An integral approach to programming sensor networks

Remi Bosman, Johan Lukkien, Richard Verhoeven
Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands

***Index Terms*—Wireless Sensor Networks, Service Oriented computing, Event-based language, Active messages**

***Abstract*—In this paper we present a complete approach to programming sensor networks[1]. The main contribution concerns the separation of three abstraction levels, viz., application level, network level and node level, leaving more room for standardization than with current practices. In addition we propose to program the network from an overall perspective rather than programming individual nodes. The proposed model is event-based, corresponding closely to the nature of wireless sensors and admits content-based addressing of nodes and groups of nodes. The paper describes how the three abstraction levels come together and gives examples of the approach at all three levels.**

I. Introduction

The vision of wireless sensor networks is to deploy networks of cheap and "intelligent" sensors in order to gather information from an environment or to run highly decentralized applications. Such sensors are intended to be ubiquitous, low-cost and easy to deploy. A wide application range is foreseen where the focus lies on long-term monitoring, and operation under harsh or difficult environment conditions. In this way data can be gathered or monitoring solutions created that are not possible with current standard approaches.

Much more than in regular computer systems the value of a sensor network comes from the collaboration among the nodes. One might say that an application is, in fact, this collaboration. Also, this collaboration and possible tradeoffs therein are of dominant importance from a performance perspective, which is a crucial aspect in Wireless Sensor Networks.

It is quite common in the design of advanced software systems to balance performance and abstraction. When the abstraction is known and fixed an implementation is often internally optimized, leaving the abstraction to be merely conceptual. In addition, improvements can be achieved by using knowledge about a components' environment. In layered systems this is called *cross-layer optimization*. It tends to limit interoperability, especially when the used knowledge is also a requirement on the environment.

In this paper we concern ourselves with the programming of sensor networks. This has been a widely studied subject since the well-accepted abstractions of layered systems and modularization do not work very well for performance-constrained devices, leading to the mentioned shortcomings. Also, sensor networks are quite different from the regularly used distributed systems. In normal client-server systems or peer-to-peer systems, the involved machines typically have software installed and are started through a regular user interface. Our approach focussed on programming entire sensor networks. We present an integral view from application programming to network communication downto the execution environment on the nodes.

The paper is organized as follows. In section II related work is discussed. We designed a message format that is described in section III. Our programming model is described in section IV. In section V we describe our execution environment. Section VI explains our programming language. In section VII we summarize the results and look at future research.

II. Related work

A substantial amount of research has been done to increase the programmability of wireless sensor nodes.

Reprogrammability of nodes is vital to adapting to changing application requirements. Early approaches tried to have the requirement changes built-in through simple configuration or parameterization. This approach is obviously insufficient, since it is impossible to foresee all changes.

Solutions have been proposed to dynamically link binary modules [5] or upload virtual machine code [1], [6] onto nodes over the network. The latter don't require writable program memory and tend to be smaller which offsets the energy overhead of interpretation, especially if the network is frequently reprogrammed. Maté can "infect" nodes with program updates by spreading code capsules, but lacks flexible mechanisms to install code only on specific nodes or update arbitrary parts of deployed services. SwissQM focusses on a multiple-source to single-sink communication pattern. Our programming model does not have this restriction. Connections between nodes in our model (*subscriptions*) are many-to-many.

With regard to the distributed programming aspect of sensor networks, research is being performed to express the behaviour of an entire network in a single program, an approach which is called *macroprogramming*. TinyDB [2] offers a database view on an entire sensor network, such that information can be retrieved through an SQL like interface.

In Kairos [7] a high level program is used which contains constructs for i/o on variables at nodes, iterating through single-hop neighbours and addressing arbitrary nodes. This program is compiled to node specific binaries and then uploaded into the runtime on the nodes. The Pleiades [8] language also allows programmers to express network functionality as a central application which is analyzed and partitioned into node-level programs which are mapped onto the network.

Similarly, [9] provides a thin distributed layer such that the entire sensor network appears to applications as a single (Java) virtual machine. Their systems cuts a regular java bytecode program at the object boundaries and distributes the code amongst nodes in the network. This system lacks a mechanism which gives the programmer strong control where code is run, such as our content-based addresses.

[1]The research described in this paper was funded partially from the IST project WASP, IST.034963.

Finally, our message format is based on the same idea as found in SensorScheme [10]. A header which identifies some global function followed by data items which are passed as the arguments. These messages are self-contained and free of context such that they can be dispatched directly at the receiving node. This idea is similar to the *active messages* described in [11].

Many ideas in our programming model exist in one way or the other, however there are no systems that we know of that seamlessy integrate all of these providing built-in mechanisms for parameterization, partial or full code updates and replacements at targeted nodes and third party binding (many-to-many) of services deployed on nodes. All of which can be specified in a single macroprogram. Our goal is to find the correct abstraction for specification of network behaviour and optimal runtime performance.

III. WSP Message format

A. Basic concept

In a layered communication system the message exchanges typically realize information exchange between one or more sources and one or more sinks, with different layers in this communication visible as nested protocol handlers inside the messages. In a role-based architecture [3] we abandon this nesting; what is left is a collection of handlers to be executed upon receipt of the message. The handling is not necessarily layered. Also, new handlers can be added to messages upon forwarding.

A message can be of variable length and is self-contained. When a node receives a message, it is able to completely and immediately process it without requiring additional communication. Unknown handlers are simply ignored.

Handlers have parameters that are found following the handler identification. Handlers are identified through a tree-like data structure at the beginning of the message payload. Given two messages for a certain destination, the message format is such that they can be combined into a single new message containing the combined set of handlers.

Each message contains extra-functional fields to steer the processing of the messages within a node; together these are called the Quality of Service fields. These fields apply to all handlers addressed by the message. A priority field indicates this quality with respect to processing queues, transmission queues and storage. A deadline field indicates when a message becomes obsolete, such that a node can discard it. A reliability field indicates whether the application allows the message to be discarded in case of resource overload conditions. Together, these QoS fields allow for different message scheduling policies depending on the properties of the nodes and the network conditions. Advanced nodes can use a complex optimization algorithm to determine which messages are processed, taking all the fields into account. Simple nodes can use a simple algorithm, taking only a few fields into account.

Finally, each message contains an *application id*, which is a means to group handlers and generalizes the notion of a protocol version. A message also contains a check field that

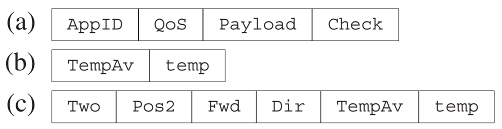

Fig. 1. The WSP message format. Fig. (a): the message headers. An ID to identify a group of handlers, Quality of service fields (see text) and a Check field for security purposes. Fig. (b): example payload. A call to the *TempAv* handler with one argument. Fig. (c): example payload. A composition of two handlers, one for forwarding and the payload of (b).

can be used for security purposes. Fig. 1 summarizes the message format, referred to as the WSP format.

B. Binding

The goal of our message format design is the intrinsic possibility of cross-layer optimization as well as the ability to "pay-as-you-go". The latter means that the functionality of the message passing is just as complicated as needed for each application. For example, an application that does not need any routing also does not need to spend any message space or handling to this issue.

Since the message format is expressive enough to define routing with it, the binding can be done to any protocol at any layer that allows the exchange of messages. Since every layer in a protocol stack typically adds a header to a message, introduces processing and hides details of the lower layer, the network binding should be to the lowest layer possible, as it provides the most freedom in optimization. However, binding to a lower layer might result in having to implement more functionality, like message routing and fragmentation. The bottom line is that when an existing layer is not performing optimally for the intended application, that is a valid reason to implement it with this format and correspondent handlers.

For wireless sensor networks, a common technology at the physical layer is IEEE802.15.4 [12], which among others provides message transport between nodes, encryption and link-level acknowledgments. Currently we bind directly to IEEE802.15.4, since standardized higher layers as zigbee [13] or 6LoWPAN [14] are not available yet.

IV. Services, events, handlers and subscriptions

A. Basic model

The second part of our design concerns the execution model. The WSP message format refers to handlers called upon receipt of a message. The question is how these handlers are defined, in what way messages are generated and how message sources and destinations are related. There is, of course, complete freedom to do this. However, our design uses the WSP format as effectively as possible.

We use only a few concepts in our execution model: *services* containing *event generators* and *handlers* (also called *actions*), and *subscriptions*. Nodes expose services on the network that represent a handler call interface and/or an event interface. By making a logical connection between a service event in one node and a service handler in another node, cross-node eventing is realized. Such a connection is called a *subscription*. This is the only interaction that we consider. An example of an event generator is a temperature reading; it can call a handler that records a temperature histogram. Another example is a temperature reading generated upon the temperature exceeding

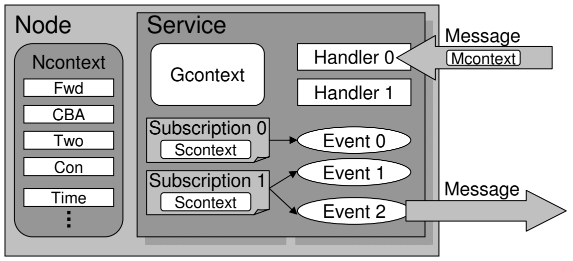

Fig. 2. The execution contexts. *Ncontext* is a node wide context consisting mainly of system calls. *Gcontext* consists of the global variables of a service (shared between all subscribers). *Scontext* contains subscriber specific variables as well as timing and QoS parameters. The body of event notification messages defines *Mcontext* (a handler ID and its arguments).

25 degrees and calling a handler to deal with this. The timing associated with these readings (e.g., the sampling time) is entirely determined by the destination of the subscription.

To look in a bit more detail, a service can be regarded as an object in an object oriented language. Its methods can be activated in one of two ways: by the receipt of a message and by expiration of a timer. Upon receipt of a message a handler is called; upon a timer expiration an event generator is called. In both cases, new messages may be generated for subscribers. Event generators are (condition, action) pairs; the action is executed when the condition evaluates to *true* and typically comprises message generation for subscribers.

The context in which the functions of a service are executed consists of four parts. First there is a list of globally available, node-wide, constants and variables. Second, we have local, internal variables of a service. Thirdly, the message that arrived contributes to the context of handlers and finally, any information given by a service subscriber is context for the event generators. These four contexts are called *Ncontext*, *Gcontext*, *Mcontext* and *Scontext* respectively. An example of an element of *Scontext* is a boundary value in an alarm, e.g. the value of 25 in case the subscriber wants to be notified upon the temperature exceeding 25 degrees. *Ncontext* contains (references to) system calls, functions that a service may call which are typically implemented by the OS (e.g. read out sensor hardware) or a supporting library (e.g. optimized functions for complex computations). It also contains the installed services and possibly some constants deemed useful upon deployment. An example of a system call is a function to inspect a temperature sensor or the energy level. An example of a constant is the node address or a node location if that is fixed.

B. *The system service*

In order to map the service model described above to the WSP message format we need several standard handlers and event generators with corresponding standardized input. These handlers are needed for the tasks of defining new services and defining subscriptions, more or less bootstrapping the system. Together these are called the *system service*.

For the mapping of this execution model to the message format we assume a closed naming scheme for all references, i.e., handlers, event generators and all other references are by numbering, similar as in machine coding. Purposely we avoid any lookup needs implied by open naming for the sake of efficiency. This implies that we assume that all services, handlers and subscriptions are generated from a single source or, at least, that somewhere a global lookup table for these references is maintained. This single source represents a "network program" which we call "the application". The application defines the Application ID in the WSP format and determines the services and subscriptions.

Content-based addressing

We use content-based addressing [15] in two ways: first in the placement of services on nodes and second in the subscription of handlers to event generators. A content-based address is a boolean expression defined on *Ncontext* combined with *Mcontext*. The number of different content-based addresses in a single application is limited. On a node we can therefore enumerate these expressions as we do with other references. Examples of content-based addresses are "node contains temperature reading function", "node energy level exceeds threshold" or "groupid equals specified value".

Content-based addresses (CBA) are handled by the CBA handler. This handler takes the expression (number) and its parameters and evaluates the result. If the result is *true*, the remainder of the message payload is interpreted as a regular payload.

The subscription handler

A node can send a subscription to a content-based address which is accepted by any node for which the content expression yields true. That node adds the subscription to the service referred to in the subscription. Other parts of the message specify sampling frequency, delivery deadlines and QoS parameters needed by the event generator upon creation of a message. In addition, any parameters for the event generators in the service are found in the subscription.

The configuration handler

The task of the configuration handler is to install new services and to define new content-based expressions. Its argument is a set of instructions. It is, in fact, a virtual machine with just 6 instructions:

- **DEFA** defines a content-based address ($id \times code$).
- **SERV** selects a service for configuration (id).
- **DEFG** declares a global service variable, i.e. in Gcontext ($id \times size$). Initializes the variable to zero.
- **DEFGI** declares and initializes a global service variable ($id \times size$, followed by *size* values for initialization).
- **DEFE** defines an event generator ($id \times code$).
- **DEFH** defines an event handler ($id \times code$).

The code argument in the `DEFA`, `DEFE` and `DEFH` instructions consists of a length specifier followed by the code itself. In our current implementation this code consists of bytecode (similar to [1], [4], [6]), but this can be extended to support binary code for a specific architecture. This combines perfectly with content-based addressing which can restrict loading of the code to certain machine types.

Other functions

There are examples in which no message forwarding is needed, e.g. when all nodes are one hop away from the infrastructure. However, if this is not the case, a basic means of reaching all nodes is needed, typically using flooding. It is therefore reasonable to require a flooding handler as a standard handler. Using this handler, it is possible to describe routing as part of the application.

In addition to this, it may be useful to discover the services in the network. Service discovery pertains to discovering *Ncontext*. A typical way to realize this is to install a specific event generator in the system service. Discovery proceeds by subscribing to this event generator.

V. The Virtual Machine

We have kept our virtual machine simple: it uses an execution stack for expression evaluation and regular flow control for conditionals and repetitions. The specific part, which makes it relevant to define a new virtual machine instead of using an existing one, concerns the use of the four contexts which act as four different memory spaces. Load and store instructions may operate on each of these contexts. In addition, there are three special instructions: the `CALL` instruction, to invoke a system call, the `FLOOD` instruction, to generate a message that subsequently is flooded into the network and the `NTFY` instruction, to generate a message and send it to subscribers. A listing of the instructions of the virtual machine can be found on our website [18]. Fig. 3 gives a simple example program. It is actually the translation corresponding to the program in Fig. 6.

```
TWO      9          # Two Handler
CON                 # Configuration Handler
DEFA     0     5    # CBA address 0
CALL     18         # NodeType (built-in function)
PUSHC    1          # ”Necknode”
EQ                  # NodeType()==”Necknode”
CBA      0     0    # Content Based Address Handler
CON                 # Configuration Handler
SERV     0          # HeadPosition service
DEFG     0     0        # hub
DEFGI    1     0    255 # strength
DEFG     2     0        # head_down
DEFE     0     24   # Event 0        || DEFH     30     21     # SetLoc.
CALL     28         # MotionY        || PUSHG    1             # strength
PUSHV    1          # $ytreshold     || PUSHA    125           # sig. str.
LESS                                 || LESS
PUSHG    2          # head_down      || PUSHG    0             # hub
NEQ                                  || PUSHA    0             # hubID
RJF      16         # if cond:       || EQ
PUSHG    2          # head_down      || OR
NOT                                  || RJF      8             # then:
STOREG   2          # head_down      || PUSHA    0             # hubID
PUSHV    0          # $Handler       || STOREG   0             # hub
CALL     25         # CowID          || PUSHA    125           # sig. str
PUSHG    2          # head_down      || STOREG   1             # strength
PUSHG    0          # hub            ||
NTFY                                 ||
```

Fig. 3. An example program for the virtual machine (in a single message). This is the compiled version of the HeadPosition service in Fig. 6. Handlers are in bold, configuration instructions in italic. Handlers and configuration instructions are explained in sections III-A and IV-B.

The virtual machine has a static memory footprint of approximately 1.3KB of RAM. Fig. 4 presents a breakdown and the ROM footprint on the left, and the dynamic memory usage per element on the right.

VI. The network programming language

One of the factors that contribute to the complexity of sensor programming is the extreme degree in which functionality is distributed over various entities within sensor networks. Typically one writes a number of small programs in isolation which should result in an overall network behavior (a *scenario*) when deployed on the nodes. This deployment often means physically accessing the sensor nodes. Our language is aimed at decreasing the gap between scenario and actual running code on individual nodes by programming at the level of the entire network, i.e. write one single program, compile it and deploy it to various nodes via the network.

	ROM	RAM
OS image	24084	1191
VM	1664	20
Sys calls	1530	146

	variable part (n)	footprint (bytes)
subscription	subscr. vars	$16 + 2n$
service	state vars	$14 + 2n$
CBA	bytecode	$4 + n$
event gen.	bytecode	$4 + n$
handler	bytecode	$6 + n$

Fig. 4. Static and dynamic memory usage of nodes.

The elements described in the previous sections come together in the network programming language. A program in this language consists of service definitions and subscriptions. A service consists of state, event generators and event handlers. Subscriptions are directed connections from event handlers of one service to event generators of another.

A service definition may refer to *system calls* or *system constants*. These are functions and values that can be accessed by an application running in a node. They typically consist of OS-calls like sensor access and global context. In fact, this system information refers to *Ncontext* in section IV-A. We assume here that a network-wide unique numbering is used. A table of these functions and constants, which we assume to be maintained separately, is an extra input to the compiler.

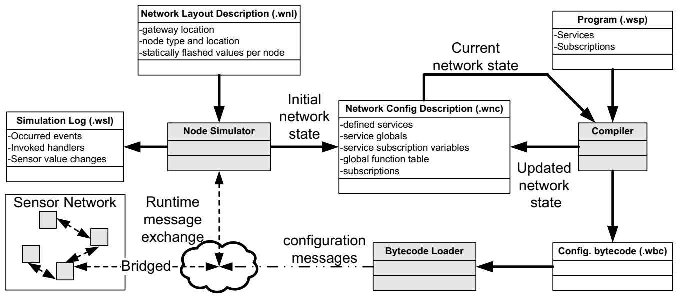

Fig. 5. The toolchain for compiling and loading programs. It consists of three programs, a compiler for generating configuration messages from the high level program, a loader for broadcasting these messages into the network and a simulator (or an actual sensor network) for testing and measurements.

Fig. 5 gives an overview about how the system is used. The compiler takes a program and a network configuration description as input (i.e. the table of installed functions, installed services and constants). Essentially, this description is used to intitialize the symbol table. The output consists of configuration messages with embedded bytecode of the form shown in Fig. 3 and an updated description file. The loader takes these configuration messages and broadcasts them into the network. Our simulator, which takes a scenario (network layout, sensor readings, etc..) as input, receives these messages and logs a run of the network into its log file. Alternatively, the configuration messages can be loaded into a sensor network running our interpreter on BSN nodes [17].

We demonstrate the language by means of a simple scenario; a more detailed description is presented on our site [18]. The scenario is part of a testbed being deployed in cooperation with the University of Wageningen (a dutch university for life sciences). In this scenario several cows are in a meadow and each of them is equipped with a sensor in its neck. The meadow itself contains nine stationary nodes (hubs) equally

```
service LocationService($Handler)                                                          1
  for [Network|*|NodeType()=="Hubnode"]
    on event localize when True do SendMessage([Network|1], $Handler, NodeID())            3

service HeadPosition($Handler, $ythreshold)                                                5
  for [Network|*|NodeType()=="NeckNode"]
    define                                                                                 7
      hub := 0
      strength := 255                                                                      9
      head_down := False
    on event change_head_position when (MotionY() < $ythreshold) != head_down do          11
      head_down := !head_down;
      SendToSubscribers($Handler, CowID(), head_down, hub)                                13

    action SetLocation(hubID) do                                                          15
      if (strength < self.signal_strength) || (hub == hubID) then
        hub := hubID; strength := self.signal_strength                                    17
      fi
                                                                                          19
subscription LocalizationSubscription
for [Network|*|NodeType()=="Gateway"]                                                     21
to LocationService($Handler=SetLocation)
on [Network|*|HasService(LocationService)]                                                23
with (period=3s, deadline=10s, send="normal", exec="normal")
```

Fig. 6. Example program which installs localization beacons on hub nodes and monitoring services on nodes attached to the neck of cows. For simplicity we assume that the cow is eating if it has its head down and standing otherwise. For brevity, processing the readings of the HeadPosition service by a gateway is not shown.

spaced in a grid which are used as static infrastructure. The goal is to collect information on the cows. The example described here, tracks when cows are eating (in a very simplified manner) and roughly where in the field they are located.

The program in Fig. 6 shows the code which implements the scenario. The program defines two services.

The first service configures all stationary nodes to be configured as beacons which periodically broadcast their ID. This is implemented in an event generator, which is an (named) event-condition-action rule [16]. It is always triggered by a timer, whose period is specified in the subscription. In line 3, a message containing a handler ID (specified by a subscription) and the beacon node ID is broadcast to all neighbour nodes.

The target nodes are specified by the content-based address $[Network|1]$. A content based address specifies a subgroup of the network , a hopcount and optionally a predicate. All nodes which satisfy the predicate (if any) receive the sent message. Addresses with predicates are shown in lines 2, 6, 21 and 23.

The second service monitors the position of the cow heads. It defines global service state (i.e. *Gcontext*) to track the closest hub and head position. Lines 11 to 13 show the event generator tracking the changes in head position. Unlike SendMessage, SendToSubscribers doesn't take a content-based address as argument, but rather fetches its destination from subscriptions (Scontext). The last part of the service defines a handler to update the nearest hub.

The program ends with a subscription. A subscription statement specifies a many-to-many subscription by providing content-based addresses for subscribers (*for*-clause) and providers (*to*-clause). Furthermore, the subscription provides the values for the subscriptionvariables (*$Handler*) and the sampling period and QoS parameters (*with*-clause).

VII. Conclusion

In this paper we presented an integral approach towards programming of sensor networks. Our approach is divided into three views on a sensor network: the application view, the network and the nodes.

At the application level a programmer writes a single event-based program to specify the behaviour of the network. Services (state, events, and handlers) are defined and installed onto nodes specified by content-based addresses. Connections between services are created by making subscriptions. These subscriptions parameterize the eventing behaviour (such as frequency, priority and boundary conditions) of providers.

The second view is the network. Here platform independency is achieved by standardizing on a small set of handlers and a message format akin to *active messages* containing a function ID and its arguments. The application program is compiled into a series of small configuration messages which are flooded through the network and conditionally installed onto nodes (based on content based addresses specified in the application program).

Finally, on the nodes itself, a simple virtual machine is run which abstracts away from the hardware specifics of the node. In principle any virtual machine can be used, provided they are augmented with the contexts and operations defined by the programming model.

Currently we have an operational implementation of our programming model consisting of the toolchain described in Fig. 5 an initial version of the virtual machines on the nodes and a simulator which runs virtual nodes on a pc. The virtual nodes can communicate transparently with the real nodes and are fully interchangeable which provides a testing environment for programs and protocols. Future work consists of optimizing and finalizing the bytecode and set of standard handlers, extending the expressivity of our language and performing case studies by implementing scenarios.

References

[1] P. Levis, D. Culler. Maté: a tiny virtual machine for sensor networks, Proc. of ASPLOS-X, 2002.
[2] S.R. Madden, M.J. Franklin, et al. TinyDB: an acquisitional query processing system for sensor networks, ACM Transactions on Database Systems, Vol.30, Issue 1, March 2005.
[3] Braden, R., Faber, T., and Handley, M. 2003. From protocol stack to protocol heap: role-based architecture. SIGCOMM Comput. Commun. Rev. 33, 1 (Jan. 2003), 17-22. DOI 10.1145/774763.774765
[4] P. Levis, D. Gay, and D. Culler, Active Sensor Networks. Proc. of NSDI, 2005.
[5] C. Han, R.K. Rengaswamy, et al. A Dynamic Operating System for Sensor Nodes, Proc. of MobiSys, 2005.
[6] R. Mller, G. Alonso, D. Kossmann, A Virtual Machine For Sensor Networks, Proc. of EuroSys 2007.
[7] R. Gummadi, O. Gnawali, R. Govindan. Macro-programming Wireless Sensor Networks using Kairos, Proc. of DCOSS, 2005.
[8] N. Kothari, R. Gummadi, et al. Reliable and Efficient Programming Abstractions for Wireless Sensor Networks, Proc. of PLDI, 2007.
[9] H. Liu, T. Roeder, et al. Design and Implementation of a Single System Image Operating System for Ad Hoc Networks, Proc. of MobiSys, 2005.
[10] L. Evers, P.J.M. Havinga, et al. SensorScheme: Supply chain management automation using Wireless Sensor Networks, Proc. of ETFA, 2007.
[11] T. von Eicken, D. Culler, et al. Active messages: a mechanism for integrated communication and computation, Proc. of ISCA, 1992.
[12] IEEE Computer Society, IEEE Std 802.15.4-2006: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs), 323 pages, September 2006.
[13] ZigBee Standards Organization, ZigBee Specification, Document 053474r13, 534 pages, December 2006
[14] G. Montenegro, N. Kushalnagar, J. Hui, D. Culler, Transmission of IPv6 Packets over IEEE 802.15.4 Networks, RFC4944, 29 pages, September 2007
[15] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, Jan. 2000.
[16] U. Dayal and J. Widom, Active database systems. ACM SIGMOD International Conference on Management of Data (tutorial), San Diego, California, June 1992.
[17] http://ubimon.doc.ic.ac.uk/bsn/m621.html
[18] http://www.win.tue.nl/san/wsp
[19] Prasad et al. SyD: A Middleware Testbed for Collaborative Applicationss over Small Heterogeneous Devices and Data Stores. Proc. of Middleware, 2004.
[20] The LiMo Foundation. The LiMo middleware. http://www.limofoundation.org/