Resource-Aware Service Architecture for Mobile Services in Wireless Sensor Networks

Jan Blumenthal
Institute of Applied Microelectronics
and Computer Science
University of Rostock
jan.blumenthal@uni-rostock.de

Dirk Timmermann
Institute of Applied Microelectronics
and Computer Science
University of Rostock
dirk.timmermann@uni-rostock.de

Abstract— **After deployment of sensor nodes, the software of sensor nodes usually cannot be changed anymore. Thus, dynamical changing requests to the network and adaptations of the evaluation methods are not possible.**

We present a service architecture (RASA) for small, mobile, and resource-critical sensor networks. This architecture features software changes by injecting services at runtime. These services are executed on the sensor nodes. The design of RARA simplifies data aggregation and locally collaboration of sensor nodes. On this way, it is possible to extract implicit information of the network and to adapt the software to dynamically changing processes. Hence, RASA is able to react and to change the network behavior depending on the current conditions.

The proposed service architecture is optimized to execute simple, but local services. RASA meets the strong requirements of sensor networks regarding small resources, e.g. low memory consumption, supporting mobility, and robustness. Additionally, it supports the reusability of existing services or parts of services.

Index Terms— **Services, Service Architecture, Wireless Sensor Networks**

I. Introduction

Miniaturization technologies and advances in communication technologies lead to development of extreme small, cheap, and smart embedded devices, so-called sensor nodes. Hundreds or thousands of these sensor nodes build a sensor network [1]. These sensor nodes are deployed randomly in mostly impenetrable target terrains to measure a specific set of conditions.

Main task of sensor nodes is measuring the environment conditions with build-in sensors. One simple example is gathering the temperature around a node. If a given threshold is passed, an event should be triggered to inform other nodes or the base station. That means, sensor nodes simply measure and transmit data as assumed in most of the application scenarios. But in reality, this is inefficient due to missing data aggregation. In worst-case, hundrets or thousands of messages are transmitted and especially sensor nodes close to the base station would resign very quickly due to exhausted batteries caused by a huge number of transmissions. Nevertheless in case all messages were received by the base station, the evaluation of all messages would require very huge resources. As an example in a sensor network with *n* sensor nodes, exact positioning of nodes requires several *n***n* matrices. Moreover to detect a movement, the base station must be recalculate the position continiously based on incoming sensor node's data. Thus, transmitting all measurements to the base station is inappropriate. A local data aggregation between a group of nodes is necessary.

Data aggregation algorithms can be implemented directly into software of sensor nodes. This software is usually programmed into flash memory. But sometimes, environment conditions are different from the expected ones and flashed algorithms may be working wrong. In other cases, it is necessary to program additional evaluation algorithms to determine implicit data which was unknown before deployment. This can be unexpected velocity behavior, surprising termination of nodes, building of mixed groups, or any other phenomenon. To determine exactly what happens in the network, the software must be adjustable or changeable. Hence, all nodes require software updates to fulfil new requests. But a collection of all nodes to reprogram new adapted software is economically senseless. Thus, software on sensor nodes must be adaptable at runtime in the terrain. One possibility to support dynamical requests and data aggregation at runtime is based on mobile services.

In this paper, we present a new service architecture which considers special basic conditions in wireless sensor networks and supports highest flexibility within these close boundaries.

The remainder of this paper is organized as follows. Section II introduces services and considers the requirements of service architectures in mobile wireless sensor networks. Next, Section III surveys already related data collection protocols and service architectures supporting data aggregation and downloadable executable code. Further on in Section IV, we describe our service architecture followed by Section V. This section addresses the runtime behavior and lifetime aspects of a service. Finally, the paper ends with a conclusion.

II. Preliminaries

A service is a piece of software to reply remote requests, to interact between devices, and to hide the heterogeneity of a distributed system. Services are a growing

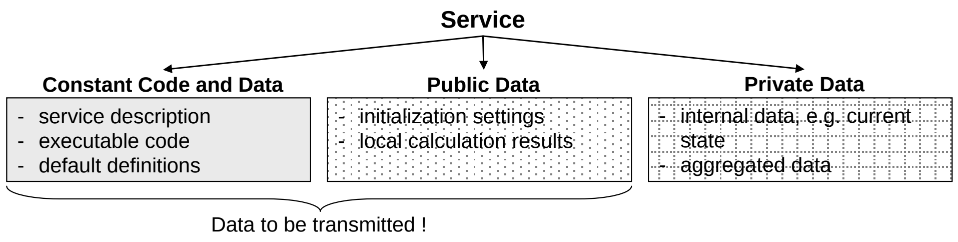

Figure 1. Structure of a service in RASA. It is divided into constant (executable) code/data, public data, and private data. Both, constant code and public data, are designed to be transmitted to neighboring nodes within a service message.

technology in mobile ad hoc networks. But in sensor networks, services hit upon several barriers caused by small resources, e.g. small batteries and energy consumption respectively, few program memory, very few data memory, highly specialized and tiny operating systems. Thus, saving messages to temporary buffers for retransmission or storing huge service data is nearly impossible. Further, development of one service architecture for sensor networks is restricted by different and complex memory architectures of microcontrollers used in the sensor nodes, lot of memory types with different addressing schemes, separation or combination of program and data memory, and argument delivery to subfunctions (stack based, memory based, or register based).

Based on our analysis, a service architecture for sensor networks must meet their special needs compared to resource-uncritical ad hoc networks, must be highly optimized regarding the strong requirements, and therefore fulfil the following criterions:

- Supporting new services at runtime
- Interaction and collaboration of services between sensor nodes
- Autonomously running services
- Extraction of implicit information
- Simple programming of services with large flexibility
- Fast and resource-aware seeding of services
- Low usage of data memory
- Few data transmissions
- Specialized to the specific memory architecture
- Reusage of existing program modules

III. Related Work

There are already some service architectures or data aggregation schemes for sensor networks proposed. All of them match most of the introduced criterions but not all.

MATÉ [2] is a byte-code interpreter for TinyOS [3]. It is a small communication-centric virtual machine designed as a component for the system architecture of TinyOS. The motivation to develop MATÉ was to solve novel problems in sensor network management and programming, in response to changing tasks, e.g. exchange of the data aggregation function. However, the associated inevitable reprogramming of hundreds or thousands of nodes is restricted to energy and especially storage resources of sensor nodes. Furthermore, the network is limited in bandwidth and network activity as a large energy draw. MATÉ attempts to solve these problems, by propagating small code capsules through the sensor network. The virtual machine of MATÉ provides the possibility to compose a wide range of sensor network applications by using a small set of primitives. In MATÉ, these primitives are one-byte instructions and they are stored into capsules of 24 instructions together with identifying and versioning information. In contrast to our design issues, MATÉ lacks the flexibility to use existing system components and it is limited by the available data memory to store the capsules and the runtime data. Further, it is difficult to use MATÉ in conjunction with complex data aggregation schemes or group building algorithms. Thus MATÉ is very useful for very small and easy applications.

TinyDB is a query processing system for extracting information from a network of TinyOS sensor nodes [4]. TinyDB provides a simple, SQL-like interface to specify the kind of data to be extracted from the network along with additional parameters, e.g. the data refresh rate. The primary goal of TinyDB is to prevent the user from writing embedded C programs for sensor nodes or composing capsules of instructions regarding to MATÉ. The TinyDB framework allows data-driven applications to be developed and deployed much more quickly as developing, compiling, and deploying a TinyOS application. Given a query specifying the data interests, TinyDB collects data from sensor nodes in the environment, filters and aggregates the data, and routes it to the user autonomously. The network topology in TinyDB is a routing tree. Query messages flood down the tree and data messages flow backup the tree participating in more complex data query processing algorithms. Thus, the flexibility of TinyDB is limited to the TinyDB implementation. But more important, TinyDB is simply a query processing system and not a service infrastructure to interact between neighboring sensor nodes. Similar approches are Cougar [5] and SINA [6].

IV. Resource-aware Service Architecture

In this paper, we present a service architecture (RASA) which meets the introduced requirements (changing runtime behavior, collaboration of nodes, extracting implicit data and others). This service architecture uses

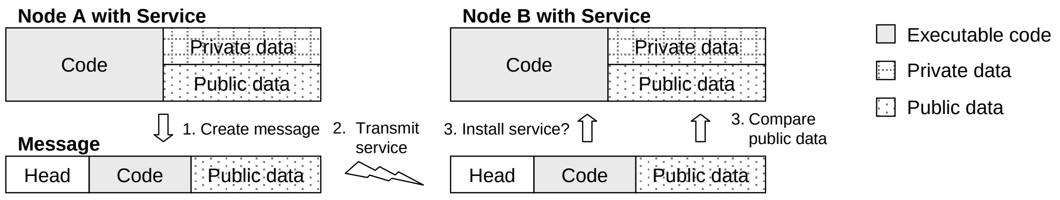

Figure 2. Creating a message (Step 1) out of service components and forwarding the service's messages containing public code and data from node A to node B (Step 2). Node B installs the executable code if necessary (Step 3) and executes the code to compare node's A public data with local public data (Step 4).

services which are injected to the network by a node (inquirer) at runtime. A service is forwarded to other nodes and installed as well as executed successive on all nodes of the network. After installation, services are accessible by other nodes and other services, too. Due to small available data memory at sensor nodes and the relatively high energy costs of data memory compared to flash memory, it is important to reduce consumption of data memory. Therefore, our objective is to store the code and most of data in flash memory.

A service is usually characterized by handling the heterogenity of a network and is executed on top of the middleware layer lying upon different operating systems. We tightened this definition to meet the special conditions regarding to the limited resources in wireless sensor networks. In our view, a special middleware layer on top of the operating system, as the well known CORBA [7], to support completely different devices is oversized and not applicable. In principe, a sensor network has only one main task, it should measure and aggregate data followed by a transmission. Therefore, a sensor network is build of homogeneous sensor nodes to reduce production costs, to ease programming of nodes as well as to reduce adaptation effort. There is no reason to deploy and support different types of sensor nodes. In our view, most node applications will use same or similar hardware and run applications adapted for the underlaying hardware. Thus, our middleware layer has reduced functionality. It contains basically a service dispatcher and hides the driver layer. The service dispatcher manages, connects, and runs services. The resulting middleware layer does not hide a heterogenic network, but therefore, it provides an interface to services which is significantly more relevant than hiding at all costs.

A. Services

Mobile agents are tiny programs which move through the network and are executed on the nodes. If necessary, these mobile agents are forwarded to neighboring nodes and executed, too. Our service architecture is based on the concept of mobile agents [8].

In our service architecture, a service is divided into three parts: constant code or data, private data, and public data (Figure 1). First, constant data contains executable code (mobile agent) either as native code or as virtual code designed for a virtual machine. Native code has, compared to virtual code, the advantage that the service is extremely fast and has a very high degree of freedom caused by direct hardware programming. Constant data are definitions and defaults to initialize the local instance of a service. Second, public data contains information representing the state or current results of the service instance. These data are public and therefore accessible by other nodes. In contrast, private data is not accessible by other nodes. It is used to store measurement values or internal variables.

Constant code and data as well as public data are transmitted by one service message from one node to another (Figure 2). Therefore, public data must be serializable. It must not contain data pointers or other references to local conditions. Additionally, public data is relocatable. After receiving a message, constant code is programmed into the non-volatile flash memory. The runtime system provides data memory required for private and public data. Received public data is used to optimize start conditions of the service and to aggregate data.

The selection which data is assigned to public or private data highly depends on the service. Usually, measured data is stored in private data memory and aggregated data is stored in public data memory to share these results with other nodes. Nevertheless, an important design criterion is the reduction of message's length.

B. Modules

According to our requirements, our service infrastructure is designed to reuse existing software components. In our context, these components are called modules. A module is a small piece of executable code with public as well as private data. Modules can be string together to fulfil a specific task. At least one module is required per service.

Figure 3 demonstrates a service built up of four modules (analysis, grouping, aggregation, and transmission). The modules are usually executed in sequence or in any order if specified. Each of these modules has its own private and public data area within the service's data space. Thus, if the service is transmitted to another node, the complete code of all modules and the public data block are transmitted as exemplified in Figure 2.

The service presented in Figure 3 denotes another nice feature of our service architecture – using internal modules. On this way, software components such as *grouping of nodes* may be outsourced, defined as internal modules and installed before deployment. These modules can be used by any installed service at runtime

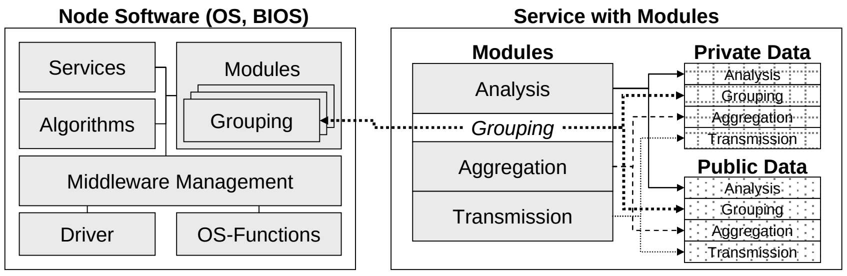

Figure 3. Interaction between original installed node software and flashed service with access to an internal module (Grouping). The grouping definition of the service is only a frame which references directly to the internal grouping module. Note, the internal grouping module uses service's data memory to store own data.

simultaneously due to using different parts of the data memory. Internal modules must not be transmitted anymore, because they are enclosed within the node software. Thus, the size of code within a service message reduces significantly.

C. Embedding a Service

Services are managed by a service dispatcher. To store received services, node software holds preallocated pieces of flash as well as data memory. Due to the separation of constant code/data and public data, this service architecture is able to run on systems with von-Neuman and Harvard configurations. Figure 4 shows the memory organization scheme for a Harvard architecture (8051 mircocontroller). The bold framed areas visualize the available memory preallocated by the node software to store services. If a service was received, the service dispatcher checks whether the service fits into the remaining memory space and installs the service, if possible. During installation, the service dispatcher sets the correct absolut memory pointer to access the correct data area.

Figure 4 displays further, how a service is organized within the memory pool. It starts with a service description (Head: Service) which defines required data space, number of modules, and a unique service identifer (ID) to distinguish different services. After the service definitions, each module is described. Here, relative data offsets point to the public and private data memory within the assigned data area. The module definition further contains relative offsets to module-specific functions within the constant code area to initialize, start, and end the service. After the definitions, application depending constants and executable code follow.

All defined offsets begin at the start of the service. Thus, the service dispatcher can easily calculate the physical addresses of the modules as well as functions. Internal modules are announced by a special flag which supplementary indicates that relative code offsets are added to the start address of the BIOS memory space. At this address, a system-wide jump table separates BIOS from services. On this way, different BIOS versions can support the same service. In most cases, internal modules require additional data memory. The size is determined at compile time and is taken into account to the data offsets for private and public memory areas. Thus, calling internal modules leads to an invocation of a BIOS function by using a part of the data memory assigned to the current running service.

D. Service Code Reduction

One of the challenging issues to create services is the reduction of the code size to prevent segmented message transmission. Figure 5 visualizes two steps to reduce the code size. First at the compilation level, the source of the modules is compiled to one monolithic block of code. The compiler removes unused or redundant components. It optimizes the interfaces between compiled modules and optimizes the data access. The application level symbolizes the code reduction caused by using internal modules.

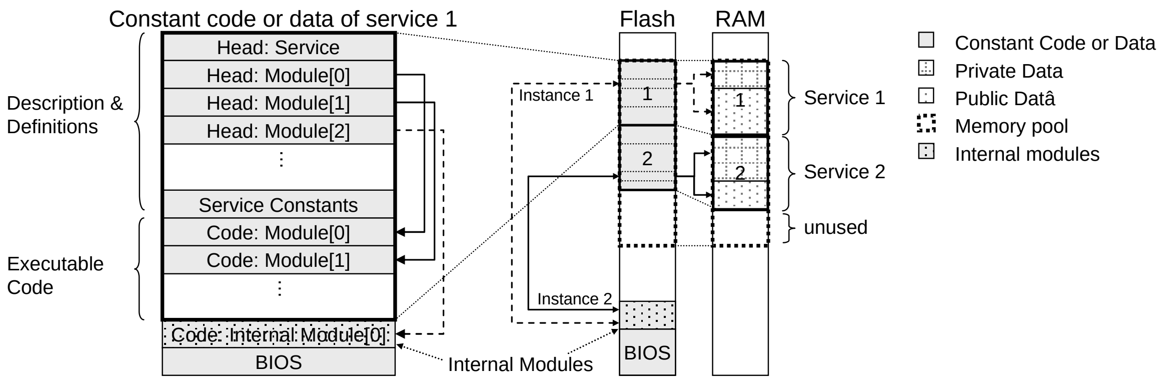

Figure 4. Memory organization scheme for a Harvard architecture (8051 mircocontroller) of two possible services (instance 1 and instance 2). Each service references to an internal module. To store runtime data, this internal module was assigned to a part of the service's data memory.

Services in RASA using native code are written in C language. At source code level, all modules are more or less platform independent and theoretically support heterogeneity. They are exchangeable and therefore reusable in other projects or services. But after compilation, all modules are merged together to a monolithic block. Thus at this level, they cannot be exchanged anymore.

Since the code size highly depends on the processor architecture, RASA supports virtual code, too. In this case, a specialized virtual machine as stated in [2] is required as additional component within the service dispatcher.

V. Running a Service

Before using a service, it must be compiled for the target hardware platform and the specific installed BIOS of the deployed sensor nodes. This is usually done at a PC which is connected to one sensor node acting as a gateway to the sensor network. This gateway is the *initiator* of the service. The identifier of the initiator, usually a MAC address, is added to the service together with the initiator's service count, the number of already injected services by the initiator, to distinguish different services and to prevent running services more than once on remote nodes.

After downloading the service from the PC to the gateway, the service is transmitted to its neighbors. All neighboring nodes receiving the service message behave as described in Figure 6. They firstly check, is this a service message and discard the message, if it is not. If the received message is an unknown service, the service is installed if enough memory is available. Next, the service is started. After finishing, the service's code and the results (public data) are transmitted. Therefore, yet another neighboring nodes receive the service and will install it. Thus, the service is broadcasted to all nodes in the network. Moreover, transmitting service's code and public data lead to a very robust behavior regarding service distribution. Especially in mobile sensor networks, sometimes unconnected sensor nodes exist due to separation. Presumed one uninfected node gets anytime a link to an already infected sensor node, this node will be infected, too. Thus in mobile sensor networks, all nodes are infected over time.

If a service was installed correctly, the service itself acts as a state machine managed by the service dispatcher (Figure 7). The service state machine starts with the state *Initialize Service* and automatically passes over to the state *Ready* after finishing. In *Ready*, the service waits for incoming events such as incoming service messages of neighboring nodes, new data at sensors, or timer expirations. Then, the service enters next state *Init Run Instance* to initialize one service call. In this state, the service call is instanciated, e.g. additional global data memory is acquired if needed. Further, several start conditions are defined to run through the modules, e.g. setting the module to executed first. If no error occurs, state *Run Module* is entered. In this state, the current module in the list of modules is executed (Figure 5). After execution, the module returns the number of the next module to be executed or EOM (End-Of-Modules). If another module is given, state Run Module is reentered and this module is executed. The looping continues until EOM is received. EOM signalizes the end of the service run. Thus, state *Finish Run Instance* is entered and all resources allocated during *Init Run Instance* are freed.

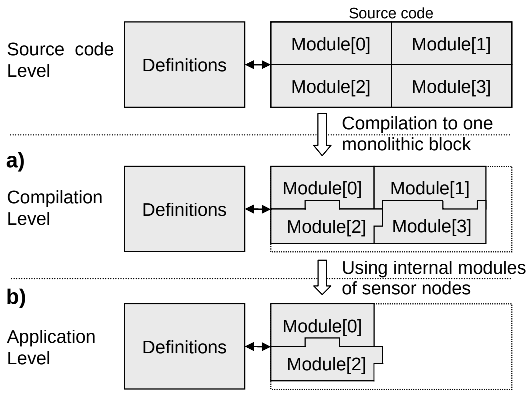

Figure 5. Code reduction levels to create small modules in services can be: a) the compiler system due to making one monolithic block, b) using internal modules to prevent unnecessary code transmission.

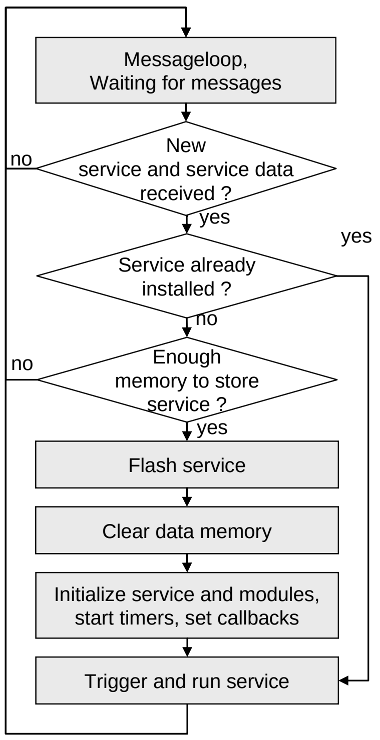
Messageloop,
Waiting for messages
New service and service data received ?
no
yes
Service already installed ?
yes
no
Enough memory to store service ?
no
yes
Flash service
Clear data memory
Initialize service and modules, start timers, set callbacks
Trigger and run service

Figure 6. Flow chart of a service treatment after receiving a service message

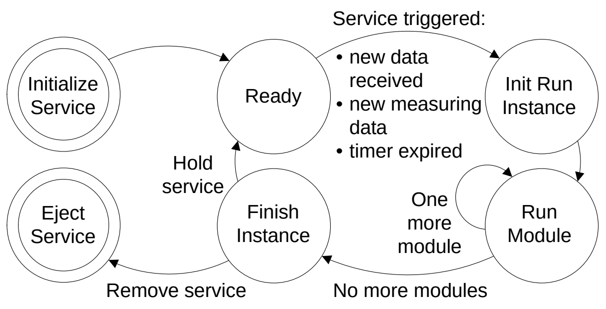

Figure 7. State machine of a service.

Using module numbers allows the building of a complex and dynamic flow charts with jumping between the modules depending on the runtime conditions. On this way, different modules can be connected without knowledge of each other. They only need the number of the next module which can be defined separately.

Sometimes, it is necessary to remove services from all sensor nodes. Then, a service message is injected by the initiator containing the service and special flags set to a removal command. This flag triggers a process to remove the service at each receiving sensor node. After removing the service, the message is broadcasted again.

During a removal process, it may sometimes be possible that some nodes do not receive the removal message due to leaving the network in the meanwhile. After rejoining the network, they start to distribute their still flashed services again. To prevent this unintentional behavior, there are at least two possibilities:

- A message containing a service number acting as threshold is transmitted. All services below this number are discarded. Thus, reinstallation of on old service is impossible. After receiving an old service from a rejoining node, the transmitting sensor node is invoked by an immediately sent reply to remove this service and to correct its own threshold number. This method is particularly applicable and resource-friendly.
- Each service contains a heartbeat number. After installing the service, the heartbeat is counted down. If it becomes zero, the service is removed. A heartbeat signal is transmitted periodically by the initiator. It resets the heartbeat number to the default value and prevents removal of the service. Thus, separated nodes remove its services by the time and reinstall services after rejoining the network. This method is not very resource-friendly due to the periodically transmission of heartbeats.

In our test example, a service measures a temperature, calculates the average and transmits the result to its neighbors. We implemented the proposed service architecture on the SeNeTs test and emulation environment [9] and on a Chipcon CC1010 microcontroller. Overall, the code size of this service at CC1010 including three modules is about 174 bytes.

VI. Conclusion

In this paper, we presented the novel service architecture (RASA) for mobile services in wireless sensor networks. These mobile services are software parts including executable code which are injected into the network without previous knowledge and are executed at every node. Moreover, a service consists of several modules to support reusability of already written components and to reduce service’s size by outsourcing often used components to the node software (BIOS).

This architecture meets the strong requirements applied to sensor networks such as updating the network with new services heeding small resources, data aggregation and extracting implicit information from the network. Due to its design, the service architecture is applicable in different hardware architectures (von Neuman, Harvard) and guarantees highest degree of freedom in programming services.

References

1. F. Akyildiz, S. Weilian, Y. Sankarasubramaniam, and C. Erdal, "A Survey on Sensor Networks" in *IEEE Communications Magazine*, pp. 102-114, vol. 40, 2002.
2. P. Levis, D. Culler, “MATÉ: a tiny virtual machine for sensor networks”, in *Proc. of ACM Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, December 2002.
3. J. Hill et al., System architecture directions for networked sensors, in *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
4. S. Madden, J. Hellerstein, and W. Hong, “TinyDB: in-network query processing in TinyOS”, Intel Research, IRB-TR-02-014, October 2002.
5. C. Jaikaeo, C. Srisathapornphat, and C.-C. Shen, "Querying and Tasking in Sensor Networks," SPIE's 14th Annual Int'l. Symp. Aerospace/Defense Sensing, Simulation, and Control, Orlando, FL, Apr. 2000
6. P. Bonnet, J. Gehrke, and P. Seshadri, “Querying the physical world”, IEEE Personal Communication, October 2000.
7. Object Management Group, “Common Object Request Broker Architecture (CORBA/IIOP)”, V3.03, March 2004.
8. D. Chess, C. Harrison, A. Kershenbaum, “Mobile Agents: Are they a good idea?”, IBM, New York, 1995.
9. J. Blumenthal, M. Handy, D. Timmermann: “SeNeTs - Test and Validation Environment for Applications in Large-Scale Wireless Sensor Networks”, 2nd IEEE International Conference on Industrial Informatics, INDIN 04, page 69-73, ISBN: 0-7803-8513-6, Berlin, Germany, 2004.