

Advanced Middleware Support on Wireless Sensor Nodes

Daniel Barisic, Guido Stromberg
Infineon Technologies AG
81726 Munich, Germany
daniel.barisic,guido.stromberg@infineon.com

Michael Beigl
Distributed and Ubiquitous Systems Group,
Technical University Braunschweig
38106 Braunschweig, Germany
beigl@ibr.cs.tu-bs.de

ABSTRACT

State of the art solutions employ gateways for protocol translation between wireless sensor networks (WSNs) and IT systems, which results in significant management overhead for WSN deployments. In this paper, we investigate the feasibility of IT middleware support on the sensor node itself to eliminate the need for gateways. A major challenge in this respect is the storage of relatively large network messages inside a sensor node, which might easily exceed the available resources. This turns the efficient utilization of memory into the key requirement for middleware support on sensor nodes. In order to cope with this, we derive and analyze a generic layer model for protocol processing and inspect implications on the memory requirements for each layer. Cross layer optimization is employed to gradually develop an architecture in which application relevant information is directly extracted from network packets, which drastically minimizes the overall memory consumption. We finally present the first implementation of a Universal Plug and Play (UPnP) protocol stack for sensor nodes. Measurements confirm the feasibility of UPnP support for even highly restricted nodes and suggest that support for more complex protocols is possible.

Author Keywords

WSN, Middleware, SOA, UPnP

ACM Classification Keywords

D2.11 Software Architectures: Domain-specific architectures,
C2.4 Distributed Systems: Distributed applications

INTRODUCTION

Motivated by the vision of ubiquitous computing, research on wireless sensor networks (WSNs) aims at the development of small, intelligent, networked sensors and actuators. These are used to create added value in diverse application domains like home automation, habitat monitoring, military, industrial automation or safety. Advancements in this area have already brought forth solutions in form of hardware and software [3] that are mature enough to be adopted in real world applications [10]. Having mature technology as a

basis, the question on how to leverage the potential of WSNs becomes more and more important. Some researchers envision autonomous WSNs which use in-network processing to solve a certain task [7, 8]. For this purpose, new programming approaches and communication protocols have been developed. In other areas, like e.g. industrial automation, the WSNs are envisioned to extend the reach of IT systems into the physical world [4, 9], thus calling for the convergence of WSNs and IT. To this end, communication between WSNs and IT systems has to be facilitated.

In IT systems, the interoperability of basically arbitrary systems is desired. This motivated the vast adoption of the Service Oriented Architectures (SOA) paradigm and its implementation in form of the Web Services suite, which facilitates interaction on a high abstraction level. For the purpose of interoperability, programming and platform independent technologies like e.g. XML and HTTP are used. These technologies are usually considered to impose high requirements on computing power and memory. For WSNs, the communication mechanisms are dominated by the resource constraints (energy, computational capabilities, memory) of the nodes, so that typically efficiency is prioritized over interoperability.

State of the art solutions overcome this conflict by the use of so called gateways [6, 9]. The gateways, which are powerful nodes at the edge of a WSN, map the WSN traffic to IT compliant communication and thus preserve the freedom to use efficient communication means inside the WSN. However, the introduction of gateways leads to a dependency on an infrastructure for the use of the WSN. This restricts desirable features of WSNs like ad-hoc connectivity as a gateway needs to be installed before any interaction with the WSN. Moreover, the gateways themselves need to be maintained and therefore require additional management effort. Further, errors inside the gateway will make the WSN inaccessible. This turns the gateway into a single point of failure.

The next evolutionary step is the support of IT communication protocols directly on a sensor node, thus eliminating the need for gateways. A first step has already been made in this direction with the creation of the IETF working group on IPv6 over lower-power wireless network [2]. Support of the IP protocol allows WSNs to homogeneously blend into the IT landscape, as already shown by state-of-the-art sensor applications [4]. However, these approaches leave the support of higher level protocols (above IP) inside the sensor node an open research issue. While IP provides a general

basis for interaction, it focuses on the delivery of data rather than on semantic interaction. In contrast to that, IT middlewares define means to explicitly model and share the services provided by the participants. This allows the easy creation of complex, distributed applications without detailed knowledge about the underlying technologies. Due to the lack of middleware support on sensor nodes, these advantages are only available for WSNs to a limited extent.

In this paper we will present an approach to facilitate middleware support directly on sensor nodes to overcome these shortcomings. To this end, we investigate the bottlenecks of middleware support using the Universal Plug and Play (UPnP) as an example. Following architectural considerations, we will propose an optimization for protocol stacks on sensor nodes to reduce the consumption of memory. Finally, our results are used to build a prototype implementation of UPnP for the Sindrion [5] sensor node platform. Measurements show that UPnP support is feasible even on restricted sensor nodes.

UNIVERSAL PLUG AND PLAY

Universal Plug and Play (UPnP) is a widely used and commercially accepted middleware that allows devices to be described and controlled over the network. A peer-to-peer philosophy is inherent to UPnP, so that no central component is needed to facilitate interaction among the participants of a UPnP network. As we are interested in the realization of UPnP on a sensor node, we need to understand the requirements imposed by UPnP.

Functional Features and Protocols

The main features incorporated in UPnP [11] are *Addressing*, *Discovery*, *Description*, *Control* and *Eventing*. The hierarchical view of the protocol stack is shown in Fig. 1.

SSDP		SOAP	GENA
HTTPMU	HTTPU	HTTP	
UDP		TCP	
IP			

Figure 1. Layer Model for UPnP Protocol Suite

For *Addressing*, UPnP requires each participant to obtain an IP address via DHCP or Auto-IP. Support for this mechanism is trivial and can be expected to be provided by any IP enabled device. In order to facilitate *Discovery*, which is the dynamic look up of devices and services in the network, the Simple Service Discovery Protocol (SSDP) is used. SSDP messages are communicated via UDP unicast (HTTPU) or multicast (HTTPMU). The relevant information is encoded as plain text in specific HTTP header fields. Thus, SSDP requires a node to receive and transmit HTTP messages in the magnitude of 100 bytes as well as to conduct text parsing. Via *Description*, devices provide a description of their capabilities to the network. The description is given in XML format and transferred via HTTP. A node has to support the transmission of the descriptions, whose size is in the order of kilobytes. *Control* is the mechanism of invoking the

so called actions using the Simple Object Access Protocol (SOAP), which relies on XML message exchange via HTTP. A sensor node has to support reception and parsing of the SOAP/XML messages as well as the transmission of the responses. The size of all these messages are in the order of kilobytes. For *Eventing*, UPnP employs the General Event Notification Architecture (GENA), which specifies means for (un-)subscription of observers and asynchronous event notifications. Similar to SSDP, GENA uses specific HTTP headers to transport the information for (un-)subscriptions. The event notifications consist of an XML message that is transferred via HTTP. To this end, a node needs to support HTTP message parsing and the transmission of the event notifications with similar size as SOAP responses.

Summing up, in order to make WSNs natively UPnP compliant the sensor nodes have to support parsing of incoming HTTP headers (for SSDP, GENA) and XML messages (for SOAP). Further, the reception and transmission of messages in the order of kilobytes has to supported.

MIDDLEWARE STACK ARCHITECTURE

Let us now consider the implementation of the UPnP protocol suite on a sensor node. As UPnP uses layered protocols, the implementation will naturally follow these layers. In order to support generic statements on the issue of middleware support, we will not investigate the implementation of all UPnP protocols in particular but rather define a generic, coarse-grained layered stack architecture. We will use this architecture to identify and solve bottlenecks of middleware support on sensor nodes.

Layer Definition

- **Application layer:** This is the highest layer of our model and realizes the application logic. It uses the features of a middleware to implement its functionalities. For UPnP, the application layer contains the implementation of UPnP device and service logic using the features of UPnP, as described earlier.
- **Middleware layer:** The middleware layer provides the middleware features like *Discovery*, *Control*, *Eventing*, etc. to the application. To this end, it translates between the semantics of the middleware protocols and their representation in a message. For UPnP, handling of the protocols SSDP, SOAP and GENA is located in this layer. These protocols specify how the UPnP semantics for e.g. eventing subscription are encapsulated in messages, i.e. which HTTP headers carry relevant information and in which format.
- **Messaging layer:** The messaging layer is responsible for the delivery and reception of complete middleware messages, using the underlying transport protocols. For UPnP, the support of HTTP over TCP and UDP (HTTPU/MU) is provided in this layer.
- **Transport layer:** The transport layer is responsible for the end to end delivery of packets over the network. To this end, it covers issues like fragmentation, routing and physical transmission. With regards to the UPnP protocol, the

transport layer supports TCP, UDP and IP. Note that we ignore the lower layer communication layers in this study since we assume them to be given.

Information Flow

Let us now investigate the information flow through the generic middleware stack architecture, depicted in Fig. 2, in order to understand the implications of the architecture on memory consumption.

Incoming Message

The transport layer receives and handles incoming packets and therefore needs at least one buffer to store such a packet. The size of the buffer depends on the physical packet size, which is normally defined in the MAC protocol. For the IEEE 802.15.4 protocol, the maximum packet size is 127 bytes. The Sindrion MAC protocol supports 512 bytes per packet. The messaging layer composes the incoming packets to a message, which is then provided to the middleware layer. As discussed earlier, the message size can be relatively big (in the case of XML messages) and thus will require a bigger buffer than for the MAC layer. Further, it is possible to receive multiple messages at a given point in time. Therefore, parallel reception of messages has to be accounted for. This means that, in a straight-forward layer-based implementation, multiple message buffers need to be realized.

The middleware layer extracts the middleware relevant information from the messages. The information handled inside the middleware layer requires less storage size than in the message layer (discussed in the next Section). Finally, the application layer receives the information from the middleware layer and extracts the relevant information.

Outgoing Message

The transmission of an outgoing message is initialized by the application. To this end, information about the type of the message and possible parameters is prepared and passed to the middleware layer. The middleware layer encapsulates this information into the middleware compliant format (e.g. HTTP header fields and XML tags). The message layer creates the complete message as required for transportation over the network. Similar as above, it is possible that multiple messages are transferred at the same time, e.g. event notification are sent to multiple observers in parallel. Thus, a buffer for multiple messages is required. The transport layer handles the transmission of single packets and therefore requires a single packet buffer.

Memory Bottlenecks

Considering the memory requirements discussed above, we conclude that the queuing of incoming and outgoing messages consumes a considerable part of the overall memory. If we consider an exemplary UPnP stack, which e.g. should be capable of queuing five incoming and five outgoing messages with a maximum message size of 1,5 kilobyte, we need 15 kilobyte of RAM only to store the messages. This requirement can hardly be covered by existing sensor nodes, like the Mica2 (4 kB of RAM) or the Tmote Sky (10 kB of

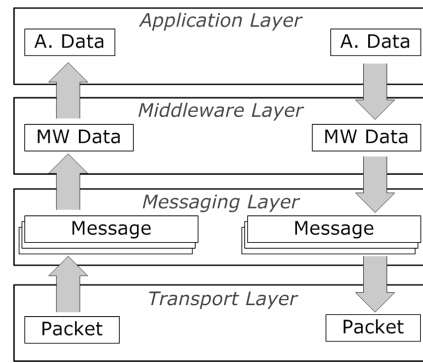


Figure 2. Four Layered Stack Architecture

RAM) [3]. Therefore, we need a way to minimize the memory consumption.

OPTIMIZATION CONSIDERATIONS

In the previous section we have inspected an architecture for layered protocol handling in which the storage of incoming and outgoing messages have been identified as a memory bottleneck. In order to cope with that, we will investigate a more compact storage of information.

Messaging Layer

The messaging layer usually requires large buffers to store the messages communicated over the network. As these messages serve the purpose to realize middleware features, we leverage middleware knowledge to optimize their storage. The middleware provides side information about message types, message structures, possible parameters, etc. which allows for a compressed storage of information. To this end, different compression approaches can be followed. For example, instead of storing the parameters of an UPnP action invocation as an XML structure (e.g. `<newStatus>0</newStatus>`), they can be stored as a key value pair of parameter name and parameter value (e.g. `newStatus,0`). Additionally, parameter values can be stored considering their data types, thus allowing for a more compact binary encoding instead of ASCII encoding like used inside a message. This shows that queuing on the middleware layer is more favorable than straight-forward message queuing.

In order to leverage this for our stack architecture, we merge the messaging and the middleware layer (see Fig. 3). Instead of assembling messages from incoming packets, we extract and store information relevant to the middleware on-the-fly. The same principle is applied to the outgoing communication, for which packets are created directly from the compressed data. The benefit of this approach is that the relatively large message buffers are replaced by smaller buffers containing only the compressed information. In order to realize this, we require a protocol handler that supports messaging and middleware protocols simultaneously. At the same time, we can also leverage the cross layer idea for protocol handling, as only the features of the messaging layer need to be supported that are actually required by the middleware. E.g. , in UPnP the HTTP support can be restricted

to support POST and GET messages as only these are used by SSDP, GENA and SOAP. In summary, this means that by merging the messaging and middleware layer, we minimize the memory consumption as well as the complexity of the stack. This brings us one step closer towards realizing middlewares on sensor nodes.

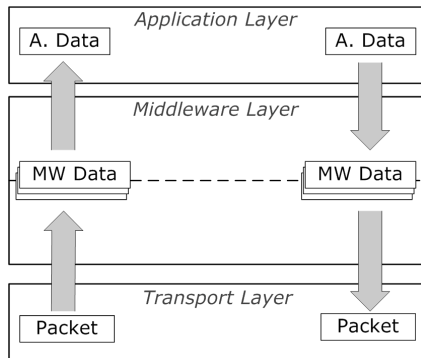


Figure 3. Three Layered Stack Architecture

Middleware Layer

In the previous discussion we have used middleware specific side-information to store a message in a more compact form. We now apply the same concept to reach even higher compression as we use application specific side-information. When we optimize the stack for a specific application, only information relevant for the application needs to be stored. This information is likely to require even less memory. While the middleware compression has to reflect the full feature set of the middleware, the application layer does not. For example, in UPnP and also other protocols (DPWS, Web services) resources are identified using URIs (Universal Resource Identifiers) or UUIDs (Universal Unique Identifiers). While these identifiers are suggested to be long in order to guarantee their global uniqueness, a practical application will only support a limited amount of resources. As a result, URIs and UUIDs can be represented in an application specific way using only a few bits rather than a few dozen ASCII characters. Further, application knowledge can also be used to store parameters of actions in a more efficient way. While the middleware layer uses information about data types (e.g. 'int') for compression, the application knowledge allows the consideration of value ranges (e.g. '0 - 9') which leads to higher compression.

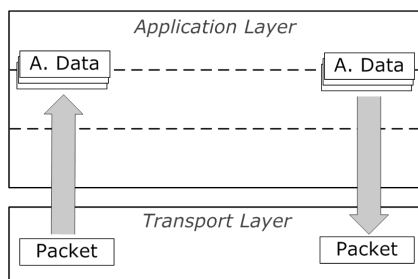


Figure 4. Two Layered Stack Architecture

For the stack architecture, this means that the extraction and storage of application information is beneficial for the memory consumption. To this end, messaging, middleware and application layer are merged and only application specific buffers are implemented (see Fig. 4). As a result, the protocol handling inside the stack considers application knowledge and is therefore optimized per application. However, the manual optimization of protocol handling is a tedious and error prone task. Thus, the automatic generation of a stack is recommended. To this end, a generator for UPnP stacks has been developed. It is presented in the next Section.

Transport Layer

Until now, we have subsequently minimized the memory consumption of the stack, leaving buffers only on the application layer and the transport layer. The transport layer contains a buffer for incoming and outgoing messages which has the size of a maximum MAC packet. As sensor nodes only provide half duplex communication, a single buffer should be sufficient. In order to realize a single buffer solution we merge application and transport layer, resulting in the architecture depicted in Fig. 5. Incoming packets are parsed directly on arrival in order to free the buffer as soon as possible. For the same reason, outgoing packets are written into the buffer only directly before transmission. The result is that even features like retransmission, used in reliable transport protocols like TCP, can be supported efficiently. In our approach, every transmitted packet is stored in memory in its compressed form. When a retransmission is required, the packet is decompressed into the buffer and transmitted. Therefore a packet only occupies the buffer when absolutely needed.

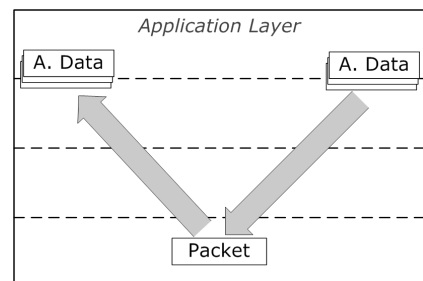


Figure 5. One Layered Stack Architecture

Summary

In this section we have discussed the optimization of the memory consumption of middleware stacks for sensor nodes in general. Application specific compression of data has been identified as the most efficient means to capture the information necessary to represent the global state of the stack. Therefore, aggressive cross layer optimization has been applied, which aims at the direct conversion of a packet into application information (and vice versa). In order to achieve this, an application specific protocol handler, that supports multiple protocols at the same time, is required. As a result, a stack has to be created per application and is therefore ideally generated.

In the following we will present an implementation of a UPnP stack that follows these design considerations. As we will see, our optimization approach results in a significantly reduced memory footprint. This finally allows the execution of the stack directly on a sensor node, thus underlining the benefit of our findings.

PROTOTYPE IMPLEMENTATION OF EMBEDDED UPNP STACK

As underlying basis for our investigations, we use the Sindrion prototyping platform [5]. The Sindrion node is composed of a 16 bit controller @ 11 MHz, 128 kB RAM and 2 MB ROM. Its RF unit allows communication in the 868 MHz band with approximately 50 kbps transmission rate. A proprietary operating system as well as IP protocol support are available. A so called network adapter allows access to the nodes on IP level from a PC, which is the basis for direct node to PC interaction.

The UPnP stack is written in plain C and it consists of a small static core which is extended by a set of generated, application specific modules. These modules are created via a Java based code generator that uses the UPnP device and service descriptions as input and creates the C modules and header files as an output. The developer customizes the generator by specifying, among other parameters, how many connections are served in parallel (queue size) and how many event subscribers are supported in parallel. The protocol handling inside the stack is build around a Boyer-Moore type of text searching algorithm for SOAP and HTTP parsing. The necessary preprocessing of application specific search strings is conducted by the code generator at compile time. For the outgoing messages, application specific message templates are created by the code generator. The templates are compressed using an ASCII encoded dictionary. This allows the decompression to be implemented using only little dynamic memory. When a packet is transmitted, the relevant part of the message is decompressed directly into the transmission buffer.

In order to evaluate the memory consumption, we have generated various stacks which differ in the supported UPnP services, message queue size and number of eventing subscribers. The presented measurements show the RAM consumption of the modules of the UPnP stack (not including the packet buffer, which imposes a fixed offset of one MTU size depending on the underlying communication protocol). The values have been directly extracted from the linker MAP file, thus representing actual and not hypothetical values.

Table 1 shows measurements for a simple service, where the message queue size and the number of eventing subscribers are varied. In a first step, the service consists of a single 'GetStatus' action that delivers a boolean value and does not require eventing. In its minimal form, the stack only requires 154 bytes. Even when we increase the number of messages that are served in parallel to five, the memory consumption only increases moderately to 230 bytes. In a second step, we have modified the simple service to support eventing of a boolean value. The 'GetStatus' action remains

the same. In its minimal form, the stack, including eventing functionalities, consumes 390 bytes. Although the absolute memory consumption is still quite low, we can see an increase of over 100% in comparison to the stack without eventing. This is partially due to the fact that eventing requires exchange and storage of a URL and hostname, to identify an event subscriber. These are lengthy strings and have no potential for application specific compression, as they can be defined freely at runtime by the subscriber. As a result, the stack requires 1106 bytes in order to implement the service with five parallel subscribers. In summary, the overall memory consumption of the stack is still low enough to fit into current sensor nodes (like Mica, Tmote Sky, Sindrion). However, we also see that eventing is a relatively costly feature with regard to memory consumption.

Scenario	A	B	C	D	E
#Messages	1	5	1	3	5
#Subscribers	-	-	1	3	5
RAM in bytes	154	230	390	726	1106

Table 1. Measurements for a simple Service

In a second step, we analyze the influence of the device and service complexity on the memory consumption. To this end, we have generated stacks for the standardized UPnP devices Binary Light and Dimmable Light, and a standardized temperature sensor service out of the HVAC specification. The corresponding measurements shown in Table 2 reveal that the complexity in terms of number of services and actions does not have a dominant influence on the memory consumption of the stack. Although the Dimmable Light device provides twice as many services and actions as the Binary Light, the memory consumption is merely the same. The difference of a few bytes originates from using an integer parameter instead of a boolean. Further, the HVAC device consumes more memory although hosting fewer services and actions. The reason for this is that a string based variable with arbitrary content needs to be supported to represent the application context of the service. This variable has a similar effect on the memory consumption as the URL used for eventing.

Device Type	Binary Light	Dimmable Light	HVAC
#Services	1	2	1
#Actions	3	6	3
	RAM in bytes		
#Msg. = 1 #Subscr. = 1	390	394	468
#Msg. = 5 #Subscr. = 5	1110	1122	1388

Table 2. Measurements for Standardized UPnP Devices/Services

In summary, we conclude that we have realized a UPnP stack with a low enough memory profile to be suitable for sensor nodes. Furthermore, we have seen that the UPnP device complexity does not have direct influence on the memory consumption. Therefore, an elaborate representation of a

senor node's functionalities in a UPnP network is possible without resulting in higher memory requirements. However, parameter types and values should be chosen carefully in order to allow efficient storage. For example, restricting a string parameter to an enumeration that only consists of a few distinct values instead of defining it as a generic string results in smaller memory requirements. In UPnP, this can be done by defining so called 'allowed value lists' for the parameters inside the service description.

RELATED WORK

Different approaches to support IT protocols on embedded systems can already be found in literature. The most prominent approach to facilitate Web services on embedded systems is the gSOAP toolkit [12]. Its unique feature is a binding between C/C++ code and SOAP elements, which allows implementation of Web Services on strongly typed C structures. To this end, the code for de-/serialization is created per application, using the WSDL description of the web service. As a result, programming of applications is simplified and the speed of parsing and generating messages is increased. However, the cross layer optimization idea, presented in this paper, is only partially reflected in gSOAP. Although application specific code is generated, it only accounts for the data type translation between C and SOAP. The core functionalities are wrapped inside a static module, which does not use application information for optimization. Further, gSOAP stores the complete incoming and outgoing messages in a buffer, which we have seen as a burden for the memory consumption. Intel [1] provides a toolkit for the development of UPnP applications. Code generation is used to create an application specific UPnP stack as well as skeleton methods, in which a developer can add the application logic. We witness a stronger employment of the cross layer optimization here, as even code for message parsing is specifically generated for the application. However, information about the application is not used to reduce the memory consumption. Similar to gSOAP, outgoing messages are generated via generic functions which create the complete message inside a buffer.

In summary, we see that the generation of application specific stacks is a common technique. The discussed toolkits however do not leverage the available information to the full extend to optimize the memory consumption.

CONCLUSION

In this paper we have discussed the challenges of IT middleware support on sensor nodes. We have seen that middleware support requires the handling of multiple, large messages in parallel, which easily requires more dynamic memory than available on a sensor node. To this end, the reduction of memory consumption of the middleware stacks is a crucial issue. Analyzing the layer model of a generic middleware stack, we have seen that higher layers tend to have more side information about messages and therefore allow for a more compact representation of the message content. In order to leverage this, we have proposed an architecture in which messages are not stored in their on-wire format but in a compressed, application specific form. In order to achieve the highest compression, the stack implements application

specific protocol handling. This calls for code generation techniques to create application specific middleware stacks. Finally, a prototype implementation proved that support of UPnP on sensor nodes is actually possible, which is a finding that exceeds the state-of-the-art. In its minimal form, the stack requires 154 bytes of dynamic memory, which is suitable for even highly restricted nodes.

REFERENCES

1. Intel Software for UPnP Technology. <http://www.intel.com/software/upnp/>, 2007.
2. Ipv6 over low power wpan (6lowpan). Technical report, The Internet Engineering Task Force, 2008.
3. J. Beutel. Metrics for sensor network platforms. In *Proc. ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 06)*, pages 26–30. ACM Press, New York, June 2006.
4. European 6th Framework Project. PROMISE - Product Lifecycle Management and Information Tracking using Smart Embedded Systems. <http://www.promise.no/>, 2005-2008.
5. Y. Gsottberger, X. Shi, G. Stromberg, T. F. Sturm, and W. Weber. Embedding Low-Cost Wireless Sensors into Universal Plug and Play Environments. In *EWSN*, January 2004.
6. M. Isomura, T. Riedel, C. Decker, M. Beigl, and H. Horiuchi. Sharing sensor networks. In *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, page 61, Washington, DC, USA.
7. P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
8. S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
9. M. Marin-Perianu. Decentralized enterprise systems: a multiplatform wireless sensor network approach. *IEEE Wireless Communications Magazine*, 14(6):57–66, 2007.
10. T. Riedel, C. Decker, P. Scholl, A. Krohn, and M. Beigl. Architecture for collaborative business items. In *ARCS*, volume 4415 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2007.
11. UPnP Forum. Universal Plug and Play Device Architecture 1.0. <http://www.upnp.org>, June 2000.
12. R. van Engelen and K. Gallivan. The gsoap toolkit for web services and peer-to-peer computing networks, 2002.