centralized; even to the point that only one RM is supported within the space. We believe pure distributed solutions are not suitable for this work; however hybrid approaches should be considered. Secondly, the monitoring mechanism can be improved by reporting only

the changes to the information, removing redundant information delivery. Moreover, instead of sending monitoring subscriptions periodically, RM might take a conditional subscription message sending, in case it misses reporting. In the current implementation, message sending periods and conditions are determined by RM, whereas these parameters could be automatically adjusted by the nodes according to their type and run-time situation. Thirdly, energy expenditure can be minimized by automatically adjusting timing parameters to the dynamicity of the network. Finally, the current solution implements management mechanism at its very basic and it lacks service recovery. Management mechanism could be improved for more complex service mapping (e.g. complex decision making, one-to-many service mapping, etc.) and arbitration for minimal resource usage.

Appendix

Simulator Screenshots

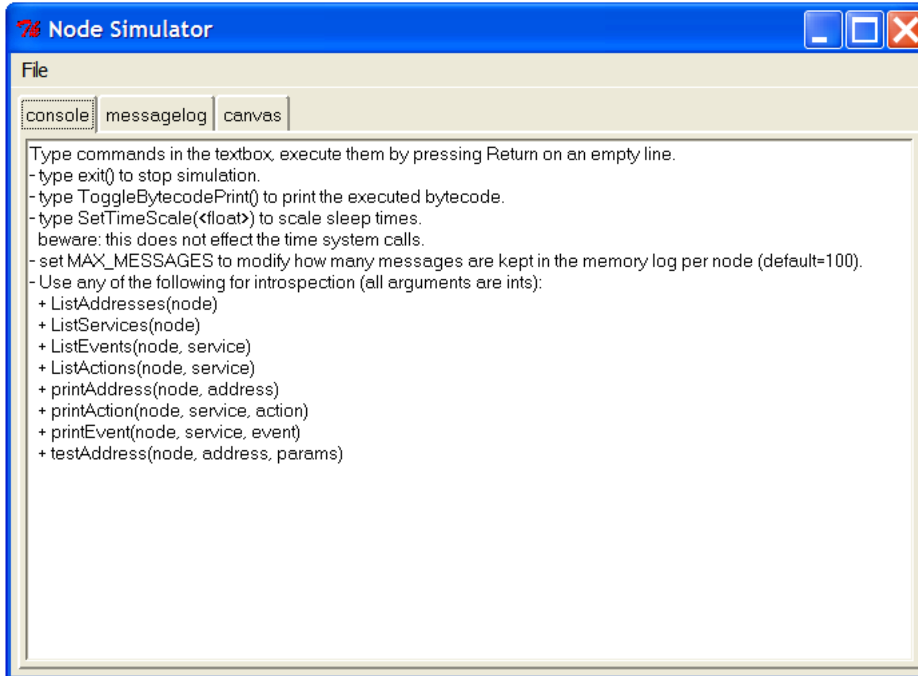

Figure 33: Simulator Console Tab

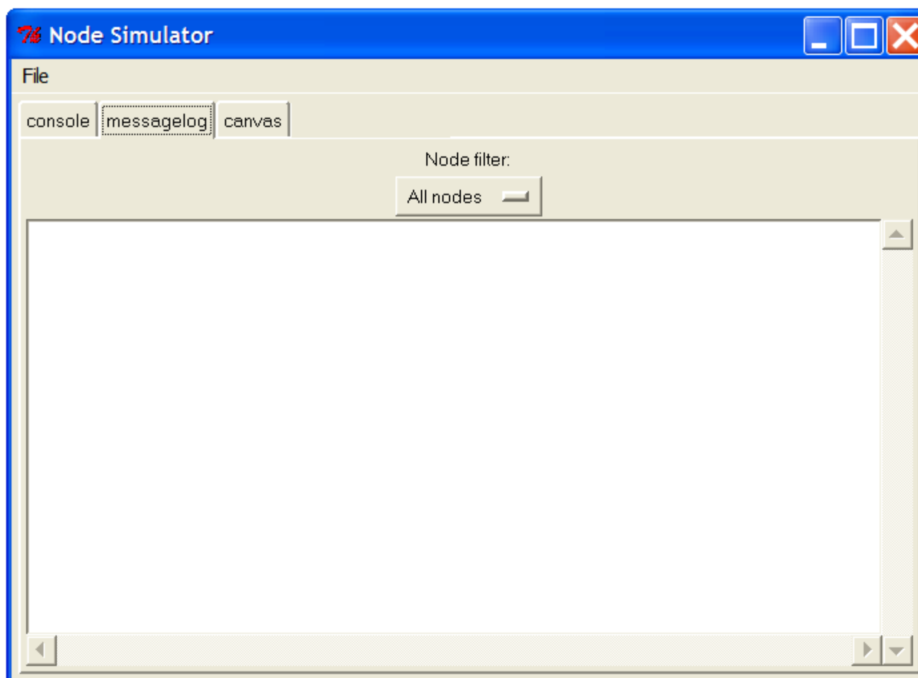

Figure 34: Simulator Message Log Tab

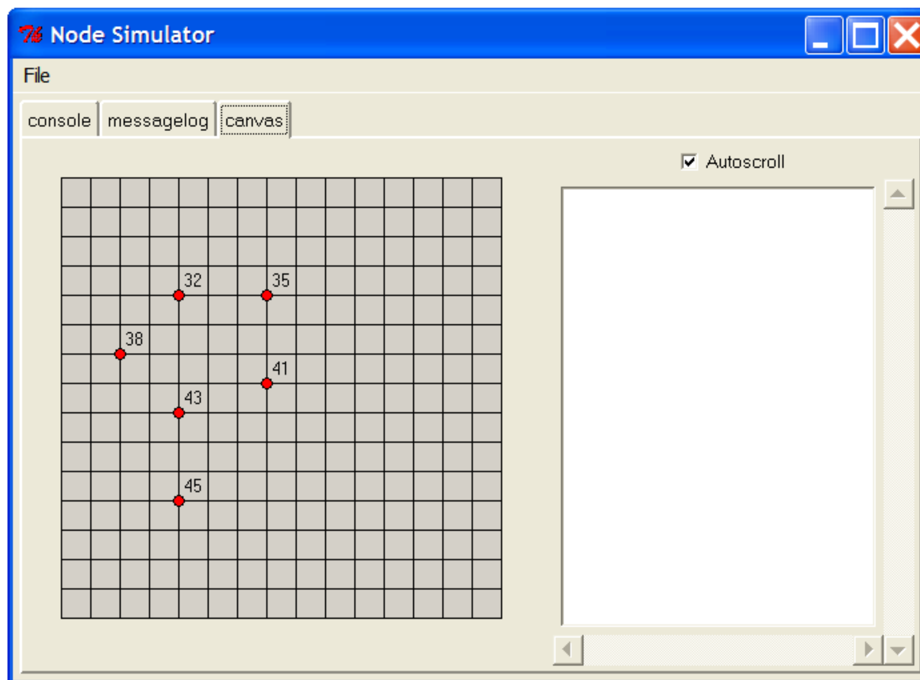

Figure 35: Simulator Canvas Tab

BSN Node Specification

Size: 19mm x 30mm
Processor: 16-bit ultra low power RISC processor
Memory: 48KB flash/ 10KB RAM
Channels: 8 channels 12-bits ADC, 2 channels DAC, 2 USART
Radio Transceiver: TI CC2420, IEEE 802.15.4 (2.4GHz DSSS)
Range: 50m (indoors), 125m (outdoors)
External storage: 4MB external EEPROM
Operating System: TinyOS

References

[1] Weiser M.; “The computer for the 21st century”; Scientific American, 265(30):94–104; 1991.

[2] Banavar G., Beck J., Gluzberg E., Munson J., Sussman J.B., Zukowski D.; “Challenges: an application model for pervasive computing”; Mobile Computing and Networking, pp. 266-274; 2000.

[3] Saha D., Mukherjee A.; “Pervasive computing: A paradigm for the 21st century”; IEEE Computer, 25–31; March 2003.

[4] Satyanarayanan M.; “Pervasive computing: Vision and challenges”; IEEE Pers. Commun., vol. 8, pp. 10–17; August 2001.

[5] Hong J.I., Landay J.A.; “An Infrastructure Approach to Context-Aware Computing”; Human-Computer Interaction, vol. 16, nos. 2–4, pp. 287–303; 2001.

[6] Dey A.K., Abowd, G.D.; “Towards a better understanding of context and context-awareness”; Proceedings of the Workshop on the What, Who, Where, When and How of Context-Awareness, New York; 2000.

[7] Dey A.K.; “Understanding and Using Context”; J. Personal and Ubiquitous Computing, vol. 5, no. 1, pp. 4-7; Feb. 2001.

[8] Van der Meer S., Jennings B., Barrett K., Carroll R.; “Design Principles for Smart Space Management”; 1st International Workshop on Managing Ubiquitous Communications and Services (MUCS), M-Zones whitepaper; December 2003.

[9] Karl H., Willig A.; “Protocols and Architectures for Wireless Sensor Networks”, Wiley Press, 2007.

[10] Brodt A., Sathish S.; "Together we are strong"; IEEE International Conference on Pervasive Computing and Communications, pp.1-4; 2009.

[11] Bardram J. E.; “The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications”; Centre for Pervasive Computing Technical Report, Aarhus, PB–61; 2003. Available at http://www.pervasive.dk/publications.

[12] Arnold K., O'Sullivan B., Scheifler R.W., Waldo J., Wollrath A; “The Jini Specification”; Addison-Wesley Press; 1999. See also www.sun.com/jini.

[13] Monson-Haefel R., Chappell D.; “Java Message Services”; O'Reilly Press; 2001. See also http://java.sun.com/products/jms/.

[14] Gu T., Pung H.K., Zhang D.Q.; “A service-oriented middleware for building context-aware services”; J. Network and Computer Applications, vol.28, pp.1–18; 2005.

[15] Gu, T.; Qian, H.C.; Yao, J.K.; Pung, H.K.; "An architecture for flexible service discovery in OCTOPUS"; The 12th International Conference on Computer Communications and Networks, Proceedings., vol., no., pp. 291-296; 20-22 Oct 2003.

[16] Kagal L., Korolev V., Chen H., Joshi A., Finin T.; "Centaurus: A Framework for Intelligent Services in a Mobile Environment"; 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01), pp.0195; 2001

[17] Efstratiou C., Cheverst K., Davies N., Friday A.; “An architecture for the effective support of adaptive context-aware applications”; Mobile Data Management (MDM), Hong Kong, Springer, pp. 15–26; 2001.

[18] Van der Meer S., O’Connor R., Davy A.; “Ubiquitous Smart Space Management”; 1st International Workshop on Managing Ubiquitous Communications and Services (MUCS), M-Zones whitepaper; December 2003.

[19] Helal S.; “Standards For Service Discovery And Delivery”; IEEEPervasive Computing, 1:95.100; July-Sept 2002.

[20] Cummins S., O’Grady J.P., O’Reilly F.; “Managing Wireless Ad-hoc Networks, IP Addressing and Service Delivery”; 1st International Workshop on Managing Ubiquitous Communications and Services (MUCS), M-Zones whitepaper; December 2003.

[21] Henricksen, K., Indulska, J.; “A Software Engineering Framework for Context-Aware Pervasive Computing”; Second IEEE International Conference on Pervasive Computing and Communications; IEEE Computer Society, pp. 77–86; 2004.

[22] Chen H., Finin T., Joshi A.; "An ontology for context-aware pervasive computing environments"; The Knowledge Engineering Review, Volume 18, Issue 03; 2003.

[23] Baldauf M., Dustdar S., Rosenberg F.; “A survey on context-aware systems”; International Journal of Ad Hoc and Ubiquitous Computing, 2(4):263-277; 2007.

[24] Bosman R., Lukkien J.J., Verhoeven R.; "An Integral Approach to Programming Sensor Networks"; Consumer Communications and Networking Conference; 6th IEEE , vol., no., pp.1-5, 10-13 Jan. 2009. See also http://www.win.tue.nl/~rbosman/publications.html.

[25] Tjiong M., Lukkien J.J.; “An Investigation Into Soft-state Protocol Parameters”; Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2008 ; July 14, 2008.

[26] Van Rossum G.; “The Python programming language”. Available at www.python.org

[27] Levis P., Madden S., Polastre J., Szewczyk R., Whitehouse K., Woo A., Gay D., Hill J., Welsh M., Brewer E., Culler D., “TinyOS: An operating system for wireless sensor networks,”; Ambient Intelligence, Springer-Verlag; 2005.

[28] Lo B., Thiemjarus S., King R., Yang G., “Body Sensor Network - A Wireless Sensor Platform For Pervasive Healthcare Monitoring”; The 3rd International Conference on Pervasive Computing; May 2005. See also http://www.bsn-web.org/.

[29] Wirelessly Accessible Sensor Populations Project. See http://www.wasp-project.org/.