2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing

UbiMASS - Ubiquitous Mobile Agent System for Wireless Sensor Networks

Faruk Bagci, Julian Wolf, Benjamin Satzger, and Theo Ungerer
Institute of Computer Science
University of Augsburg, Germany
Email: {Bagci, Wolf, Satzger, Ungerer}@Informatik.Uni-Augsburg.DE

***Abstract*—Ubiquitous systems build on the vision that great amounts of fixed and mobile microchips and sensors will be integrated in everyday objects. Developing services on basis of sensor nodes and limited devices is an arduous task. A program running on such a device is static and limited to a single service. If a new service needs to be performed, devices have to be fundamentally reprogrammed and reloaded. For dynamic service distribution a mobile agent system is devised and services are distributed by mobile agents. The idea in this paper is to load a service on a limited device or sensor node when it is needed and to switch services dynamically. The service comes in form of an agent to each device. We developed a mobile agent system called UbiMASS that is downsized to run on sensor nodes. UbiMASS was evaluated on the sensor nodes ESB430 which has multiple sensors and actuators on-board. For real test scenarios, we could investigate the possibility and advantages of service changes on a wireless sensor network using mobile agents. UbiMASS offers an easy and convenient way to dynamically reprogram tiny devices with wireless connections.**

I. Introduction

It is a vision for a new computer era - *Ubiquitous Systems* are computing resources integrated into the human environment and not perceived as stand-alone computer systems. The aim of microchips is to support the users in an independent and invisible way instead of forcing them to adapt to the computerized world. Most of the devices will be attached to a fixed location whereas some will be mobile carried by persons. Service development on memory and energy limited devices is a complex task. An example for a service is a fire detection application on basis of temperature sensors that are spread out in an area. A program running on such a device is static and usually limited to a single task. If a new service needs to be performed each device has to be fundamentally reprogrammed. Regarding the large number of sensor nodes envisioned for future sensor network applications, this could lead to an intractable task. Running multiple services on one device is usually not possible because of memory and performance limitations.

Regarding the decentralized approach, the openness, and heterogeneity of wireless sensor networks, the paradigm of mobile agents presents itself for ubiquitous systems. The idea in this paper is to bring a service on limited devices and sensors when it is needed and to switch services dynamically. Imagine a smart office building that has multiple microcontrollers and sensors integrated in walls, doors and office equipments, and users who have portable devices as well as microchips integrated in badges. In the initial state, the microcontrollers perform no services, but have the possibility to host mobile agents. The service comes in form of an agent to each device and leaves if it is no longer needed.

Agents in general can be defined as software units with certain autonomy. They perform services by order of a user or other agents. Mobile agents have an additional property: they can autonomously migrate, i.e. they can transfer program code, data and continuation pointer to a remote computer and resume with the program execution. Beside this physical mobility, mobile agents have the possibility to communicate with each other in order to exchange information.

This paper describes a mobile agent system for ubiquitous environments called *UbiMASS*. The mobile agents in UbiMASS can receive information from the real environment through appropriate interfaces. The light-weight design of UbiMASS allows it to run on wireless sensor networks, that consist of several sensor nodes. Usually each node has its own processor, memory and application specific sensors. It is a characteristic of sensor nodes that all resources are extremely limited:

- **Energy:** Since the energy supply is provided by a battery or solar cell, it is an important requirement to reduce the energy consumption. Therefore, wireless communication and performance intensive tasks should be avoided.
- **Memory:** For current prototypes only a limited memory capacity is available. On the average sensor boards have less than 20 kilobytes of RAM and about 100 kilobytes of flash memory.
- **Performance:** In order to reduce the energy consumption, low performance processors are used on sensor boards. In most cases it is an 8-bit microcontroller. Therefore, the performance and speed is very limited.
- **Communication:** Wireless communication has naturally a weak data throughput. Additionally, problems with packet failures, packet loss, and collisions lead to increased energy consumption and time delays.

UbiMASS was successfully implemented and evaluated on ESB430 sensor boards that are developed at the Freie University of Berlin [2]. This board works with a TI MSP430, an ultra low power 16 bit RISC-based microcontroller with 60 KB flash ROM and 2 KB RAM. ESB430 has multiple sensors on-board. Figure 1 shows a picture of the ESB430 with its

978-0-7695-4049-8/10 $26.00 © 2010 IEEE
DOI 10.1109/SUTC.2010.38

sensors and actuators.

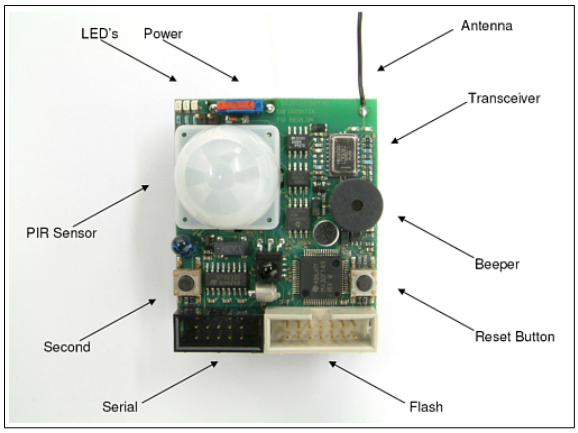

Fig. 1. The ESB430 sensor board

The next section describes related mobile code approaches for wireless sensor networks. Section III introduces the UbiMASS architecture and its components. UbiMASS is evaluated on the ESB430 sensor boards within several scenarios. Section IV describes the evaluation results. The paper ends with the conclusion.

II. Related Work

Although there was an initial excitement about the idea of mobile agents in the late nineties, there are only a few projects that exist concerning agent systems on wireless sensor networks. The most famous project in this area is *Agilla* [5] [6].

Agilla is a mobile agent middleware for wireless sensor networks. It offers a special extension for *TinyOS*, an open source operating system for embedded systems. The aim of Agilla is to create mobile agents and to allow them to spread out their code and current status over a wireless sensor network. Through this local control, the mobile agents obtain more flexibility to find an optimal position for application specific tasks. Based on the multitasking ability of TinyOS, Agilla allows to run several agents on a single sensor node. The number of possible agents differs depending on the local memory capacity. Each agent is autonomous but shares various resources of the middleware with other agents, i.e. a *neighbor list* and a *tuple space*. The neighbor list contains the addresses of neighboring nodes. Mobile agents use this list for planning their next migration. The tuple space offers a decoupled communication capability among agents. It forms a shared memory architecture, where addressing is performed over defined field names instead of memory addresses. A tuple describes a typified data object. All tuples exist even if the inserting agent does not exist any more. Another agent could later reuse the data in the remaining tuple. In this manner the tuple decouples the sending agent from the receiving agent. Neither knowledge about the location nor the existence of the communication partner is required. For the implementation of agents Agilla uses a stack-based assembler language. This approach is the main disadvantage of Agilla, since even the programming of simple applications is really hard and needs a lot of code lines. Complex applications are nearly impossible. First step to enhance this was to provide a higher programming language which is automatically transformed in Agilla code. But still only simple command sequences are possible. Another weakness of Agilla lies in the realization of the tuple space. In a wireless environment it is obvious that inconsistencies can arise. Due to connection failures data could not be reached or several tuples with the same value could exist. A *global tuple space* would be a solution but would contradict to energy and resource requirements of wireless sensor networks. Agilla was established and tested on the sensor boards *Mica2* and *MicaZ*, that have more extensive hardware resources than the sensor boards used in this work.

Another project that uses code mobility but not in the manner of mobile agents is *Contiki* [4] [3]. It is an open source operating system for an 8-bit controller. Contiki is implemented in the language C and is already ported to several microcontroller. It is not an agent system but it offers the possibility to dynamically re-programm sensor nodes during runtime using mobile code approach. The code itself cannot trigger the migration instead it is pushed by the underlying middleware. Contiki was designed for embedded systems with extremely low memory capacity. A typical Contiki configuration uses only 2 kilobytes of RAM and 40 kilobytes of ROM. The operating system consists of a simple event-triggered kernel, where applications can be loaded and unloaded dynamically at runtime. The processes of Contiki uses the so called *protothreads*, which provide optional preemptive multitasking. Inter-process communication is performed using message passing through events. Sensor nodes can communicate over a reduced TCP/IP stack called *uIP*. Contiki uses the standardized *ELF*-format [1] for the dynamic binding and loading of code. UbiMASS is also based on ELF which is described in the next section.

III. UbiMASS Architecture

Based on the requirements of ubiquitous agent systems, this section presents the ubiquitous mobile agent system for sensor networks (short UbiMASS). UbiMASS describes an agent middleware, which offers a fundamental basis, to develop a range of applications from the ubiquitous computing area.

The primary goal of UbiMASS is to offer mobile agents the opportunity to distribute on their own initiative the code, as well, existing data, and the current state over a wireless sensor network. The agents, using their own local control and flexibility, perform application specific tasks. It is not necessary to transfer data over an unreliable wireless connection because the agent itself has been moved to the data. This approach is an elegant and energy-efficient solution for distributed applications. Calculations proceed only at relevant locations sparing the entire communication network from useless traffic loads.

UbiMASS consists of multiple wireless connected agent hosts offering a platform for mobile agents. A UbiMASS host has a modular architecture with several components. Besides the management of the agent system, the hosts must run the agent and control its communication and migration.

Actually the agent middleware has three components: the ELF loader, the Migration Engine, and the Sensor-Actuator Interface. Figure 2 shows the UbiMASS host architecture.

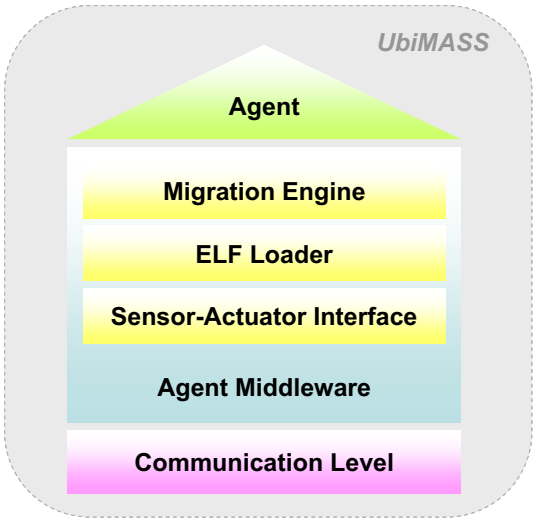

Fig. 2. UbiMASS host architecture

A. *UbiMASS ELF Loader*

The agent system UbiMASS uses the standardized format ELF for dynamic loading of the agent. In the case of an agent migration the middleware calls the loading method of the ELF loader to begin the execution of the agent. In order to explain this operation it is necessary to describe at first the basics of the ELF standard.

ELF stands for Executable and Linkable Format, that is basically a standard for executable files. It is mostly used in UNIX systems where it replaces the old and unflexible *a.out*. ELF aims to rearrange the machine code in order to load it fast and efficiently into the memory for execution. The standard describes how commands have to be organized, interpreted and loaded. The representation of control data is, in contrast to other proprietary formats, always platform independent. [1] distinguishes between three kinds of ELF files (object files):

- **Executable File:** This type of file comprises a program that is ready to be executed. All necessary information to create a new process is available. This process has access to the code and data within the corresponding file.
- **Relocatable File:** In this case the file does not contain an executable program, but only position independent code and data. These can be linked with other object files to produce an executable program or a dynamic library.
- **Shared Object File:** This file comprehends also code and data, that can be linked on two ways. On the one hand a new object file can be created using other relocatable or shared object files. On the other hand executable or shared object files can be used to produce a process image.

The object files are required for binding or linking of the program as well as for executing the program. Therefore, ELF defines different views of the same file: linking view and execution view (see Figure 3). The linking view constitutes the file as an alignment of *sections*. The execution view divides the file into *segments*. Each view has a table describing the several sections. Notice that sections and segments describe actually the same file only from different points of view.

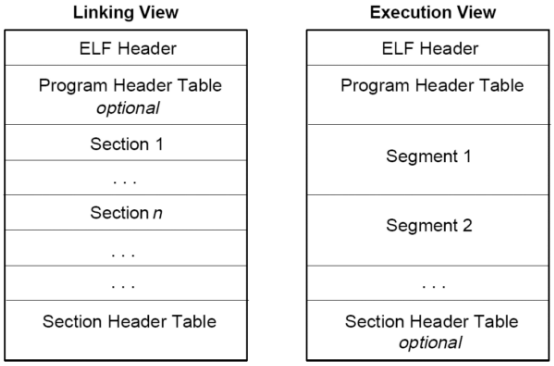

Fig. 3. Different views of an ELF file

The sections are used by compilers, assemblers, and linkers to arrange the file into several parts. In this manner the executable code is located in the section *.text*, all initialized variables in the section *.data*, and all not initialized variables in *.bss*. Additionally, there is a *string table* which contains all names of sections and symbols. Another essential section is the *symbol table*.

A symbol constitutes a kind of pointer with a name and a value. A name of a function or a variable can be associated with a physical address in this manner. The management of all symbols is done with the symbol table to allow an easy way of locating symbols. Besides index, names, and addresses of symbols the table also contains in the column *info* additional information about the visibility of the binding and the classification of symbols. The binding can be defined as *local, global* or *weak*. Local symbols are not visible outside of the object file similar to a local variable. Globally defined symbols are visible in all object files, i.e. they must be unique. Weak symbols actually behave like global ones, but have lower priority.

Actually an agent in UbiMASS is a relocatable ELF file. This file is transferred from one agent host to the next in case of a migration. In order to load the agent, the receiving agent host calls the function *elfloader_load()* which performs the following steps:

- At first the header is checked to ensure that it is a correct and compatible ELF file.
- The next section header is designed to get the number

and size of the according entries. Using the string table the names of all other sections is determined.

- Now all sections can be parsed. The section *.text* contains the current program code of the ELF file. The section *.data* presents all initialized data, whereas *.bss* contains the non initialized data. for the relocation is stored in both sections *.rela.text* and *.rela.data*. Furthermore the symbol table and the string table are used to set the pointer on all the relevant sections and to store the section number for dissolving the final addresses.
- Two functions which are ported for the architecture of the sensor board ESB 430 are used to allocate memory. Sections *.data* and *.bss* are stored in the RAM, whereas section *.text* gets into the flash memory.
- Using the symbol table next step is to relocate in sections *.text*, *.bss*, and *.data*. This leads to the integration of the location independent code into its environment where it can be executed later.
- Finally the completely linked code is written into the allocated space. The loaded programm can now be started.

This mechanism offers the possibility to integrate easily new agents into running UbiMASS agent system which ensures high flexibility and reliability.

B. Migration Engine

The Migration Engine of the UbiMASS agent middleware is responsible for complete and correct process of the agent migration. One of the key features is that the agent can decide on its own which current values of variables it needs to take to the destination host. In this manner, the agent can trigger an entry point in order to continue to work where it stopped at the previous host.

The process of the migration is shown in Figure 4.

1) Before starting the migration the agent can send current values of integer and string variables to the agent middleware using two functions *set_init_data(int id, int data)* and *set_init_data(int id, char data[])*. Because the agent itself will receive this list of tuples, there are no further specifications necessary. The agent knows how to handle the data on the destination node.
2) Using *start_migration()* the agent initiates migration and lets the middleware to perform the required steps.
3) Agent middleware initiates the migration. Actually, it is a weak migration, because only the program code and data of the agent is transferred, but not its status. The agent starts from the beginning at each time. The agent and its memory address that is known from the initial loading is forwarded to the communication level. On the destination host, the agent is loaded and started using the ELF loader described in the previous section.
4) In order to receive the last values of its variables the agent calls at the beginning the middleware function *demand_init_data()*. This turns the migration into a strong migration, because the agent can now continue to work with its previous status.
5) The destination host requests the list of variables by sending an *INIT* to the source host.
6) The source host packs all tuples into a char array and sends it to the requesting host. Figure 5 gives an example of the char array.

i	32	3	s	4	t	e	s	t	...

Fig. 5. Example: sending of variable values

At the beginning of each variable the type is specified by an *i* for integer and *s* for a string. Because integer variables require 16 bits two chars are used to deliver one integer right-shifting the first bits by 8. The value in the example can be calculated by $(3 << 8) + 32 = 768+32 = 800$. After the string type, length of the string is indicated. After successful transfer of the message, agent host can delete the agent process and deallocate memory space or agent can remain, if cloning of agents is desired.

7) After waiting for the variable value to receive the agent can access values using middleware functions *get_init_data(int id)* and *get_init_string(int id)*. Because the agent knows in which order it has put the variables, it also knows which variable matches its id.

The Migration Engine works as a fundamental component of the overall agent system. For this work our goal was to develop and to evaluate the basic functionality.

C. Sensor-Actuator Interface

Ubiquitous systems use context information to adapt to environmental changes. These changes can be detected using sensors. UbiMASS provides a sensor-actuator interface that is used by the agent system to access sensor and actuator information from the hardware. The access is strongly dependent on the used sensor board and its firmware. In the current version of UbiMASS the sensor-actuator interface is ported to the ESB430 sensor board that has multiple sensors and actuators on-board. Figure 6 gives an overview of current components of the UbiMASS interface.

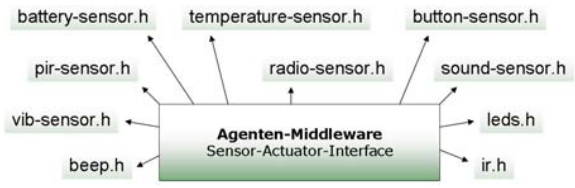

Fig. 6. Access of sensor and actuators of the ESB430 in UbiMASS

Porting to other devices is possible because of the modular concept of UbiMASS. It is only necessary to link the firmware components to the sensor-actuator interface of the new hardware.

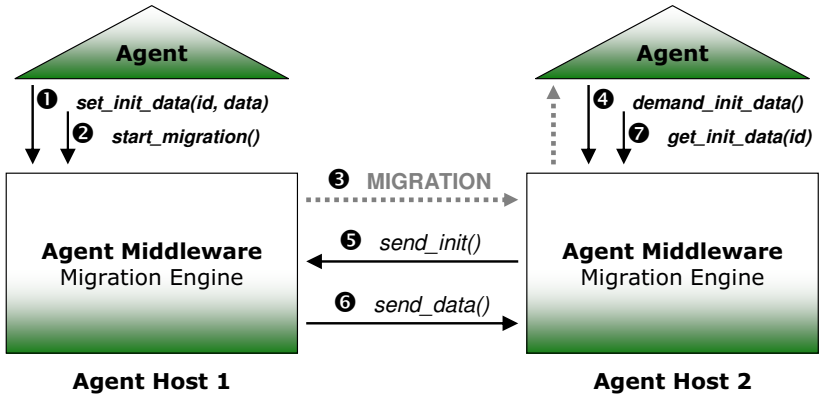

Fig. 4. Agent migration in UbiMASS

D. Communication Layer

The agent middleware in UbiMASS is positioned on top of the communication layer that is responsible for the transfer of messages between sensor nodes.

The communication bases on the *User Datagram Protocol (UDP)* a minimal and connectionless net-protocol. During the migration process an agent is fragmented into small packets. Each data packet has a size of 96 bytes and gets an additional header of 8 bytes. The UDP header contains four 16 bit fields as shown in Figure 7:

- *ID* field identifies the current agent. It is set randomly at the beginning and is incremented for each new agent.
- The *type* can get two possible values: a usual data packet has the type *TYPE_DATA*, whereas a response packet that indicates a packet loss has the type *TYPE_NACK*.
- Address field is used to reassemble the agent. The address of first packet gets the value of memory address the agent on the source was stored on. Each succeeding packet increases the value by the data length (96 Bytes). Destination host can use this information to reassemble packets in the right order.
- *Length* field shows the overall length of the sent data. This is used to check completeness of the entire data.

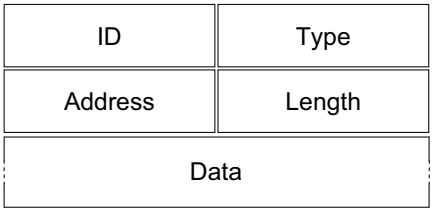

Fig. 7. UDP datagram format in UbiMASS

Since UDP is a connectionless and non-reliable transfer protocol, there are no guarantees that a sent packet reaches the receiver or that packets arrive in a sorted order. The communication layer of UbiMASS closes this gap by checking the address and ID fields in the packet header. If there are missing or incorrect packets, UbiMASS requests them by sending a *NACK* message that contains the address of incorrect packet. UbiMASS ensures in this way a reliable communication between sensor nodes.

E. UbiMASS Agent

The core of an agent system is the agent itself. Agents in UbiMASS can be easily implemented using the common C language. Figure 8 illustrates an example of a UbiMASS agent.

```
1  #include "ubimass-esb.h"
2  #include "migration/migration-tmp.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5  static struct etimer timer;
6  int i = 0;
7  int count_beeps = 2;
8
9  demand_init_data();
10 etimer_set(&timer, CLOCK_SECOND*4);
11 if (init_ok() == 1) {
12   count_beeps = get_init_data(0);
13 }
14 count_beeps++;
15 set_init_data(0, count_beeps);
16
17 while(i < count_beeps) {
18   etimer_reset(&timer);
19   beep_long(CLOCK_SECOND/4);
20   i++;
21 }
22
23 int len = migration_getLen();
24 start_migration(len);
25 }
```

Fig. 8. Example of a UbiMASS agent

This agent has the simple task of activating the beeper on the current sensor board. The count of beeps is increased with each

migration. Therefore, the agent uses variable `count_beeps`. After arriving on the current node agent has to request variable values from the previous node. This is shown on line 9 of the code, where the agent calls the middleware function `demand_init_data()`.

After a waiting time in which the agent host requests variable values from the source host, the agent can check the status by calling `init_ok()`. If variables are received correctly, the agent can access values over the function `get_init_data(id)`. After increasing the number of beeps, agent stores a new value by calling `set_init_data(id, value)` in order to be able to access the variable on the next node. The migration begins after calling the function `start_migration()`.

IV. Evaluation

We have evaluated UbiMASS in several scenarios using real sensor boards. In order to illustrate the advantage of UbiMASS compared to traditional over the air reprogramming, we chose a scenario where the sensor nodes were attached in the way that the entire network forms a ring. It is assumed that in the middle of the ring there is an object (e.g. a mountain) that blocks the wireless connectivity, so that each node has only a single predecessor and successor. Figure 9 describes the scenario.

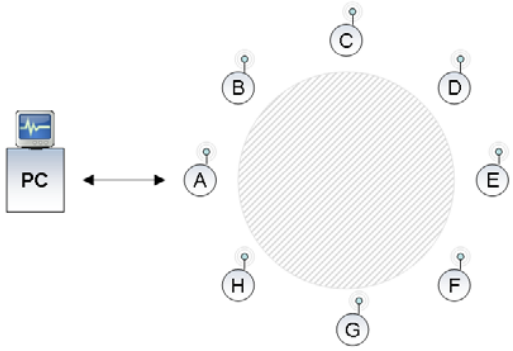

Fig. 9. Test scenario

In order to demonstrate reprogramming of the sensor network, the agent that has the simple task to count the number of sensor nodes is started first. The agent hops from node to node and increases the number of visited hosts. Since the network forms a ring, the agent will finally arrive at the first node where it displays the number of nodes on a connected PC. The round trip traverse of this agent is shown in Figure 10.

Since we want to demonstrate the service switch, we start a second agent that accesses the light sensor on each node (e.g. to check the weather situation). A great advantage of UbiMASS in this context is that the second agent can be started with a short delay after the first agent. In this way,

Fig. 10. Round trip of a mobile agent in UbiMASS

the agents provide different sensor information without losing much time. Figure 11 visualizes this specific scenario.

UbiMASS offers agents the possibility to work asynchronously and autonomously. Since the agent carries all the information and sensor values, there is no need for additional communication except for the migration. After migration of the agent, the sensor node can perform different services by allowing other agents to run on it.

It is interesting to compare this feature with other reprogramming systems. To investigate this, we implemented the same scenario with two successive agents as mentioned above on the open source operating system Contiki. With the specific application of *code propagation and storage* of Contiki it is possible to reprogram sensor nodes. After loading a program from a PC on a sensor node this program can be transferred to neighboring nodes by broadcasting the code. But this is limited to only one time, i.e. nodes that are reachable over two or more hops cannot be programmed with this approach. In our ring scenario, this leads to an enormous effort since we have to attach a PC at each second sensor node. After this work, it is necessary to send a round-trip message in order to receive the sensor values. Loading a second service requires the same effort again from the beginning. Figure 12 describes the required steps in Contiki compared to round trip of a single agent in UbiMASS (see Figure 10).

In order to evaluate the transfer time and loading time of agents, we measured the communication between two neighboring nodes. Transfer time is only the time to send the packets from one node to another without considering the dynamic loading time of the agent. Loading time contains also the time for starting the agent on the destination node. Figure 13 shows a chart where the size of the agent is increased and the respective time is measured in system ticks (since the timer functions of the firmware provide system ticks).

As expected the time increases linearly according to the size of the agent. Also the loading time is linear because the ELF loader has to handle more code if the agent size increases.

UbiMASS provides two different migration modes. Using weak migration the agent starts always from the beginning on the new node. All variables are initialized with pre-defined values. The strong migration offers the possibility to begin at the same point of code and the same variable values, where the agent stopped working. We measured a round trip time of agents with different sizes using weak and strong migration. Figure 14 illustrates the results. The strong migration has only a small offset compared to weak migration, which results from the additional transfer time of variable values.

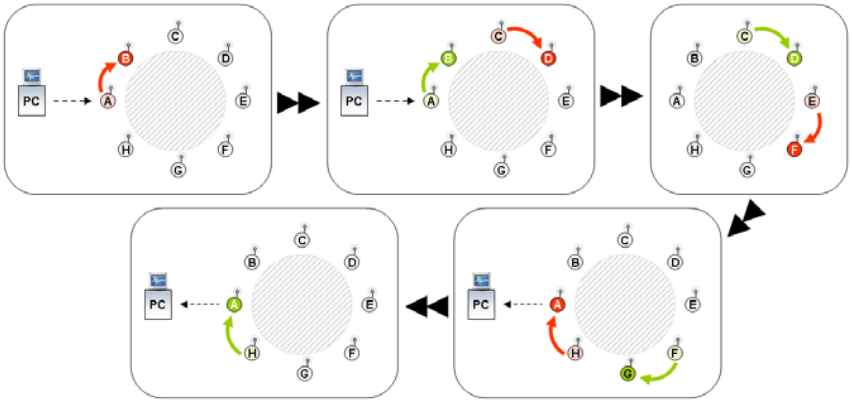

Fig. 11. Round trip of two successive mobile agents in UbiMASS

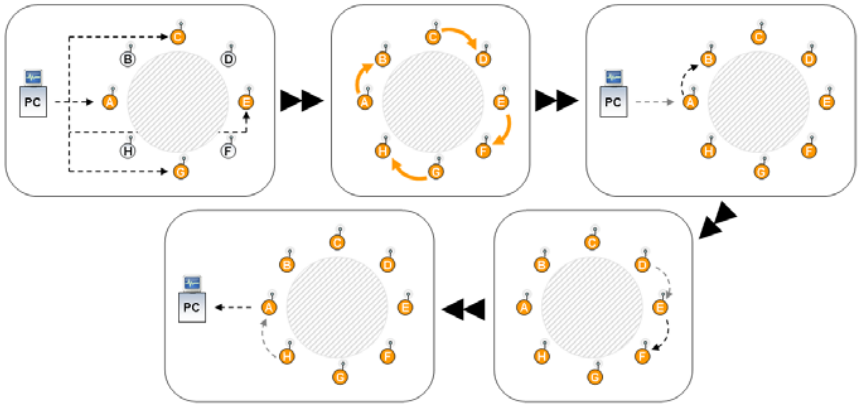

Fig. 12. Round trip in Contiki

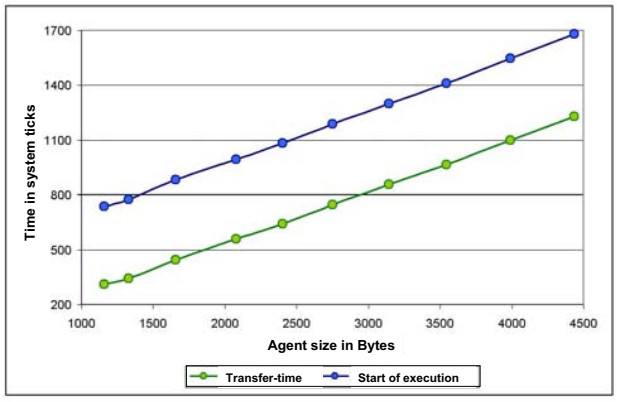

Fig. 13. Receiving an agent: only transfer time vs loading time

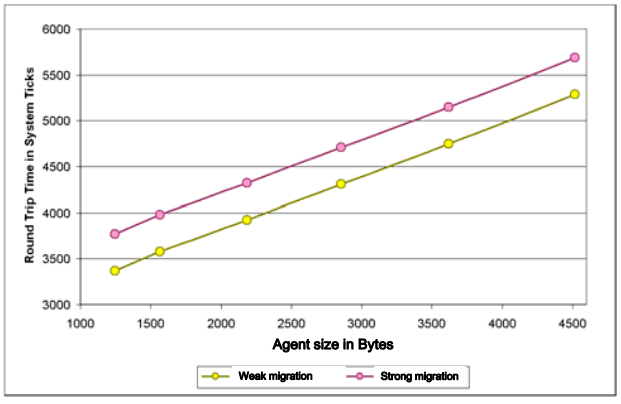

Fig. 14. Round trip time: weak migration vs strong migration

It is also interesting to compare the loading time of UbiMASS to on the air programming of Contiki. Figure 15 visualizes the results of this comparison. The chart clearly shows that the communication layer of UbiMASS works more efficiently. UbiMASS takes much less time to load the new service. Increasing the size of the code also increases the distance between both lines.

For the next scenario we implemented an agent that migrated to another node and back. We measured the round trip

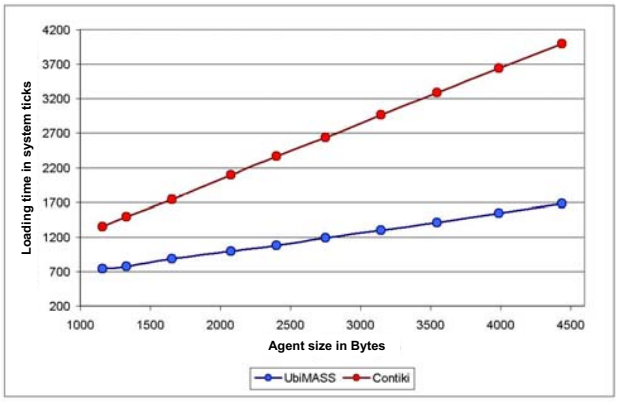

Fig. 15. Receiving an agent: comparison with reprogramming in Contiki

time in system ticks. It was also interesting to investigate the communication in relation to the distance between nodes. We performed several measurements by changing the location of the nodes. Figure 16 shows that increasing the distance leads to more packet losses which results in higher transfer times. Also obstacles like persons or walls and other interferences highly affect the communication time.

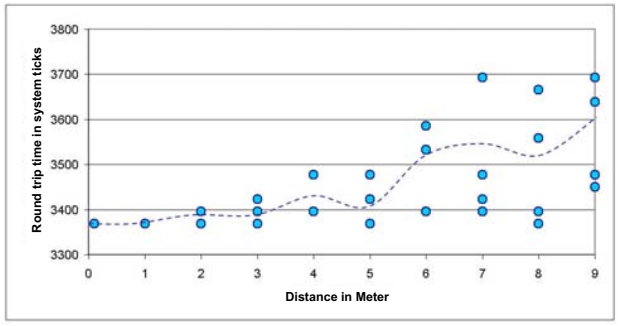

Fig. 16. Round trip time subject to distance between sensor nodes

V. Conclusion

This paper presented the *Ubiquitous Mobile Agent System for wireless sensor networks - UbiMASS*. UbiMASS is very light-weighted so that it can run on tiny devices like sensor boards. It is designed using a modular architecture in order to ease adaption for other hardware and to offer a convenient way for programming new services in the form of agents. Using the wireless link agents can migrate to remote nodes. The UbiMASS communication layer is responsible for the reliable transfer of agents. In order to load and bind agent code dynamically, UbiMASS provides the ELF loader that is based on a standard approach. The agent can access sensor and actuator information using interfaces of the underlying middleware. The migration engine offers agents the possibility to autonomously decide if they want to move to other nodes. We proved the usability of UbiMASS in scenarios with real sensor boards. Compared to other systems the reprogramming of nodes can be done in a more flexible and dynamic way using UbiMASS.

References

[1] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[2] ScatterWeb Homepage, 2007.

[3] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *Proceeding of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[4] Adam Dunkels, Bjoern Groenvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[5] C.-L. Fok, G.-C. Roman, and C. Lu. Mobile Agent Middleware for Sensor Networks: An Application Case Study. In *Proceedings of the 4th International Conference on Information Processing in Sensor Networks (IPSN'05)*, pages 382–387, April 2005.

[6] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A Mobile Agent Middleware for Sensor Networks. Technical Report WUCSE-2006-16, Department of Computer Science and Engineering, Washington University in St. Louis, 2006.