Performance Comparison of Multi Agent Platforms in Wireless Sensor Networks.

Master’s Thesis in Embedded and Intelligent Systems

Bernhard Bösch
Performance Comparison of Multi Agent Platforms in Wireless Sensor Networks.

Master’s Thesis in Embedded and Intelligent Systems

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden
Preface

I would like to thank everyone who supported me during the work on this thesis. Without their experience and help this thesis would not have been possible. First of all, I would like to take the chance to state my highest gratitude to my supervisor Edison Pignaton de Freitas for his advice and great support. In addition to that I would also like to thank my family and friends, who have encouraged me during the strenuous time of writing.

Bernhard Bösch

Halmstad University, November 2011
Abstract

The technology for the realization of wireless sensors has been available for a long time, but due to progress and development in electrical engineering such sensors can be manufactured cost effectively and in large numbers nowadays. This availability and the possibility of creating cooperating wireless networks which consist of such sensors nodes, has led to a rapidly growing popularity of a technology named Wireless Sensor Networks (WSN). Its disadvantage is a high complexity in the task of programming applications based on WSN, which is a result of its distributed and embedded characteristic. To overcome this shortcoming, software agents have been identified as a suitable programming paradigm. The agent based approach commonly uses a middleware for the execution of the software agent. This thesis is meant to compare such agent middleware in their performance in the WSN domain. Therefore two prototypes of applications based on different agent models are implemented for a given set of middleware. After the implementation measurements are extracted in various experiments, which give information about the runtime performance of every middleware in the test set. In the following analysis it is examined whether each middleware under test is suited for the implemented applications in WSN. Thereupon, the results are discussed and compared with the author’s expectations. Finally a short outlook of further possible development and improvements is presented.
# Contents

**PREFACE** .............................................................................................................................................. V  
**ABSTRACT** ........................................................................................................................................ VII  
**CONTENTS** .......................................................................................................................................... IX  

| List of Figures |xi |
| List of Tables | XII |
| List of Code Listings | XII |

1. **INTRODUCTION** .......................................................................................................................... 1  
   1.1 Application and Technology Area .............................................................................................. 1  
   1.2 Motivation and Problem Studied ................................................................. 1  
   1.3 Approach Chosen to Solve the Problem ................................................................................. 2  
   1.4 Thesis Goals and Expected Results ...................................................................................... 2  
   1.5 Thesis Outline ....................................................................................................................... 2  

2. **BACKGROUND** ........................................................................................................................ 3  
   2.1 Agents in Action .................................................................................................................... 3  
   2.2 Middleware for an Easier Development ................................................................................. 4  
   2.3 Centralized Mobile Object Tracking ...................................................................................... 5  
   2.4 Decentralized Search ............................................................................................................. 8  

3. **METHODS AND TOOLS** ........................................................................................................ 13  
   3.1 Methodology ......................................................................................................................... 13  
   3.2 Sun Spots ................................................................................................................................ 13  
   3.3 AFME ....................................................................................................................................... 15  
   3.4 Maps ....................................................................................................................................... 17  
   3.5 JADE - LEAP .......................................................................................................................... 19  

4. **SYSTEM ARCHITECTURE AND DESIGN** ............................................................................. 23  
   4.1 Centralized Mobile Object Tracking ...................................................................................... 24  
      AFME ....................................................................................................................................... 25  
      Maps ....................................................................................................................................... 30  
      JADE ....................................................................................................................................... 34  

ix
List of Figures

Figure 1 Agent migrations in the original WSN tracking application...............................................................6
Figure 2 The WSN for the pheromone based search with agent cloning approach..............................................10
Figure 3 Pheromone base search concept with agent migration approach........................................................11
Figure 4 The composition of Sun SPOTs out of different layers. Picture taken from (15)........................................14
Figure 5 CLDC two stage verification process. Adapted from [18]........................................................................15
Figure 6 Overview of the available JAVA versions and run configurations. Adopted from [27].................................20
Figure 7 JADE LEAP execution modes: (a) Stand-alone execution mode;..........................................................21
Figure 8 Architecture of the Centralized Mobile Object Tracking application....................................................24
Figure 9 Class diagram AFME platform used in the Centralized Mobile Object Tracking application. .................26
Figure 10 Class diagram of the used module, perceptors and actuators by RAC implementation in AFME.................30
Figure 11 Class diagram of the MAPS agent platform and the implemented class of the RAT.............................31
Figure 12 State diagram of the state machine implementing the MAPS based RAT’s behavior............................32
Figure 13 State diagram of the state machine implementing the MAPS based RAS’s behavior...........................32
Figure 14 State diagram of the state machine implementing the MAPS based SCA’s behavior..........................33
Figure 15 Screenshot of implemented Android user interface............................................................................35
Figure 16 Architecture of the implement agent based Android user interface.................................................36
Figure 17 UML class diagram of an AFME platform supporting agent mobility................................................38
Figure 18 UML state diagram modeling the behavior of a MAPS based Decentralized Search agent.....................41
Figure 19 Test configurations for experiments..................................................................................................44
Figure 20 Node deployment in the WSN for a) Centralized Mobile Object Tracking b) Decentralized Search......46
Figure 21 Combined diagrams showing CPU load and energy needs for a) Centralized Mobile Object Tracking and b) Decentralized Search ......................................................................................................50
List of Tables

Table 1 Measured CPU utilizations in % of the centralized mobile object tracking application and the decentralized search application in comparison ................................................................. 47
Table 2 Memory utilization during execution (in kilo bytes) of the centralized mobile object tracking application and the decentralized search application in comparison ................................. 48
Table 3 Energy consumption of the implementations .............................................................................. 49
Table 4 Wireless traffic results for implementations of the centralized mobile object tracking application ................................................................................................................................. 51
Table 5 CPU and Energy consumption results for agent migration test scenario ................................. 52

List of Code Listings

Code Listing 1 necessary source modification of MAPS internal class
MobileAgentCommunicationChannelSender .......................................................................................... 19
Code Listing 2 AFME platform definition for a sensor node ................................................................... 25
Code Listing 3 AFME definitions of all AFME based agents for the tracking application a) RAT, b) RAS, c) SCA and d) RAC .................................................................................................................. 28
Code Listing 4 Agent definition of the AFME decentralized search agent ............................................. 39
1. Introduction

1.1 Application and Technology Area

Progress in the fields of electronics and miniaturization has led to the development of small embedded wireless sensors, which are used in large numbers in different applications. The wireless communication capability of such sensors is the main reason for the emergence of new applications, in which their wired counterparts are not suitable [1]. The potential of these wireless sensors can increase even further, when they form a network of cooperative sensor nodes. These wireless sensor networks (WSN) are an emerging and promising technology for a wide field of applications in the civilian and military sector, such as in border line surveillance or in the monitoring of patients in health care.

Besides their potential, the development of a wireless sensor network application is a challenging task due to the distributed and embedded characteristics of such a network. Out of several proposed programming paradigms to overcome this challenge, the software agent based approach is very promising [2; 3] and is therefore addressed in this thesis. In general, the execution of software agents requires a middleware. A great variety of middleware that supports software agents on different hardware-platforms is available for different programming languages. Yet the choice for a suitable middleware can be difficult, because it is hard to compare their performances directly.

1.2 Motivation and Problem Studied

The problem that is focused in this thesis is to compare the suitability of selected middleware approaches for a specific application in the WSN domain. Several functional prototypes of two applications, the Centralized Mobile Object Tracking and the Decentralized Search, are developed. These prototypes make it possible to analyze their performances and needed resources. The results provided by this analysis can be used as evaluation criteria to choose the most appropriate middleware among those that are tested.
1.3 Approach Chosen to Solve the Problem

In order to solve the problem addressed in this thesis, the following approach is used to support the process of choosing a middleware for a specific application. For the comparison of three middleware in their performance of execution, two predefined and then implemented applications are realized on a chosen hardware platform. The applications are a tracking application, which is a well known application for WSN and a Decentralized Search application, which is based on a pheromone-coordination strategy (for more detailed information about the implemented applications see chapter 2.3 and 2.4). To be able to compare the middleware in the context of a chosen application, prototypes of the applications are then implemented on the evaluated middleware. After these implementations experiments are conducted, which are executed according to defined test configurations in order to extract metrics representing the run time execution performance to certain criteria of the middleware executing the test application.

1.4 Thesis Goals and Expected Results

- Implementation of functional and comparable prototypes for a given set of middleware for the tracking and search application.
- Extraction of measurements about the performance and resource needs of every middleware in the test set.
- Analysis of the measurements to determine the suitability of every member of the set of middleware for the specific applications.

1.5 Thesis Outline

This thesis consists of seven main chapters in addition to this introduction. Chapter 2 provides an overview about software agents and middleware as well as the necessary background information for the implemented applications. In Chapter 3 the used hardware platform, its main properties and the set of the chosen agent middleware used in this thesis are introduced. Furthermore, the methodology of the work is presented in this chapter. Chapter 4 focuses on the overall architecture of both applications as well as on the architecture of their components. The necessary distinctions in the overall architectural design, which are the result of different properties and characteristics of the different middleware, are also shown. In Chapter 5 the results are presented and the method of extraction is shown. These results and measurements are discussed in Chapter 6. Several related works are outlined in Chapter 7 followed by a conclusion of the work and suggestions for further improvements and future work in Chapter 8.
2 Background

Programming a WSN application is not trivial due to the limitations of the used hardware platforms, like processing power as well as memory and energy resources and the complexity of a highly distributed wireless system. Most implementations for WSN are application- or domain-specific and necessitate trade off in the fields of task complexity, communication patterns and resource usage. Hence these specific implementations include most likely modules for routing mechanisms, time synchronization, node localization and data aggregation. These modules have dependencies to their applications and therefore these parts can hardly be reused. A WSN-aware programming paradigm is needed to support a rapid development and a highly flexible deployment of WSN software. This paradigm will be explained in the following subchapter. The agent-based approach has a high potential for the use in WSN applications, as it meets the demands mentioned above.[2; 4]

2.1 Agents in action

There has been a lively discussion in the literature about the precise definition of the term software agent and about how software agents differ from ordinary computer programs. In [5] the authors Franklin and Graesser show the similarities and differences of several definitions for software agents and provide a taxonomy that attempts to contain all of them. The authors also extract some mutual concepts from the definitions discussed in their work. According to these concepts, a software agent can be defined as a software entity or as an additional abstraction layer. Franklin and Graesser state that a software agent has to support the following properties:

- Autonomy: software agents can make decision about how to reach a certain goal and which actions to take without any interaction with the user or other programs.
- Reactivity: software agents are capable of receiving events and environmental changes and trigger a responding action.
- Social ability: Interaction between a software agent and other entities (e.g. user, other agents etc...) is possible and can lead to coordination, cooperation or even competition.
- Persistence: The execution of a software agent is continuously, in contrast to the sequential execution of operation in normal software.

In [6] a popular definition for software agents can be found, in which behavior is a fundamental concept. There are other works that provide various taxonomies for the classification of software agents. Different taxonomies are presented for example in the works of Hector [7] or Sakarkar and Shelke [8].
According to [9] another essential property for a software agent in the WSN domain is agent mobility. A mobile software agent can transfer its execution from one system to another system in the network and can subsequently continue its operation on the target system. The process of transferring an agent from a source system to a destination system is often called migration. While migrating the code of the software agent, which implements its behavior, it is possible for the agent to carry its data, also named state, during the transfer. The migration is called stateful, if the state is included in the transmission. If the state is excluded, the migration is referred to as stateless.[10] After the agent is transferred, the migrated agent is only executed on the destination system.

There is an additional concept called cloning in the context of agent mobility, which extends agent migration [11]. When using agent cloning, a copy of the original agent runs on the destination system and the original agent itself is still present on the source system. Hence, the difference to agent migration is that with agent cloning the copy and the original are executed on both systems simultaneously. To accomplish that the agent’s execution is paused, a copy is made which is transferred onto the target system and the agent’s execution is resumed.

The previous mentioned social property of software agents leads to multi-agent systems (MAS), in which agents can interact with each other. It should be noticed that the interaction is not necessarily between two or more agents, but it is also possible that an agent interacts with humans or other entities. MAS are suited for complex tasks, which are difficult or even impossible to be performed by one single agent or a conventional software system. [12]

2.2 Middleware for an Easier Development

The development of a MAS from the scratch is a challenging task and would exceed the time budgets of most WSN application development projects. Therefore, many multi-agent platforms (MAP) are provided by several companies, academic institutions or open source projects to develop, run and manage MAS.

MAPs are implemented as an additional abstraction layer between the operating system and the agents executed by the platform and thus they function as middleware. The software developer is supported with a flexible framework for a rapid implementation of MAS applications by MAPs. This is made possible through the supply of an environment, in which an agent can be executed and through essential services e.g. agent communication, migration, scheduling and accessing system resources. [13]
In this thesis the focus is on lightweight MAPs which are able to run on target systems with limited hardware resources, because these systems are most common in a WSN. Virtual machines are also available for such embedded systems. Due to their availability, Java as object orientated high-level programming language can be used to develop applications for the target systems mentioned above.

Two reference mobile object tracking applications are implemented on the following Java based MAPs: JADE, MAPS and AFME on Oracle Sunspots [14] as the chosen hardware platform capable of the execution of Java byte code.

2.3 Centralized Mobile Object Tracking

One of the mobile object tracking applications, the Centralized Mobile Object Tracking, is based on the work of Freitas et.al. [1] and Allgayer et.al. [15]. Contrary to their work, in which mobile agents are used, the centralized version of the mobile object tracking presented in this thesis is implemented with cooperating static agents. The reason for that is that one of the used MAPs cannot support agent migration. In this section the approach proposed by Allgayer et.al. is discussed and afterwards the modifications for the implemented tracking application without agent mobility are demonstrated.

The WSN consists of distributed sensor nodes on predefined positions. These nodes are able to perform processing and communication with their neighboring nodes and to sense the distance to the target node. Depending on the characteristics of the mobile target, different types of active and passive sensors can be used for this purpose. In the implementations that are developed in this thesis the wireless interface is employed as sensor. The target node continuously sends radio signals, which are called beacons. These beacons are received by the sensor nodes with their local wireless interface. During the reception the sensor nodes measure the signal strength. Hence the software calculates the distance between the target node and the receiving sensor node via the omnidirectional model of electromagnetic wave propagation. This model states that the power of the beacon signal declines quadratically over the distance.

The suggested sensor network consists of three types of nodes. The first type is the already mentioned target node, which can be tracked by the network. The second type, the sensor node, has the function of sensing the beacon frames broadcasted by the target. The WSN consists of several sensor nodes, but to accomplish a successful localization of the target object at least three nodes of this second type have to be in range of the target. The third type is a coordination node, which is responsible for the start of the mobile agents. It also functions as sink node, in other words the
calculated result - in this case the location of the target – exits the WSN over the coordination node. Figure 1 shows the structure of the WSN with all three types of nodes for the Centralized Mobile Object Tracking.

The authors in [1] propose an implementation with two types of software agents: Collaborative Agents (CAs) and Resident Agents (RAs). RAs are not mobile and therefore they stay on a fix node. These agents can either be a RA_Target (RAT), a RA_Coordinator (RAC) or a RA_Sensor (RAS) depending on which type of node the agents are created. RAS are executed on sensor nodes. They perform the measurements of the signal strengths of the received beacons broadcasted by the target object. In addition to that the RAC runs on the coordinator node, which is also the starting point of the execution of the CAs.

CAs are able to migrate between nodes. Their function is to perform the localization of the target. Therefore, these agents communicate with their local RAS that provide the distance to the target. A CA can either be a CA_Master (CAM) or a CA_Slave (CAS). CAS are simply forwarding the distance information received via the local RAS to the CAM. The CAM uses this information together with its local reading to calculate the target’s position. An additional task handled by the CAM is the coordination of CA migrations. These migrations are essential in case the target leaves the range of a sensor node that hosts a CA.

![Figure 1 Agent migrations in the original WSN tracking application.](image)
Figure 1 depicts all necessary agent migrations and the tracking process in the original WSN tracking application. The initialization of the WSN is shown from a) to c). Three RAS detect the broadcasted beacon frames emitted by a RAT, which is also the start condition of the following step. Via messaging service provided by the MAP, three RAS inform the RAC that a RAT is in range (Figure 1(a)). The RAC subsequently requests the CAM to migrate to the sensor node closest to the target object (Figure 1(b)). When the CAM resumes its execution on the closest node, two agents cloned from the CAM are sent to the other two sensor nodes, which previously have informed the coordinator about the received beacons (Figure 1(c)). In contrast to the master agent the clones act as slaves. Figure 1(d) shows the migration of one of the CAS when the target object has left the range of its current host.

Due to the fact that not all available MAPs for the chosen hardware platform support agent cloning (one even cannot perform agent migration), some modifications of the presented solution are necessary in order to implement this application on the chosen MAPs. The mobile CAs are replaced by static agents, the static cooperative agents (SCAs) and hosted on every sensor node. SCAs have three possible roles: they can be inactive, master or slave. The task of managing the CA has moved from the CAM to the RAC so that the implementation is simplified. To be able to fulfill this task the RAC needs information about which nodes are in range of the target. Therefore, the RASs also send their distance readings to the RAC, but in a lower frequency than to the local active CA. In case the target object comes in range, the distance readings are sent immediately.

To minimize of complexity of the application the following limitation have to keep in mind. Only one target node is supported, i.e. the network is able to track just one target at a time. The target node and sensor node are on a two dimensional plane. It is assumed that the distance between neighbor nodes assures the three nodes in the triangle are in range of each other and that the communication is not error-prone. A routing protocol can be used for direct communication between all agent platforms on the sensor nodes and the platform hosting the RAC.

The three sensor nodes closest to the target object calculate the position of the target object via triangulation. Hence, the nodes have to be arranged so that three of them form an equilateral triangle, as it can be seen in Figure 1. The position of the target object \((x_0, y_0)\) can be determined by the formula represented by (1). The formula uses the positions of the sensor nodes \((x_i, y_i)\). These positions have to be known by the calculating node as well as the distance for the participating sensor nodes to the target object \((r_i)\) for \(i = 1\) to \(3\). The cooperative agent, which acts as master and
which is supplied by the slaves with their distance readings, is responsible for executing this
triangulation algorithm.

\[ P_0 = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = A^{-1} \times b \]  

For \( A \) and \( b \):

\[
A = 2 \times \begin{bmatrix} x_3 - x_1 & y_3 - y_1 \\ x_3 - x_1 & y_3 - y_2 \end{bmatrix}
\]

\[
b = \begin{bmatrix} (r_1^2 - r_3^2) \\ (r_2^2 - r_3^2) \end{bmatrix} - \begin{bmatrix} (x_1^2 - x_3^2) + (y_1^2 - y_3^2) \\ (x_2^2 - x_3^2) + (y_2^2 - y_3^2) \end{bmatrix}
\]

The role in which a SCA acts is managed by the RAC. Therefore, the RAC uses the provided distance
information to determine the three closest sensor nodes to the target object. This information is
provided by the RAS as mentioned before. The starting condition for the triangulation is that the
target object is in range of three sensor nodes. After the RAC received distance information from at
least three RAS, it organizes this information in a list sorted by the distances. The RAC then activate
the SCAs on the closest sensor nodes according to this list and assigns their roles. This is
accomplished via messages, which contain the role and all necessary information. A SCA, destined to
act as a slave, is informed about the address of the master. The master is provided with its position
and the coordinates of the slaves to be able executed the triangulation algorithm. The RAS informs
the RAC supplementary to the repeated transmissions, if the target gets in range, if the first beacon is
received and if the target gets out of range. As a result, the response to the movement of the target
object is enhanced.

2.4 Decentralized Search

The second application used in this thesis, the Decentralized Search, is based on the pheromone-
coordination strategy presented by Freitas et al. in [16]. It can be seen as a completely decentralized
version of tracking. The authors show a pheromone-based approach to coordinate a network of
unmanned aerial vehicles (UAVs) and ground sensor nodes. It is used to forward alarms from a
ground sensor network to UAV drones. In this work virtual pheromones are used to find the sensor
node closest to the mobile node. The moving nodes (UAVs) emit radio beacon equally as in the
application described above. These beacons are stored as virtual pheromone marks on the static
sensor nodes in its range. Thereby the initial value of the pheromone mark is determined by the
strength of the beacon signal. As with real pheromone trails in nature, for example trails used by ants to track food, virtual pheromone tracks fade over time. The formed pheromone trails show a gradient concentration of pheromones, which indicates the movement of the mobile nodes. While a static sensor wants to deliver a message to a mobile one, the message just follows this concentration towards the increasing direction of gradient. In Figure 2 a WSN for the pheromone based search is demonstrated. Figure 2(a) shows a small WSN with the virtual pheromone marks left by the target objects after the mobile node moved through. The illustration also depicts a possible starting position of the agent searching the mobile node. This WSN consists of one type of sensor node, which executes two kinds of agents, CA and RA as described in section 2.3. Sensing the beacons and the handling of the pheromone mark is the responsibility of a resident agent, the RA_Sensor (RAS). The second agent type is a collaborative search agent (CSA). The mobile CSAs are responsible for finding the closest sensor node to the mobile node by following the pheromone marks.

The first approach to implement the Decentralized Search can be seen in Figure 2(b). A CSA follows the trail by cloning itself to the neighboring nodes of its hosting node. When a clone starts its execution on a node, it interacts with the local RAS, which provides the current level of the pheromone trail. By comparison between the pheromone levels of the previous and the hosting node, the CAS decides its next step. In case the pheromone level on the current node increases, the CAS clones itself to the new neighboring nodes and informs the CAS from which it was cloned, about the higher level on the current node. Otherwise the CAS terminates its execution. If a CAS is informed by one of its clones about a higher pheromone level, it also ends its lifecycle. If there is no message, the current node is the closest one and therefore the desired destination node.
Due to the fact that agent cloning is not supported by all the MAPs used in this thesis, some modification to this approach are necessary. Therefore agent migration was used, which is supported by all the MAPs, instead of agent cloning. The algorithm implemented as agent behavior to fulfill the search follows these steps:
a) The searching agent obtains a list of neighbor nodes and its current pheromone level from the current hosting node. If no more unvisited nodes can be obtained or the pheromone levels are less than on a previous node, the agent migrates to the node with the highest pheromone mark, its final position and ends the search.

b) The agent executes a stateful migration to all neighbor nodes in the list, except for already visited ones, and stores the highest pheromone level found and addresses of visited nodes as state.

c) If there are no more nodes to visit in the list, the agent migrates to the node with the highest pheromone level and continues with step a).

This behavior of the searching agent is also depicted in Figure 3 that depicts the pheromone based search concept with agent migration approach. At the start node (node a) the agent gets the list of its direct neighbors (node b and c). After visiting both nodes, the agent continues on the node with the higher pheromone level (node c). Depending on the order of nodes (b,c or c,b), the agent might have already been on c. If this is not the case, the agent has to perform an additional migration from b to c. Having arrived on c, the agent obtains a new list of neighbors (d,e and a). The node a is dropped, because the agent already knows that it has been there, so the pheromone level of node a is familiar.

In case a routing protocol is provided by either the MAP or by the hardware platform, the agent is able to directly migrate to a node which is not in its range (e.g. from b to c). If this is not the case, the agent has to take a detour over a node in its range (e.g. from b over a to c).

![Figure 3 Pheromone base search concept with agent migration approach.](image-url)
3 Methods and Tools

3.1 Methodology

To be able to compare the chosen MAPs running the two applications, which are described in Sections 2.3 and 2.4, prototypes for them have been implemented. To achieve comparable results, the applications are implemented as similar as possible despite the different architectures and the distinctive concepts of the selected MAPs. Therefore most parts of the code defining agent’s behaviors were reused in the implementation for the different MAPS. The experiments of all tests are done on identical hardware, in this case the same devices, to ensure that the results are not influenced by meanderings in the hardware. Furthermore, only features and services provided by all MAPs are used in the implementations. Different characteristics of the MAPs, which are shown in this chapter, lead to necessary differences in the architecture of the overall system. Hence only the implementations of similar subsystems are compared in this work. For further similarity the common programming paradigm was chosen, therefore the agent behaviors or components develop due to the state machine programming model.

This chapter introduces the used hardware platform the Sun Oracle Sun Spot[14] as well as the MAPs on which the applications are implemented. Furthermore, the steps necessary to successfully use the MAPs as middleware (MW) on the selected hardware platform are shown.

3.2 Sun Spots

Oracle Labs, the former Sun Labs, developed the Small Programmable Object Technology (SPOT) to provide an experimental hardware platform as well as the development software needed to create a wide range of embedded wireless applications. A SPOT device, from now on called Sun Spot, is an embedded device, slightly bigger than a box of matches, equipped a wireless interface, battery supply, processor unit and several built-in sensors. As it can be seen in Figure 4 a normal Spot consists out of three different boards. There is a second type of Spots, the base stations, unlike normal Spots they only have the processor board layer (including CPU and wireless interface), but no sensors board and power supply. Base stations can act as interface for a personal computer (PC) to a network out of Spots or execute code on itself.

Besides the hardware a main characteristic of such a Sun Spot is that it is programmable completely in Java. This is made possible by the usages of a Java Virtual Machine (VM), called Squawk VM, which designed to have a minimal footprint and therefore is suitable to run on embedded devices. The
Squawk VM supports the Connected Limited Device Configuration (CLDC) 1.1 and the Mobile Information Device Profile (MIDP) 1.0. Therefore Java is the preferred programming language to program such a device.

![SunSPOT](image)

**Figure 4 The composition of Sun SPOTs out of different layers. Picture taken from (15)**

Due to the fact that Java is a high level object orientated language, the SPOT allows a more rapid development than other embedded development platforms, which are often programmed in lower level languages like e.g. C or nesC. Another advantage of the SPOT is the provided Software Development Kit (SDK) and its possible integration in popular Integrated Development Environments like NetBeans and Eclipse.[17]

In regard to WSN there is a disadvantage with SPOT. Due to the fact that the Squawk VM supports CLDC 1.1, dynamic class loading is not possible. CLDC is per definition a Java 2 Micro Edition (J2ME) configuration. These can be seen as a subset of libraries and of features of the Java 2 Platform, Standard Edition (J2SE) for mobile devices, like e.g. cell phones or PDAs with limited hardware resources. CLDC focuses on devices with limited resources. As a result, several features of the J2SE are not supported with CLDC, like e.g. the J2SE security model. It is replaced by several simpler and more resource friendly security concepts. One of these concepts is responsible for the prevention of dynamic class loading, the Sandbox model. This model states that an application cannot use any resources or libraries that are not part of its scope. This means that an application running on a J2ME with CLDC cannot load any new class, which is not part of the application jar file. Its predefined functionality can therefore not be extended. [18] In the context of software agents in WSN this results in the fact that every class, which an agent might need, must already be present on the device. Even if the agent is not running on this device, all needed classes must be available on the device to support the potential migration of such an agent. So it is not possible to simply add a new
agent with new features, which need a new code, to the network without updating every node the agent is supposed to run on.

Additional to the limitation of the security model used in CLDC there is another characteristic. In difference to the J2SE verification method, which would need too many resources (Memory and processing time), application and libraries must be pre-verified to be able to execute on a CLDC device. The verification process is depicted in Figure 5. The verification is done to ensure that only valid applications are executed on the device. This was the reason for some initial problems with the Agent Factory Micro Edition (AFME).[18]

![Figure 5 CLDC two stage verification process. Adapted from [18]](image)

### 3.3 AFME

Pervasive systems are the intended area of application of AFME, a MAP which is based on the Agent Factory Framework [19]. The MAP has a minimized need of resources and is designed for devices, which are compliant to the MIDP of J2ME and therefore support Sun Spots. AFME operates according to the Believe-Desire-Intention (BDI) paradigm [20]. This states that an agent is carried out in a sense-deliberate-act cycle, which is implemented in AME as a periodically scheduled sequence of four steps. In the first step the agent perceives information about its environment and updates the agent’s belief states. These belief states represent the set of information available to the agent about its current status and its environment. In the second step the agent uses resolution based reasoning to determine the agent’s desires. This is followed by the determination of the agent’s intentions, which are in most cases its desires. If needed, a selection process is invoked to identify the agent’s
commitments, which are the set or subset of the determined desires. According to the identified commitments, certain actions are performed by the agent through its actuators in the final step.[21]

The platform provides four main component classes that the developer has to extend for the implementation of an application. An AFME agent is composed of perceptors, actuators and modules.[21] Perceptors are called in the first step of an agent’s execution sequence and enable an agent to perceive information from its environment, from other agents or from the agent itself. Perceptors are also responsible for the update of the agent’s belief states that are based on the perceived information. Actuators are called in the last step of the execution sequence. Each of them represents a certain action, which an agent could take to fulfill its desires. Modules are used for agent internal information exchange between perceptors and actuators. The usage of modules is necessary due to the loose coupling of agent components. There are no references and dependencies between one object and another object. This results in the advantage that components can easily be replaced or updated without touching additional components. Modules can only be employed by their agents. To enable data exchange between agents, AFME uses objects which extend the service component. The platform also offers predefined services, e.g. the Message Transport Service (MTS) which is used for message based communication between agents or the Radio Migration Manager which handles agent migration. Services are directly started by the agent platform itself.

For implementing an application with AFME, it is suggested that both the declarative model and the imperative programming model are employed. The implementation of modules, services, perceptors and actuators is done imperatively in Java. The definition of the agent platform and the agent’s behavior should be done in a declarative AFME language, which is a minimized version of the Agent Factory Agent Programming Language 2 (AFAPL2). This AFME language is used to define a declarative set of rules representing the agent’s behavior and the properties of the agent platform. A compiler is generating java code out of the definitions and hardware specific templates. Through these templates a definition of an agent can be reused on different hardware platforms by simply switching these templates. For example, there is a template for Sun Spots which generates a code without a graphical user interface (GUI). A complete imperative implementation with Java is possible, but has disadvantages like e.g. there is no syntax checking for rules and the support of rules which include mathematical expressions is missing. Three exemplary rules can be seen below. Each rule consists out of beliefs and actions separated trough a greater-than sign. If an AFME agent belief set includes the beliefs on the left side of a rule, it will be committed to do one or more actions which are represented on the right side of a rule. That the evaluation of the conditions for an action can be
considered to be true, a resolution-based reasoning is done. Furthermore, negation (3), variables (2) and mathematical expressions can be applied in AFME rules.[21; 4]

\[ b_1, b_2 \rightarrow \text{doSomething}; \quad (1) \]

\[ c, d(?\text{var}) \rightarrow \text{doSomethingWith(?\text{var})}; \quad (2) \]

\[ !a, e \rightarrow \text{doSomethingElse}; \quad (3) \]

Due to the fact that it is possible to use a different template for different hardware platforms AFME can be executed on a PC which is connected to a Sun Spot base station. This combination can act as a gateway to the WSN and/or as sink node.

### 3.4 MAPS

The Mobile Agent Platform for Sun Spots (MAPS) is a middleware for software agents designed for WSN. The MAPS was developed for the SPOT and therefore utilizes some specific features of the Squawk VM. This makes the Squawk VM a requirement for this MAP. The main characteristics of MAPS are that they have lightweight agents, an agent server architecture, a provision of minimal core services and a plug-in-based extensions architecture. The lightweight agent architecture ensures agent migration and execution with high efficiency. The core services offer support for migration, for agent naming and for communication and they provide scheduling and access to sensor readings. The platform can be easily extended with other services due to the plug-in-based extension architecture. MAPS itself is composed of several components. These components interact with each other via using an event based approach. MAPS agents are implemented according to the imperative programming model, which defines an agent’s behavior in a multi-plane state machine. A multi-plane state machine was chosen to enable role specific behavior. Every plane corresponds to a certain role of an agent and the state machine represents the behavior of the agent in this role. The result of this architecture is that three popular programming paradigms for WSN are utilized in MAPS. These paradigms are event-, state- and agent-based programming. [4]

As previously mentioned MAPS requires specific features of the Squawk VM. The most important one is *Isolates*, which is not exclusive in the Squawk VM, but is defined in Java Specification Request (JSR) 121 (Application Isolation API) [22]. Extending JSR 121, Isolates in the Squawk VM possess one additional feature, isolate migration. This feature allows that the execution of such an Isolate can be paused, serialized and then it can be transferred over a network or stored on a storage device. After the Isolate is reloaded in a host’s memory (even from hosts with a different machine word byte-
order), the execution of the Isolate is continued from an instance of the Squawk VM.[23] The main concept of Isolates is that an application is isolated from other applications via threaded objects managed by the VM. Isolates can be seen as a possible implementation of the sandbox model, which is discussed in section 3.2. Using MAPS, the Isolate represents a very important concept, because the agent platform itself and all agents are realized as Isolates.[3] MAPS utilizes the Squawk VM Isolate migration for agent migration processes.

Although MAPS was developed for SPOT, some problems with the supplied library occurred during the implementation of the Centralized Mobile Object Tracking and the Decentralized Search. The MAPS library used in this work is the version 1.1. With this version, it was not possible to run the tutorial application taken from the MAPS documentation, because of several runtime exceptions which led to a restart of the hardware. Therefore a java decompiler was needed to obtain the source code of MAPS, which is published under the GNU General Public License [24]. The original source of MAPS was not available. The MAPS team provides a Subversion repository, but that was empty at the creation time of this work (24.10.2011.) [25]. After some minor modifications to the obtained source code were performed, all errors highlighted from the IDE are corrected and the modified version of MAPS is now able to start on a Sun Spot. Later in the implementation process, a new problem concerning the outgoing communication occurred: the platform stopped transmitting. After a second source code review, an internal class was extended, which is responsible for all outgoing communication of a node. This modification gives MAPS agents the ability to communicate over the entire runtime. These changes can be seen in the code presented in Listing 1. In the modified version an additional while loop, which runs as long as the node, executes the agent platform and ensures ongoing communication and node discovery. The review of the code also revealed that MAPS does not support agent cloning, because the responsible method in the source code is empty. This contradicts the statement of Aiello et. al. in [26] that MAPS supports agent cloning.
As mentioned above MAPS provides a basic service for accessing system resources. This service is initiated when the platform starts, regardless if needed. Therefore it is not possible, without modifications, to use MAPS on SPOT base station because of the missing sensor board.

### 3.5 JADE - LEAP

JADE stands for Java Agent DEvelopment framework. JADE is a middleware for distributed MAS and in contrast to AFME and MAPS it was not initially designed for hardware platforms with limited resources. JADE itself cannot be executed on J2ME platforms, because it requires a Java VM supporting Java 5. Furthermore, the memory needs of JADE exceed the capacity of most CLDC devices. However, the execution of the MAP on devices with J2ME (CDC or CLDC) is possible through the Lightweight Extensible Agent Platform (LEAP), which is an add-on for JADE. Figure 6 shows the different Java platforms that are available at the time of writing. With JADE and JADE-LEAP MAS applications can be developed that can be distributed over all Java platforms except over Java Card.

In addition to that JADE-LEAP also supports .Net, Android 2.1 and higher versions of Android. Both JADE and JADE-LEAP provide almost the same Application Programming Interface (API) on all supported platforms. An exception is the MIDP, because the VM provides it with a reduced set of functionality.

---

```java
original:
public void run()
{
    waitForCommunications();
    while (this.communicationEvents.size() > 0)
    {
        ...
        //out going message handling
        ...  
    }
}

modification:
public void run()
{
    while(true){
        waitForCommunications();
        while (this.communicationEvents.size() > 0) {
            ...
            //out going message handling
            ...
        }
    }
}
```

**Code Listing 1 necessary source modification of MAPS internal class MobileAgentCommunicationChannelSender**
With AFME and MAPS every device runs a MAP and agents are hosted on the platform itself. A difference in regard to AFME and MAPS is that JADE has an additional abstraction level called container. With JADE agents are executed in a container, which is part of the platform. A platform requires one main container and can be distributed over several devices. Every device is represented through a container. In addition, it has to be mentioned that it is also possible to execute more than one container or even more than one platform on a single device (only J2SE and J2EE). JADE-LEAP provides two modes of operation: 1) the stand-alone execution mode and 2) the split execution mode. In the stand-alone mode a complete agent container is started on the device. This mode can be used on all supported platforms excluding MIDP. For MIDP devices the usage of the split execution mode is mandatory. When operating in this mode, JADE-LEAP separates the agent container in a frontend and a backend. The frontend, which is hosted on the mobile device, requires fewer resources than the execution of a complete agent container. The backend running on a J2SE or J2EE VM connects the split container to a main container. Figure 7 depicts the differences between the stand-alone and split execution mode.\[28;29;30\]

Additional to the reduced resources requirement, the split execution mode has other advantages on resource constrained wireless devices. While initializing the connection between the container and the JADE runtime, which hosts the main container, the necessary communication is completely handled through the backend. This results in a faster initialization and less wireless traffic.

Figure 6 Overview of the available JAVA versions and run configurations. Adopted from [27]
Furthermore, the binary coding of split container internal communication, which uses the wireless interface of the mobile device, is more efficient and also reduces the wireless traffic. Figure 7 b) shows a possible topology, in which the execution of the main container and the backend is performed on different devices.[30; 29]

Figure 7 JADE LEAP execution modes:
(a) Stand-alone execution mode and (b) Split execution mode (figure taken from[30])

JADE offers a lot of features and is much more powerful than AFME and MAPS, but a more detailed description would exceed the scope of this thesis. More information about JADE can be found on the official website [31].

Some important facts for the JADE-LEAP for MIDP conform devices are highlighted below, because Sun Spots are chosen as hardware platform in this work. JADE-LEAP depends on a device in a system, which is able to provide the main container for the application. Therefore it is not possible to implement the Decentralized Search application from section 2.4. For the implementation of the Centralized Mobile Object Tracking that has been introduced in section 2.3 with JADE-LEAP, some properties of the MAP have to be kept in mind. Due to the fact that a backend hides the address of the device, which hosts the frontend from the other containers on the platform [29], no direct communication over JADEs MTS is possible between the devices that host the frontends. This could be a disadvantage in WSN, because there is a higher utilization of the wireless interfaces in the network. The need for a meshed routing protocol, which ensures that every node can communicate with its backend, is another disadvantage in WSN.
JADE also offers a GUI for managing and debugging a whole agent platform and several service agents like the spy-agent, which allow the visualization of the communication between all agents on a platform.

It is not possible to use the distributed binary libraries on Sun Spots. The reason for that is related to the dependencies that the J2ME GUI package has, which are not supported by the Squawk VM. So it is necessary to generate the JADE-LEAP source for MIDP according to [30], followed by the removal of any dependencies to the GUI package from the source code. It should also be noted that for the successful generation of the JADE-LEAP source an installed Sun Java Micro Edition SDK is necessary. This step is followed by a pre-verification of the compiled library via the pre-verifier provided by the Sun SPOT SDK. Alternatively it is possible to directly include the JADE-LEAP source in development projects. For a successful connection between frontends and backends a tool called socked proxy has to be executed on a PC connected to an SPOT base station. The socket proxy allows TCP based connection from a Sun Spot to an end point in an IP based network and is part of the Sun SPOT SDK.
4 System Architecture and Design

This section focuses on the architecture of the two applications in which the agent middleware are compared. First the centralized prototype will be discussed, followed by the Decentralized Search one. Due to the collaborative characteristics of the test applications, where different parts on different nodes are working together in the same system, it is necessary to show every part of its own and how the different parts cooperate between each other.

To achieve comparable results, the prototypes are implemented as similar as possible on the MAPs. Besides the different applications, the implementation that transmits and receives the beacons is used in both applications on all MAPs. The beacons are realized as Radiogram broadcasts and employ the Radiogram class of the SPOT’s Squawk VM. This class also provides a method to process the received signal strength indicator (RSSI) of Radiogram connection, which is named getRssi(). The result of this method is used to calculate the distance in the tracking application and to set the initial value of the virtual pheromone mark utilized in the Decentralized Search application. The only difference between both applications is the frequency, in which the beacons are emitted. In the mobile object tracking application this frequency is 2 Hz, in the Decentralized Search application it is 0,66 Hz. The reason for the lower frequency of the beacon broadcasts in the Decentralized Search application is that the disturbances of running agent migrations decrease.
4.1 Centralized Mobile Object Tracking

The Centralized Mobile Object Tracking prototypes consist of three components. The first component is the WSN itself, which executes the application implemented on the three MAPs. The target positions obtained from the WSN are stored in a MySql database. From there the positions are queried and presented in the agent based GUI, which is implemented in JADE and executes on an Android device. The various properties and characteristics of the used MAPs result in necessary differences in the implementations.

Figure 8 Architecture of the Centralized Mobile Object Tracking application
Several components which are used in the different implementations share a common code base in
the Centralized Mobile Object Tracking application. This common code base was only modified to fit
the different ways of message handling of the MAPs.

AFME

In Figure 8 a) the structure of the tracking application based on AFME is shown. As it can be seen the
RAC agent is executed on a J2SE platform running on a common PC. It would be possible to execute
the RAC directly on the base station, but with the disadvantage that an additional program would be
needed, running on the PC which receives the results over the universal serial bus (USB) and forward
them to the database. An advantage of the execution on the PC is that the full functionality of the
J2SE can be accessed in difference to the reduced one provided by J2ME.

Due to the fact that AFME uses an imperative part and a declarative part for the definition of an
agent, both definitions and the way they are linked together are explained. For the execution of an
AFME based agent a defined platform is required. Such a platform definition is presented in Code
Listing 2. It specifies the platform for the coordinator nodes. The definition states that the platform
uses two schedulers, which results in a platform executed in two threads. This is followed by the
definition of the services that the platform should provide. In the illustrated example the
RadigramMTS service is providing a radiogram base message transport service on port 66 (line 3 in
Code Listing 2). This service allows message based communication between agents. After the
provided services are defined, it is specified which agents should be created and started. The value of
the agent control cycle is also defined in line 4 in Code Listing 2. In the example an agent from the
type RACoordinator, named RAC is defined with a scheduled sense-deliberate-act cycle every
1000 ms (line 4 in Code Listing 2). The initial belief states can also be set at this point for the RAC
agent. The belief “alive” is added to the agents belief set (line 5 in Code Listing 2). The used keyword

| 1 | platform Basestation Platform{ |
| 2 | scheduler 2; |
| 3 | service com.agentfactory.radio.RadiogramMTS BaseStation 66; |
| 4 | create RAC RACoordinator 1000; |
| 5 | add RACoordinator always(alive); |
| 6 | start RACoordinator; |
| 7 | template Deploylet.template Baseplatlet |
| 8 | EmuMigPlatform.template RASensorAgentPlatform; |
| 9 | } |

Code Listing 2 AFME platform definition for a sensor node
“always” declares that the agent does not drop the belief state after a cycle. Therefore, a state that is added with this keyword can be regarded as persistent, because that state is true until it is explicitly removed from the agent’s set of belief states or until the agent is terminated. The last part of the platform definition indicates which templates should be used for the code generation. The AFME compiler uses this information to generate a Java source code for the agent platform and its agents. Figure 9 depicts the class diagram of an AFME platform used in the tracking application. The class RABasePlatform represents the defined agent platform and implements the Platform interface which is provided by the AFME API. This interface defines the functionality for a minimal AFME agent platform. The usage of a service, the RadigramMTS service, is also illustrated in this diagram. Agents are presented as BasicRunnable objects to the platform. This class provides the basic functionality for the execution of an agent. The exemplified platform definition is used for all nodes in the AFME implementation of the tracking application, except for the target node. On the target node no message transport service is needed, because the RAT does not communicate with any other agent.

Figure 9 Class diagram AFME platform used in the Centralized Mobile Object Tracking application.
The declarative definition of an agent includes the used actuators, perceptors and the rules for the resolution-based reasoning, which define the behavior of the agent. The simplest agent in the application is the RAT, whose declarative definition can be seen in Code Listing 3 a). The agent has one actuator named `BeaconAct` and one rule. The rule determines that if the state “sendBeacon” is believed to be true the action “transmittBeacon” should be performed (line 2 in Code Listing 3 a)). The actuator responsible for this action is implemented in the class `BeaconAct`. The source code of the actuator can be seen in appendix c.1.1. The mapping between declarative and imperative parts is defined through a string parameter in the constructor of the actuator class. According to this rule, the agent has to believe that the state “sendBeacon” is true. This is accomplished with a persistent initial belief for “sendBeacon” in the definition of the platform hosting the agent. Due to this rule the agent shows the desired behavior and emits a beacon in every scheduled sense-deliberate-act cycle.

The RAS is responsible for sensing the emitted beacons. Therefore the agent’s definition, which can be taken from Code Listing 3 b), includes a perceptor for this task. This perceptor is named `BeaconPer` (line 1 in Code Listing 3 b)) and is executed at the beginning of the sense-deliberate-act cycle. If a beacon broadcast is received by this perceptor, the belief state “beaconReceived” is added to the belief set (see appendix c.1.2 for more information). The belief state includes two parameters: the first parameter determines the ID of the destined agent, which should process the measured signal strength and the second parameter is the signal strength itself. The agent rule defines that if this belief state is considered to be true, the action “inform” should be performed with the same parameters (line 3 in Code Listing 3 b)). This action is provided by the `InformActuator`(line 2 in Code Listing 3 b)), which is part of the middleware and which is used to send messages over the MTS of AFME. These messages are received from the RAC and the SCA.
1) `act BeaconAct;`  
2) `sendBeacon>transmitBeacon;`  

---

1) `per BeaconPer;`  
2) `act InformActuator;`  
3) `beaconReceived(?agent,?txpwr) > inform(?agent,?txpwr);`  

---

1) `per MTSPerceptor, PositionModPer;`  
2) `act InfoReceiveSCAAct, InformActuator;`  
3) `mod posMod = PositionModule;`  
4) `message(inform, sender(?agt,?addr),?msg )> receiveIncomingInfo(?agt,?addr,?msg);`  
5) `coordinator(?agent), newTargetPos(?pos) > inform(?agent,?pos);`  

---

1) `per MTSPerceptor, CoordinatorModPer;`  
2) `act InfoReceiveRACAct, InformActuator;`  
3) `mod corMod = CoordinatorModule;`  
4) `message(inform, sender(?agt,?addr),?content )> receiveIncomingInfo(?agt,?addr,?content);`  
5) `info (?agent,?msg) > inform(?agent,?msg);`  

---

**Code Listing 3** AFME definitions of all AFME based agents for the tracking application  
   a) RAT, b) RAS, c) SCA and d) RAC

Both agent definitions are shown in Code Listing 3: c) for the SCA and d) for the RAC. In order to receive the messages from the MTS, both agents use the MTSPerceptor, which is also provided from AFME (line 1 in Code Listing 3 c) and d)). This perceptor adds the state “message” to the agent’s belief set. This belief state includes three parameters: the message type, the ID of the sending agent and the message itself. As mentioned in section 3.3 of this thesis, it is not possible to share data between actuators and perceptors directly. Hence the functionalities of RAC and of SCA are implemented as modules, which act on the data provided by the actuators that handle the “receiveIncomingInfo” action. These actuators are implemented in the classes InfoReceiveSCAAct of the SCA and InfoReceiveRACAct of the RAC. Furthermore, both agents use perceptors to perceive data from their modules and to update their sets of belief states.

For the PositionModule, which performs the tracking algorithm and which is executed by a SCA in the role of the master, the perceptor is implemented in the class PositionModPer.
perceptor receives new target positions that are calculated from the module and extends the belief set with a new state “newTargetPos”. This state contains the position of the target object. If the belief state “newTargetPos” is considered to be true, the agent decides in the reasoning step of the sense-deliberate-act cycle that the action “inform”, again provided by the InformActuator, should be performed. According to the second rule (line 5 in Code Listing 3 c) in the SCA’s definition there is a second state, the state “coordinator”, which has to be true for an execution of this “inform” action and which provides the address information of the RAC as parameter.

The RAC receives messages that contain the signal strength readings and supplementary messages that include the target positions. These messages are handled by its actuator, InfoReceiveRACAct, in order to process incoming MTS traffic. Signal strength readings are forwarded to the coordinator module to manage the roles of the SCA in the WSN. Due to movement or the activation of the target, the role of a SCA might need to be changed. If this is the case, the RAC perceives all necessary information via the module. The RAC then informs the affected agents to switch their roles and to change their parameters. For example, if the master is located on a different node due to target movement, the address of the agent that acts as new master is a parameter for SCA agents in the role of a slave. This functionality is provided by CoordinatorModPer perceptor and MAPS’s InformActuator. Incoming target positions are stored in the database through a common interface which is used in all prototypes of this application.

Figure 10 shows a class diagram, which contains all used actuators and perceptors of the RAC. The figure illustrates the relation of inheritance between the implemented classes and the classes provided by AFME API. In the class diagram can be seen that both actuators are subclasses of the Actuator class that the perceptors are subclasses of the Perceptor class and that the CoordinatorModule is also a subclass of the Module class. This is true for every actuator, perceptor and module in all AFME based implementation. The class diagram also illustrates the methods, which had to be implemented to substitute the abstract definitions of the superclasses.

To fulfill the requirement of an almost similar implementation on all agent platforms, the internal structure of the implemented AFME modules is designed to function as a state machine. This programming model was chosen because it allows an almost similar implementation on the other platforms. This is due to the fact that MAPS agents are implemented according to event- and state-programming paradigms, as described in chapter 3.4. It is also possible to implement JADE based agents following this paradigm.
In contrast to AFME, MAPS cannot be executed on a base station (see chapter 3.4) and it cannot run on other VMs, because of its dependency on the Squawk VM isolates. Therefore, a normal Sun Spot has to host the RAC agent and has to act as sink node of the WSN.

The class diagram in Figure 11 shows the most important classes involved when implementing a MAPS based application. The MAP is represented through a class called MobileAgentServer with its interface IMobileAgentServer that utilizes the MobileAgentExecutionEngine through its interface. The MobileAgentExecutionEngine is the core component of the middleware. MAPS agents, which extend the Agent class provided by MAPS and which provides the agents with the requirements for the executions as Squawk VM Isolates, are created and started from the MobileAgentExecutionEngine. These Isolates are managed by the InterIsolateServer, both provided by the SPOT SDK. For the implementation of a MAPS agent at least two classes have to be created. One class represents the agent itself and the other class implements a plane that contains the state machine, which defines the agent’s behavior. As presented in chapter 3.4, MAPS supports role based programming that is achieved via the multi-plane state machine implementation. Therefore, it would be possible to implement the three roles of
SCA within a three-plane state machine, but for a higher consistency in the different implementations, every MAPS based agent in this work utilizes a single-plane state machine.

Figure 11 Class diagram of the MAPS agent platform and the implemented class of the RAT

The class diagram in Figure 11 also contains the agent and the agent plane classes implementing the RAT agent. The implementation of these two classes can be seen in appendix c.2.1 and c.2.2. As it can be seen the class RATAgent, representing the agent, includes a static main function which is called when the isolate responsible for the execution of the agent starts. The implemented state machine of the RAT has two states, in which one of them is responsible by the setup state in which the initialization of the state machine and dependencies is realized. In this case a timer is created, responsible for the scheduled and repeatedly creation of a timed event. After the initialization a state change moves the agent in the work state, in which on every timer event a beacon is created. A UML state diagram of this state machine is shown in Figure 12.
The state machine that realizes the desired behavior for the RAS agents is illustrated in Figure 13. The implementation is similar to the RAT’s state machine. Equally, a timer is used to periodically create timed events while staying in one Work state. The timed event triggers an attempt to receive a beacon broadcast on the wireless interface. If no beacon is obtained, the attempt runs into a defined timeout. If a beacon is received, its signal strength is measured and message events are transmitted to a local active SCA and the RAC.
The coordinator agent also operates on a similar state machine that consists out of a setup and a work state. If a message containing signal strength information is received via MTS, the agent updates its list of active nodes and sorts it according to the received signal strength. If state changes of SCA are necessary, the agent requests the changes from the SCA via the event based messaging system of MAPS. Furthermore, the agent reacts on message events, which contain the target positions and uses the USB interface to transmit the target positions to a connect PC.

The SCA’s state machine consists out of four states: a Setup state and one state for every possible role of the agent. The UML state diagram corresponding to the SCA’s state machine is depicted in Figure 14. The initialization that is done in the Setup state is followed by a transition into the Sleep state, in which the agent reacts only to requests of the RAC to change its role. In case of such a request, the agent switches its states either from Sleep state to one of the two working states, Master or Slave or the other way around. A direct transition from one of the two working states to the other one is not implemented. In both working states the agent can receive parameters from the RAC, which are necessary to fulfill its task. For a SCA acting in the role of a slave, the parameter includes the ID and address of the SCA that acts as master. Otherwise the parameter consists out of the addresses and positions of the nodes, which host the SCA acting as slave and the position of the master’s node.

![State diagram of the state machine implementing the MAPS based SCA’s behavior](image)

Figure 14 State diagram of the state machine implementing the MAPS based SCA’s behavior
JADE

JADE based agents are implemented with a behavior based approach. Therefore the JADE API provides an abstract object form type Behaviour, which has to be extended when implementing new types of behaviors. But the implementation of a custom subclass of Behaviour is not mandatory because JADE also provides predefined behavior classes for usage or extension. In this work only two types of predefined behaviors are utilized. A cyclic behavior implemented in the class CyclicBehaviour. This subclass of Behaviour, repeatedly executes an implemented method similar to a method call in a loop. The second behavior is implemented in the JADE class TickerBehaviour, where an implemented method is repeatedly scheduled for execution after a give time span. Due to the requirement of comparable implementations, the agent behaviors developed for this prototype are internally implemented as state machines, driven from a CyclicBehaviour or TickerBehaviour. Their internal structure is almost identical to behavior defining planes of the MAPS implementations. The RAT’s behavior for example is implemented as TickerBehaviour. The implementation of the JADE RAT agent can be taken from appendix c.3.1.
Agent Based User Interface

![Screenshot of implemented Android user interface.](image)

The GUI of the tracking application is implemented with the JADE middleware. The main task of the GUI is to visualize the target object’s position and the path which it took through the WSN. The growing popularity of mobile applications and smartphones has led to the decision to implement the GUI for the mobile object tracking application on the Android platform. This choice induced the implementation of a mobile application prototype, which features mobile real time tracking and monitoring. A screenshot of the implemented Android user interface can be seen in Figure 15. In the visualization of the WSN in the GUI, sensor nodes and their positions are represented through green dots. The actual position of the target is indicated with a small red dot. Former target positions are illustrated with blue dots. The track, on which the target has moved, is visualized with yellow lines between the target positions.

The implementation of the GUI requires independence from the MAP, on which the running WSN tracking application is implemented, because the GUI can then be used with AFME and MAPS implementations of Centralized Mobile Object Tracking application. Hence, the agents that are relevant for the GUI are executed on a JADE platform, which is independent from the WSN component of the application, although JADE could host the GUI relevant agent on the same JADE platform as the other JADE agents. If the GUI relevant agents were executed on the same JADE platform, a direct communication between the GUI agents and the JADE implementation of the RAC agent would be possible, but the GUI could not be operated with AFME and MAPS.
The GUI utilizes the split container execution mode for the agents that are hosted on an ANDROID device, just as in the tracking application implemented with JADE. Two cooperating agents are developed to realize an agent based GUI, because agent mobility is not supported by split containers[30]. The architecture of agent based GUI implementation is depicted in Figure 16.

A GUI Host Agent (GHA) is performed on a main container that runs on the PC connected to the WSN. The PC provides the data base management system (DBMS) which hosts a database for the Centralized Mobile Object Tracking application. The GHA is responsible for the query of this database. It uses the same database interface as the RAC agents and the MAPS host tool to store the tracking results. More information about the realization of the database and its interface is presented in the next subchapter. After the database query the GHA informs the second type of GUI agents, which is called GUI Android Agent (GAA) and which is executed on ANDROID devices, via JADE MTS on request.

The GAA has to request the information about the track of the target positions from the GHA. After the first request the GHA also informs the GAA continually about updates in the target positions. The GAA receives the information via MTS and forward it to an ANDROID Activity for visualization. An Activity represents a view or page of an ANDROID application. The development of ANDROID applications is not the focus of this thesis. Therefore, interested readers are referred to [32] for more information. In case a GAA is terminated, the agent informs the GHA to now lower send target location updates.

![Figure 16 Architecture of the implement agent based Android user interface](image)
Database and Interface

The main task of the MySql database is to store the positions of the target object. This is accomplished over Java Database Connectivity (JDBC), which provides an API to access database with Java. Additionally to the main task, the database also contains positions of all sensor nodes forming the WSN.

The interface to the database is implemented through a single class, which is designed according to the well known singleton design pattern [33] and which allows an object based access and storage of information. Therefore, the interface can be used without any local references of the object. To store a new target position in the database only one line of code is necessary:

```java
Common.getInstance().getDB().add(new Position(x,y));
```

As a result of the object orientated interface, the creation of data classes that simply include data is necessary. In the code line exemplified above a class, which only contains variables for a value of x and y and which represents a target position, is implemented. The free DBMS MySql hosts the database of the Centralized Mobile Object Tracking application. The database also includes tables to store results of measurements. This allows the realization of an automated process that stores the results for the comparisons of the tested MAPs.
4.2 Decentralized Search

The Decentralized Search application is developed on the existing implementations of the centralized tracking application. Some components are modified in the Decentralized Search. The target object, its platform and its agent can be reused with one modification: the frequency, in which the beacons are emitted, is 0.66 Hz.

The RAS of the Centralized Mobile Object Tracking application can also be employed in the Decentralized Search. Its behavior differs only in the recipients that receive the information of the beacon frames and in the handling of the virtual pheromone mark. The difference is that a RAS only informs the CSA of current value of the virtual pheromone mark via local message broadcast, if the CSA is present on the sensor node of the RAS. Furthermore, the RAS stores the value of the pheromone reading and decreases it over time to simulate the fading of the mark, similar to a pheromone trail in the real world.

AFME

![UML class diagram of an AFME platform supporting agent mobility.](image)

Due to the fact that the Decentralized Search utilizes agent mobility in its search algorithm, the definition of the AFME platform for sensor nodes has been modified to support agent mobility. Figure 17 depicts a UML class diagram that represents the generated JAVA classes, which result from the changes in the definition of the platform. As it can be seen the JAVA classes that represent such a platform implement the MigrationPlatform interface additionally to the Platform interface. This MigrationPlatform interface provides the necessary functionality for the agent mobility to the platform. Furthermore, a platform supporting agent mobility has to provide a service to manage agent migration. In this case, corresponding service implemented in the class RadioMigManager
has to be used for migration on Sun Spots over the wireless interface. This required additional service is illustrated in the class diagram in Figure 17. In contrast to the previous used MTS of AFME, which offers system wide message transportation, the MTS in the Decentralized Search is only used for node internal communication. Therefore, the MTS in this application does not listen to any incoming messages on the wireless interface. The sensor node, on which the migration platform is executed, hosts the modified version of the RAS. This informs the mobile CSA about the current value of the pheromone mark. To be able to fulfill its task the CSA receives the pheromone readings only from a local RAS. AFME does not support broadcast messages and therefore the RAS has to register the CSA so that the CSA can receive information about the pheromone trail from the RAS.

```
1) per MTSPerceptor, SW01Per;
2) act InformActuator, MigrateActuator, SearchBestNodeAct, DeregisterAct;
3) searchState(?state),parameter(?p),message(inform,sender(?agt,?addr),?lread),visitedNodes(?oldNodes),highestSearchReading(?hr4s)
   > search4bestNode(?state, ?p, ?lread, ?oldNodes, ?hr4s);
4) deploy,destAddr(?destaddr) > par(deregisterAtRAS , migrate(?destaddr,null));
```

Code Listing 4 Agent definition of the AFME Decentralized Search agent.

The definition of the CSA can be taken from Code Listing 4. After the CSA is resumed or started on a node, the registration of the CSA to the RAS is initiated via the already known InformActuator. To migrate to another node, CSA triggers the action “migrate” provided from the MigrateActuator that is part of the AFME API (line 2 in Code Listing 4). The rule for the migration (line 4 Code Listing 4) declares that if both a state deploy and a state destAddr, which contains the address of the destination node, are believed to be true, two action should be performed. First, the CSA unsubscribes itself from the RAS with the action “deregisterAtRAS” in order to stop the transmission of messages. This action is provided by an actuator named DeregisterAct. Second, the migration of the CSA is executed. Due to the fact that CSA is a mobile agent, all inner states have to be available on the agents destination node and they have to be in the agent’s belief set in order to be transmitted. Hence, the dimension of the rule that is responsible for driving the search algorithm implemented as state machine contains all necessary parameters. This rule is illustrated in line 3 of Code Listing 4. For more details on the state machine see the next subsection. The states searchState, parameter, visitedNodes and highestSearchReading are contain data needed for the search algorithm, e.g. visitedNodes contains a list with already visited nodes which is required to avoid that a node is again chosen as migration destination(line 3 in Code Listing 4 Agent definition of the AFME
Decentralized Search agent.). If the perceptor SW01Per perceives a push on the first button on a Sun Spot, it starts the Decentralized Search by adding the required state to the belief set.

MAPS

Similar to the AFME based application, the MAPS application reuses the same modified components and agents as in the Centralized Mobile Object Tracking application. Therefore, for the discussion of the RAT and RAS implementations see subchapter 4.1 MAPS. In contrast to AFME, MAPS supports broadcast message events (local and remote). Hence, the implementation of the registration process as described in the previous sub-section is not necessary.

After the initialization is done in a Setup state, the CSA is in a state in which the agent handles incoming message events, as illustrated in Figure 18. If the agent receives a pheromone reading, the reading is compared to the highest reading that has been achieved so far. If the reading is of a higher value than the previous one, the agent stores the value and the node address in its internal states. The CSA checks an internal list of nodes, which should be visited by the CSA. If there are nodes in the list, the agent migrates to the next node in the list. If the list is empty, the agent checks whether the current hosting node is the node with the highest pheromone mark so far. In this case the list of nodes that has to be visited is extended with unvisited neighboring nodes of the current hosting node. If the pheromone mark is not the highest one, the agent migrates to the node with has the highest pheromone mark so far. After a migration process the agent is in a New node state, in which the new hosting node is added to the list of already visited nodes. Afterwards the agent transits into the Read state in order to continue the execution of the search. Under the circumstance that the CSA cannot obtain any new neighboring nodes while being executed on the node with the strongest pheromone mark, the agent transits into a final Finished state and ends the search algorithm.
Figure 18 UML state diagram modeling the behavior of a MAPS based Decentralized Search agent.
5 Experiments and Results

For the evaluation of the MAP performances in the execution of test prototypes, four criteria are identified. The utilization of a node’s CPU and its memory are chosen to be the first criteria, because they represent a node’s degree of capacity. Energy consumption is a very important factor for embedded wireless devices due to their limited power supply. Thus, the energy consumption is also identified to be one of the evaluation criteria. Directly responsible for a certain part of a node’s energy consumption is the degree of usage of the wireless interface. Therefore, the network traffic created by the different MAP is also used as criteria. In regard to agent mobility, the migration time is the last of the chosen criteria. To be able to evaluate several MAPs according to their performances, a test scenario that realizes agent mobility is defined to measure the chosen criteria in a migration context.

The distributed characteristics of the developed test system and the different states, in which its components can operate, require a definition of the test configurations for both applications, the Centralized Mobile Object Tracking and the Decentralized Search. Configuration 1 represents an empty node as reference, on which no agent or agent platform is executed. A RAT and its platform are hosted according to Configuration 2. Due to the two different applications, which lead to different test scenarios, a distinction between the configurations has to be made. This distinction refers to all configurations that include a higher number than 2. In Configuration 2 the only difference in the implementation of the two applications is the frequency of the emitted beacons. For the Centralized Mobile Object Tracking application in the Configuration 3T a sensor node hosts one RAS and one inactive SCA. This leads to the conclusion that no target node is present in the range of the sensor node. Hosted active SCAs in the role of master or slave with an active target object in range are found in Configuration 4T and Configuration 5T.

Four different possible configurations are used for the Decentralized Search application. In Configuration 3S a sensor node executes RAS including its platform. In Configuration 3S no target node is in range in contrast to Configuration 4S, in which a target node is in the range. These two possibilities also exist, if the sensor node is additionally hosting a registered but inactive CSA. This leads to Configuration 5S, which has a registered but inactive CSA and in which the target is not in range. In Configuration 6S the CSA is also registered and inactive, but the target is in range. A migrating CSA has the focus on the Migration Test Configuration, because in this configuration three sensor nodes are arranged in a row and the CSA migrates from the first one over the second one to the third one.
All of these configurations are illustrated in Figure 19. They represent fixed setups of nodes for the conducted experiments. Due to these configurations an extraction of metrics is achieved that represents information about the execution of the two test applications. The metrics are extracted during system runtime by an instrumented code. This instrumented code is realized as a separate thread. It is executed pseudo parallel to the relative implementations and it measures the criteria over a defined time span.

---

**Configuration 1:**

---

**Configuration 2:**

---

**Tracking Application:**

**Configuration 3T:**

**Configuration 4T and 5T:**

---

**Search Application:**

**Configuration 3S:**

**Configuration 4S:**

**Configuration 5S:**

**Configuration 6S:**

---

**Migration Test Configuration:**

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*Figure 19 Test configurations for experiments*
The instrumented code is implemented in several classes designed to interfere as less as possible with each other’s measurements. An additional class, in which the separate thread for the execution measurements is started, is responsible for managing the execution and the timing of the different measurements. To minimize side effects between different measurements, several precautions were taken, e.g. manual calls to the VM’s garbage collector for the memory measurement and several pauses were performed. These pauses allow the system to settle or the garbage collector to finish.

In addition to that the measurement managing class is also responsible for transmitting the results of measurements to a nearby sink node via radiogram broadcast. It should be noted that the radiogram functionality is provided by the libraries of the SPOT SDK and therefore has no dependencies to any MAPs. Through an independent implement process on PC the measurement result is received and forwarded to the database for storage and data preparation.

The deployment of the nodes in the WSN can be seen in Figure 20 as well as the minimu communication range for one Sun Spot. The figure shows in a) the deployment of the sensor nodes for the Centralized Mobile Object Tracking application. The nodes are arranged in a way that they form equilateral triangles with a side length of two meters. The deployment of the sensor nodes for the Decentralized Search is illustrated in figure 20 b). In this application four nodes form a two by four meter rectangle.
Figure 20 Node deployment in the WSN for a) Centralized Mobile Object Tracking b) Decentralized Search
5.1 CPU Utilization

The first criterion for the evaluation is the CPU utilization of the hosting node in a certain state. The SPOT API does not provide a method for monitoring the CPU. Therefore, an energy saving feature of SPOT is utilized to determine the CPU load. The CPU of a Sun Spot is automatically set to a sleep mode to preserve energy, in case the CPU idles. So it is possible to measure a Sun Spot’s CPU load through the proportion of the runtime to the time span, in which the CPU is in sleep mode. A Sun Spot has also a third operation mode, the so called Deep Sleep mode, but this mode was deactivated for these CPU measurements.

The results of the CPU measurements can be seen in Table 1. The cyclic emission of the beacon frames require more processing time in the Centralized Mobile Object Tracking application than in the Decentralized Search application. The reason for this difference is the higher frequency of the beacons. As it can be seen in Table 1 JADE has the lowest load for configuration 1 to 3T in the Centralized Mobile Object Tracking application, but in configuration 4T and 5T JADE utilizes the CPU more than the others MAPs. This can be explained by the split execution mode of JADE. Local execution is therefore very efficient, but if it comes to an interaction with other agents, JADE has a higher demand for processing time in comparison to the AFME and MAPS. This is also displayed in the results of Configuration 5T, in which JADE needs almost twice as much processing time than the other MAPs. The comparison between AFME and MAPS in both applications show that MAPS has a lower CPU utilization than AFME. This is the result of the more complex architecture of AFME.

Table 1 Measured CPU utilizations in % of the Centralized Mobile Object Tracking application and the Decentralized Search application in comparison

<table>
<thead>
<tr>
<th>CPU Load in %</th>
<th>Centralized Mobile Object Tracking</th>
<th>Decentralized Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AFME</td>
<td>MAPS</td>
</tr>
<tr>
<td>Configuration 1 (Empty node)</td>
<td>0,26</td>
<td>0,26</td>
</tr>
<tr>
<td>Configuration 2 (Target Sunspot)</td>
<td>16,56</td>
<td>12,56</td>
</tr>
<tr>
<td>Configuration 3T (Inactive SCA)</td>
<td>3,42</td>
<td>3,63</td>
</tr>
<tr>
<td>Configuration 4T (SCA as Slave)</td>
<td>22,43</td>
<td>20,89</td>
</tr>
<tr>
<td>Configuration 5T (SCA as Master)</td>
<td>31,87</td>
<td>26,95</td>
</tr>
<tr>
<td>Configuration 6T (RAS no Target)</td>
<td>10,77</td>
<td>5,28</td>
</tr>
</tbody>
</table>

47
5.2 Memory

Java Applications executed in a Java VM have to use the VM memory management that utilizes a garbage collector to free objects, which are no longer referenced. To save energy, the garbage collector’s collection process implemented in the Squawk VM is started as soon as almost all memory resources are reserved. Although it is a great feature to preserve energy, it is most obstructive for memory monitoring. To overcome this effect, manual calls to the garbage collector are used to free unused memory before a measurement is taken.

Table 2 Memory utilization during execution (in kilo bytes) of the Centralized Mobile Object Tracking application and the Decentralized Search application in comparison

<table>
<thead>
<tr>
<th>Used Memory in kilo bytes</th>
<th>Tracking Application</th>
<th>Decentralized Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AFME</td>
<td>MAPS</td>
</tr>
<tr>
<td>Configuration 1 (Empty node)</td>
<td>86</td>
<td>86</td>
</tr>
<tr>
<td>Configuration 2 (Target Sunspot)</td>
<td>91</td>
<td>109</td>
</tr>
<tr>
<td>Configuration 3 (Inactive SCA)</td>
<td>110</td>
<td>130</td>
</tr>
<tr>
<td>Configuration 4 (SCA as Slave)</td>
<td>124</td>
<td>132</td>
</tr>
<tr>
<td>Configuration 5 (SCA as Master)</td>
<td>138</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>131</td>
</tr>
</tbody>
</table>

The results of the memory related measurements are shown in Table 2. As it can be seen, MAPS is the one MAP that uses the most memory and hence shows the worst results. In comparison with AFME, which has a modular architecture, MAPS has a monolithic structure and instantiates all platform components in the beginning, even if they are not needed. JADE shows an almost constant low memory need regardless of the test configuration. The reason for this is the split execution mode, in which the backend of the agent container is not executed on the mobile device. Thus, JADE is the best memory preserving MAP in the centralized object tracking application. In the Decentralized Search application AFME presents the best results, because of its modular architecture.
5.3 Energy

The SPOT API provides access to the information about the remaining capacity of the battery in milliamp hours. Therefore, this information is used to determine the energy consumption of the Spot. Three measurements with a different run time were performed: 10 sec, 30 sec and 300 sec. The results of these measurements are then used to calculate the energy consumption.

Table 3 Energy consumption of the implementations.

<table>
<thead>
<tr>
<th>Consumed Energy (mA)</th>
<th>Tracking Application</th>
<th>Decentralized Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AFME</td>
<td>MAPS</td>
</tr>
<tr>
<td>Configuration 1</td>
<td>54,00</td>
<td>54,00</td>
</tr>
<tr>
<td>(Empty node)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration 2</td>
<td>60,30</td>
<td>68,57</td>
</tr>
<tr>
<td>(Target Sunspot)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration 3</td>
<td>69,25</td>
<td>79,66</td>
</tr>
<tr>
<td>(Inactive SCA)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration 4</td>
<td>84,49</td>
<td>98,26</td>
</tr>
<tr>
<td>(SCA as Slave)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configuration 5</td>
<td>86,34</td>
<td>81,60</td>
</tr>
<tr>
<td>(SCA as Master)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results of the nodes energy consumption are presented in Table 3. Differently to the presented results for the CPU utilization, in which JADE has high values for test configurations that involve agent communication, the energy consumption of JADE is quite moderate in comparison to the processor load. Together with the CPU load results that, due to the more efficient coding in the backend frontend connection as mentioned in Section 3.5, a compression algorithm is reducing the usage of the wireless interface by shrinking the amount of data which has to be transferred.

The influence of the CPU utilization on energy consumption can be seen in Figure 21 which shows a combined diagram depicting the CPU load together with the consumed energy for both applications. Furthermore, the figure visualizes that MAPS needs the most energy of all MAPs in almost every test configuration. This can also be explained with MAPS’s monolithic structure.
Figure 21 Combined diagrams showing CPU load and energy needs for a) Centralized Mobile Object Tracking and b) Decentralized Search
5.4 Wireless Traffic

Statistics about the wireless traffic of a Sun Spot are also accessible through the SPOT API. The measurements are taken for a defined time span. The statistics are created in the Sun Spots low level API and show all packets that are received or sent on the Sun Spot’s wireless interface including packages which are routed through the network. The extracted results of the wireless traffic measurements are presented in Table 4. Due to the fact that all received packages are shown in these results, the higher values for the incoming parts can be explained. Almost the same results for outgoing packages in all implementations were expected due to similar implementation and timing. Yet the received results show a great discrepancy. The high amount of traffic created in JADE implementation can be explained by the split execution mode, because the detour of the connections over the backends is mandatory. This effect is expected to increase with a growing amount of nodes in the WSN, because if a node is out of range and hence has no way of direct communication with the base station, it has to utilize the in SPOT integrated routing mechanism to the node. This results in a higher amount of traffic on the route to the base station.

Table 4 Wireless traffic results for implementations of the Centralized Mobile Object Tracking application.

<table>
<thead>
<tr>
<th>packets per sec.</th>
<th>Master</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration 4 and 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>OUT</td>
<td>IN</td>
</tr>
<tr>
<td>AFME</td>
<td>15,44</td>
<td>0,43</td>
</tr>
<tr>
<td>MAPS</td>
<td>10,47</td>
<td>1,23</td>
</tr>
<tr>
<td>JADE</td>
<td>22,02</td>
<td>13,33</td>
</tr>
</tbody>
</table>
5.5 Migration

The setup for the migrations test involves three nodes, which are arranged in a row. The migrating agent (CSA) is started on the first node and it is supposed to migrate to the third node over the second node. In this setup the three possible migration states are included: an agent leaves a node, it joins and leaves a node and it only joins a node. While the migration process is executed, measurements of CPU and energy consumption are taken on every involved sensor node.

As additional criteria for these experiments the duration of a migration is utilized. The AFME based agent completed the first migration after 3.8 sec and arrived on the target node after 8.2 sec. The MAPS based counterpart needed 39 sec for the first hop and finished the test run after 81 sec on the destination node. The reason for this huge difference is on the one hand that the MAPS utilizes the earlier mentioned Isolates for the migration process. On the other hand, AFME only transfers the agent’s state and the class names of the needed components on the destination node.

In this test scenario, results for the CPU utilization and energy consumption are also extracted. They are shown in Table 5. In the CPU utilization and also in the energy consumption the results of the MAPS are constantly higher than of the AFME. It can be seen that the energy consumption and CPU load are higher on node 2 than on the two other nodes. The reason for that is that node 2 participates in two migrations when the agent is arriving on the node and when the agent is leaving the node.

<table>
<thead>
<tr>
<th></th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (%)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFME</td>
<td>12.47</td>
<td>19.77</td>
<td>14.86</td>
</tr>
<tr>
<td>MAPS</td>
<td>24.38</td>
<td>40.84</td>
<td>21.89</td>
</tr>
<tr>
<td>Energy (mA)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFME</td>
<td>78.81</td>
<td>87.13</td>
<td>73.76</td>
</tr>
<tr>
<td>MAPS</td>
<td>93.34</td>
<td>99.56</td>
<td>90.96</td>
</tr>
</tbody>
</table>
6 Discussion

The approach in this thesis for an evaluation of the available MAPs supporting Sun’s SPOT has led to the following conclusions for the cooperating agent and mobile agent model. For the cooperating agent model, as utilized in the Centralized Mobile Object Tracking application, AFME would be chosen for a realization of this WSN application. The reason for this choice is that AFME is able to use the limited hardware resources more efficient than MAPS. Although AFME has a slightly higher CPU utilization than MAPS, it requires less energy and less memory than MAPS in almost every test configuration. JADE disqualifies itself through the limitation and disadvantages of the split execution mode, which results in a higher amount of traffic. Due to the expectation of an even higher wireless link utilization with a larger amount of nodes, JADE is not suitable for the WSN domain. But JADE would be the first choice for the implementation of a MAS operating in a distributed heterogeneous environment, because it has a great variety of supported platforms. In the case of a Decentralized Search application, which is an example for a mobile agent based application, AFME would be chosen due to its faster and more efficient migration process.
7 Related Work

In the scientific field of Wireless Sensor Networks middleware is an active research topic. In their article [34], Wang et al. present a survey of middleware for the WSN domain. A reference framework is proposed for the analysis of the features of a middleware. Furthermore, the middleware for WSN are evaluated and classified with a feature taxonomy provided. In contrast to this thesis, in which the agent based programming paradigm is focused, Wang et al. also consider several other paradigms like e.g. database and publish/subscribe in their survey.

The evaluation of MAPs is the topic of the article of Alberola et al. in [13]. Three MAPs are compared in their performance of the agent communication features of a MAP in regard to scalability. Whereas this thesis concentrates on the WSN domain with an embedded hardware platform, Alberola et al. examine MAPs on usual PCs as hardware platform. Therefore and due to the focus on agent communication features, the used evaluation criterion were chosen only in regard to agent communication.

In [35], Chen et al. provide an overview of main applications of mobile agent systems in WSN and research issues in regard to their realization. According to the authors, target tracking is one of these main applications. The article shows two different kinds of target tracking which have certain similarities to the tracking applications developed for this thesis. The first proposed tracking application uses also triangulation like the Centralized Mobile Object Tracking in this thesis, but instead of fixed cooperating agents, the tracking is realized with mobile cooperating agents. The second kind of tracking application shown in the article is an object recognition application, in which a mobile agent follows the path of a target object, similar to the Decentralized Search. Furthermore, the authors present four core design components for the design of a mobile agent based WSN system.

The analysis of the performance between the classical client/server communication model and the agent based communication in WSN for health care applications is the topic of [36]. Barnes et al. come to the conclusion that the client/server model has a small performance advantage over agent based communication. According to the authors, this is the result of the limitations of the used MAP and hardware platform. Similar to this thesis, the author used Jade-LEAP on emulated Sun Spots as platform. Due to the fact that Jade-LEAP does not support agent mobility on MIDP devices, the authors had to change their test application. They worked with fixed communicating agents instead
of the desired mobile agents, equally to the necessary changes in the Centralized Mobile Object Tracking application.
8 Conclusion and Future Work

Due to progress and development in technology, wireless sensor nodes can be manufactured cost effectively and in large numbers nowadays. This availability and the possibility to create a network of cooperating nodes have led to a rapidly growing popularity of a technology named Wireless Sensor Networks (WSN). To overcome their disadvantage of a challenging complexity in programming of a WSN, software agents have been identified as a suitable programming paradigm. This agent based approach commonly uses a middleware for the execution of the software agent.

The goal of this thesis was the implementation of functional comparable prototypes for a given set of middleware for two applications, the Centralized Mobile Object Tracking and the Decentralized Search. The extraction of measurements about the performance and resource needs of every middleware in the test set and the analysis of the measurements to determine the suitability of each middleware for the specific applications were further goals in this work.

The implemented applications were a Centralized Mobile Object Tracking application based on the multi agent systems in a centralized topology and a Decentralized Search application which uses the mobile agent system concept. Both applications require different properties and features from the agent platform on which they are implemented. Through the analysis of the results extracted in the measurement, the most suitable middleware for both applications, the Centralized Mobile Object Tracking application and the Decentralized Search, was determined to be AFME.

There are some points for further developments in regard to the implemented prototypes as well as to the extension of the set of evaluation criteria. The reaction time of an agent and the overall system to events could be one interesting criteria to examine. The Centralized Mobile Object Tracking prototypes could be modified to research performance differences in comparison to the mobile agent approach of this application which is described in [15].

For the ANDROID based GUI an expedient feature would be a remote interface that allows the configuration of the overall system parameters, like the WSN setup for the Centralized Mobile Object Tracking application. A method of interaction with agents to provoke a certain behavior could be useful for more complex measurements in more dynamic test configurations.

The development of a more generic defined set of test applications, which could be applied to all mobile agent platforms regardless of the used hardware platform or programming language, could be a further field of study.
References


[38] Fortino, Giancarlo, et al. MAPS: A Mobile Agent Platform for WSNs based on Java Sun Spots. s.l. : Department of Electronics, Informatics and Systems (DEIS), University of Calabria.


**Appendix**

a. **Glossary**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFAPL2</td>
<td>Agent Factory Agent Programming Language 2</td>
</tr>
<tr>
<td>AFME</td>
<td>Agent Factory Micro Edition</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BDI</td>
<td>Believe Desire Intention</td>
</tr>
<tr>
<td>CA</td>
<td>Collaborative Agent</td>
</tr>
<tr>
<td>CAM</td>
<td>CA_Master</td>
</tr>
<tr>
<td>CAS</td>
<td>CA_Slave</td>
</tr>
<tr>
<td>CLDC</td>
<td>Connected Limited Device Configuration</td>
</tr>
<tr>
<td>CSA</td>
<td>Collaborative Search Agent</td>
</tr>
<tr>
<td>DBMS</td>
<td>DataBase Management System</td>
</tr>
<tr>
<td>CSAM</td>
<td>CA_Master</td>
</tr>
<tr>
<td>CAS</td>
<td>CA_Slave</td>
</tr>
<tr>
<td>CLDC</td>
<td>Connected Limited Device Configuration</td>
</tr>
<tr>
<td>CSA</td>
<td>Collaborative Search Agent</td>
</tr>
<tr>
<td>CSAM</td>
<td>CA_Master</td>
</tr>
<tr>
<td>GAA</td>
<td>GUI Android Agent</td>
</tr>
<tr>
<td>GHA</td>
<td>GUI Host Agent</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>JADE</td>
<td>Java Agent DEvelopment framework</td>
</tr>
<tr>
<td>J2EE</td>
<td>Java 2 Enterprise Edition</td>
</tr>
<tr>
<td>J2ME</td>
<td>Java 2 Micro Edition</td>
</tr>
<tr>
<td>J2SE</td>
<td>Java 2 Platform, Standard Edition</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java DataBase Connectivity</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Request</td>
</tr>
<tr>
<td>LEAP</td>
<td>Lightweight Extensible Agent Platform</td>
</tr>
<tr>
<td>MAP</td>
<td>Multi-Agent Platform</td>
</tr>
<tr>
<td>MAPS</td>
<td>Mobile Agent Platform for Sun Spots</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi-Agent System</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile</td>
</tr>
<tr>
<td>MTS</td>
<td>Message Transport Service</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>RA</td>
<td>Resident Agent</td>
</tr>
<tr>
<td>RAC</td>
<td>RA_Coordinator</td>
</tr>
<tr>
<td>RAS</td>
<td>RA_SensorNode</td>
</tr>
<tr>
<td>RAT</td>
<td>RA_Target</td>
</tr>
<tr>
<td>RSSI</td>
<td>Received Signal Strength Indicator</td>
</tr>
<tr>
<td>SCA</td>
<td>Static Cooperative Agent</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SPOT</td>
<td>Small Programmable Object Technology</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicles</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Networks</td>
</tr>
</tbody>
</table>
b. **Software Versions**

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun SPOT SDK</td>
<td>yellow-101117-1</td>
</tr>
<tr>
<td>AFME</td>
<td>3.3</td>
</tr>
<tr>
<td>MAPS</td>
<td>1.1</td>
</tr>
<tr>
<td>JADE-LEAP (WSN application)</td>
<td>4.0.1</td>
</tr>
<tr>
<td>JADE-LEAP (Android GUI)</td>
<td>4.1</td>
</tr>
<tr>
<td>Sun Java Wireless Toolkit for CLDC</td>
<td>2.5.2</td>
</tr>
</tbody>
</table>
c. Source Code

1. AFME

1. Beacon Actuator

public class BeaconAct extends Actuator {

    RadiogramConnection txConn = null;
    private static final String BROADCAST_PORT = "42";
    private static final int RADIO_BEACON_PACKET = 42;

    public BeaconAct(AffectManager manager) {
        super(manager, "transmitBeacon");
    }

    public boolean act(FOS action) {
        try {
            txConn = (RadiogramConnection) Connector.open("radiogram://broadcast:" + BROADCAST_PORT);
            txConn.setMaxBroadcastHops(1); // don't want packets being rebroadcasted
            Datagram xdg = txConn.newDatagram(txConn.getMaximumLength());
            xdg.writeByte(RADIO_BEACON_PACKET);
            txConn.send(xdg);
        } catch (IOException ex) {
            // ignore
        } finally {
            if (txConn != null) {
                try {
                    txConn.close();
                } catch (IOException ex) {
                    //
                }
            }
        }
        return true;
    }
}
2. Beacon Perceptor

```java
class Check4BeaconPer extends Perceptor {
    private PerceptionManager m;
    private String add;
    private RadiogramConnection rcvConn = null;
    private static final int PACKET_INTERVAL = 250;
    private static final String BROADCAST_PORT = "42";
    private static final int RADIO_BEACON_PACKET = 42;

    public Check4BeaconPer(PerceptionManager perManager) {
        super(perManager);
        m = perManager;
        add = IEEEAddress.toDottedHex(RadioFactory.getRadioPolicyManager().getIEEEAddress());
        add = add.substring(add.lastIndexOf('.') + 1, add.length());
    }

    public void perceive() {
        int q = 0;
        int nothing = 0;

        try {
            rcvConn = (RadiogramConnection) Connector.open("radiogram://:" + BROADCAST_PORT);
            rcvConn.setTimeout(rxtimeout);
            Radiogram rdg = (Radiogram) rcvConn.newDatagram(rcvConn.getMaximumLength());

            try {
                rdg.reset();
                rcvConn.receive(rdg); // listen for a packet
                byte packetType = rdg.readByte();
                if (packetType == RADIO_BEACON_PACKET) {
                    q = rdg.getRssi() + 200;
                }
            } catch (TimeoutException tex) { // timeout
                nothing++;
            } catch (IOException ex) {
                // ignore
            } finally {
                if (rcvConn != null) {
                    try {
                        rcvConn.close();
                    } catch (IOException ex) {
                    }
                }
            }

            if (nothing > 0) {
                // no beacon recived
            } else {
                // beacon recived
                adoptBelief("beaconRecived(agentID(C"+add+"),DB:"+q+"))");
                if (counter++ > 4) {
                    // Send Beacon Info to Coordinator
                    counter = 0;
                    adoptBelief("beaconRecived(agentID(RACoordinator,"
                        + "addresses(radiogram://0014.4F01.0000.71B8:66)),"
                        + q + ")");
                }
            }
        }
    }
}
```
2. MAPS

1. RAT Agent

    public class RATAgent extends Agent {
        public RATAgent(String id, String executionAgentAddress) throws NoSuchMailboxException {
            super(id, executionAgentAddress);
            RATAgentPlane tap = new RATAgentPlane(this);
            multiplaneStateMachine.addElement(tap);
        }
        public static void main(String[] args) {
            try {
                RATAgent ratAgent = new RATAgent(args[0], args[1]);
                while (!ratAgent.isTerminated()) {
                    ratAgent.waitForEvents();
                    ratAgent.run();
                }
            } catch (Exception e) {
                LedsManager.error();
                e.printStackTrace();
            }
        }
    }
2. RAT Agent Plane

```java
public class RATAgentPlane extends Plane {

    private static final byte START = 1, A1 = 2;

    public RATAgentPlane(Agent agent) {
        super(agent);
        this.currentState = START;
        this.agent.startAgent();
    }

    public void eventHandler(Event event) {
        switch (this.currentState) {
            case START:
                if (event.getName() == Event.AGN_START) {
                    this.currentState = A1;
                    Event start = new Event(this.agent.getAgentId(), this.agent.getAgentId(),
                        Event.MSG, Event.NOW);
                    try {
                        start.setParam("msg", "start");
                    } catch (CharNotValidException ex) {
                        ex.printStackTrace();
                    }
                    this.agent.setTimer(true, 500, start);
                }
                break;
            case A1:
                if (event.getName() == Event.MSG) {
                    // Timer fired
                    try {
                        RadiogramConnection txConn = new RadiogramConnection()
                        Connector.open("radiogram://broadcast:");
                        txConn.setMaxBroadcastHops(1);
                        Datagram xdg = txConn.newDatagram(txConn.getMaximumLength());
                        xdg.writeByte(RADIO_BEACON_PACKET);
                        txConn.send(xdg);
                    } catch (IOException ex) {
                        // ignore
                    } finally {
                        if (txConn != null) {
                            try {
                                txConn.close();
                            } catch (IOException ex) {
                            }
                        }
                    }
                }
                break;
        }
    }
}
```
1. RAT Agent

```java
public class RATargetAgent extends Agent {

    private IDemoBoard sensorBoard;
    RadiogramConnection txConn = null;
    private static final String BROADCAST_PORT = "42";
    private static final int RADIO_BEACON_PACKET = 42;

    protected void setup() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException ex) {
        }

        addBehaviour(
            new TickerBehaviour(this, 500) {
                public void onTick() {
                    try {
                        txConn = (RadiogramConnection) Connector.open("radiogram://broadcast:" + BROADCAST_PORT);
                        txConn.setMaxBroadcastHops(1);
                        Datagram xdg = txConn.newDatagram(txConn.getMaximumLength());
                        xdg.writeByte(RADIO_BEACON_PACKET);
                        txConn.send(xdg);
                    } catch (IOException ex) {
                        // ignore
                    } finally {
                        if (txConn != null) {
                            try {
                                txConn.close();
                            } catch (IOException ex) {
                            }
                        }
                    }
                }
            });

        try {
            Thread.sleep(500);
        } catch (InterruptedException ex) {
        }
    }

    protected void takeDown() {
        System.out.println("Agent RATarget" + getAID().getName() + " is finishing");
    }
}
```