Towards an Adaptive Execution of Applications in Heterogeneous Embedded Networks

Andreas Scholz, Stephan Sommer, Alfons Kemper, Alois Knoll
TU München, Institute of Informatics
Boltzmannstr. 3, D-85748 Garching, Germany
scholza,sommerst,kemper,knoll@in.tum.de

Christian Buckl, Gerd Kainz
fortiss GmbH
Guerickestr. 25, D-80805 München, Germany
christian.buckl@fortiss.org, kainz@fortiss.org

Jörg Heuer, Anton Schmitt
Siemens Corporate Technology, Embedded Networks and Service Infrastructure
D-81730 München, Germany
joerg.heuer@siemens.com, anton.schmitt@siemens.com

ABSTRACT

Embedded networks are emerging in many application fields, such as the automotive or building and factory automation sector. Compared to other distributed systems, embedded networks offer a new challenge for developers: heterogeneity and resource constraints. The nodes contained in these networks can differ greatly w.r.t. their storage, processing and sensing/acting capabilities, ranging from very simple sensor devices with very limited resources over programmable logic controllers to very powerful nodes such as PCs. In order to achieve an efficient execution of applications running on such a network, a middleware is required that automatically adapts the embedded network to the requirements of the installed applications. In this paper, we will present a model driven development approach that allows the specification of application requirements, and a corresponding middleware solution that supports the automatic adaptation of the application execution based on these requirements and the characteristics of the underlying hardware.

1. INTRODUCTION

Embedded networks containing a multitude of networked nodes with varying sensing, acting, and processing capabilities are gaining increasing importance in many application areas such as the automotive, building management, or factory automation sector. Besides the challenges concerning the development of suitable hardware devices and communication infrastructures, the application development is increasingly difficult. The special characteristics of embedded networks, such as resource limitations, heterogeneous hardware, ranging from PCs over embedded controllers to primitive devices like switches, and the use of diverse communication protocols pose new and unique challenges.

Service Oriented Architectures are a promising approach to overcome these diffculties and are applied in several research projects. The decomposition of monolithic control applications into smaller interoperating services has several benefits. First, the higher level of abstraction allows to safely hide implementation details from the application developer and fosters the development of solutions that allow the end-user to install and configure applications in an embedded network. This is decisive in areas such as the building and home automation sector. Second, the introduction of well defined and re-usable interfaces eases the integration of components stemming form different vendors. Third, SOAs inherently support the distributed execution of applications, because the individual services which are composed to larger applications can be distributed throughout the network. This distribution of components plays a central role for the optimization of the application execution in embedded networks, especially if the networks are heterogeneous. The exact optimization goals may vary depending on the applications scenario, e.g., one goal could be to optimize the overall lifetime of a battery powered network, another goal could be to avoid network congestions by homogenizing the utilization of network connections. An additional complexity is introduced by network dynamics: new nodes may enter the network, existing nodes may fail and network characteristics can change over time, especially if wireless communication media are used.

The complexity of large embedded networks and the dynamics mentioned in the previous section require mechanisms that allow an embedded network to autonomously adapt the execution of applications, because manual optimization by a trained expert is too expensive and too time-

SESENA'10, May 3, 2010, Cape Town, South Africa.

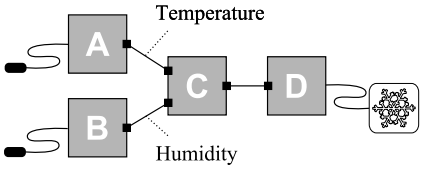

Figure 1: Example Application: Air Condition

consuming. In this paper we show how optimization goals can be specified at the application level and how these goals can be realized by a middleware that optimizes the placement of services based on these goals and the characteristics of the underlying hardware. The detailed contributions of this paper are: (1) the introduction of a system architecture that supports the adaptation of the application execution w.r.t. the underlying hardware, (2) the definition of a set of metrics that allow quantifying the quality of a placement, and (3) the development of optimization algorithms that allow deriving good placements based on these metrics.

The rest of this paper is structured as follows: In Section 2, we provide a short overview over the ϵSOA project[12], our solution for building service oriented embedded network applications. We introduce metrics for quantifying the quality of service placements in Section 3 and describe algorithms that calculate a service placement based on these metrics in Section 4. In Section 5, we provide a short overview of our demonstrator: a heterogeneous embedded network for energy management in smart buildings. We conclude the paper with an overview of related work in Section 6 and a summary and a presentation of ongoing work in Section 7.

2. SERVICE ORIENTED SYSTEM ARCHITECTURE

In this paper we aim at embedded networks that are using a service oriented paradigm. In such a system, all functionality in the network is encapsulated in services, which can either represent hardware devices such as sensors or actuators, or software components that contain the application logic. Each service possesses a metadata description which defines its in- and output ports. The communication between services is done by connecting an output of one service to a compatible input of another service. Ports are compatible, if they expect data of the same type. The services themselves are data driven, i.e., operate only on the data they receive and have no knowledge about the sources their data stems from and the targets their outputs are forwarded to. This design greatly enhances the reusability of the services and allows to move services between nodes without worrying about hard-coded addresses in the service implementations[1]. Figure 1 shows an example application comprising four services. Service A represents a temperature sensor, service B a humidity sensor. Each of these services possesses a single output that produces measurements with a configurable data rate. The data produced by these two services is consumed by a logic service C, which possesses two corresponding inputs. The metadata descriptions of these inputs ensure that the correct data is supplied by requiring

[1]This design is well known from Web services where there is also a clear separation between the services themselves and the composition language, e.g., BPEL.

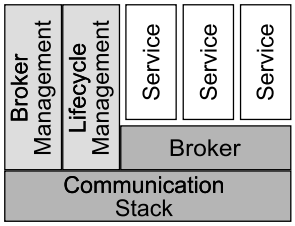

Figure 2: Node Architecture

data of type "temperature"[2] for the upper input and data of type "humidity" for the lower input. The output of service C is connected to the input of service D, which controls an air condition based on the commands received at its input. The logic service contains the "intelligence" of the application and decides when to turn on or off the air condition, based on the humidity and the temperature of the room.

2.1 Node Architecture

Figure 2 shows the architecture of a node in the ϵSOA platform, our solution for building service oriented embedded networks (this is a typical architecture for service oriented systems and can also be found in other projects). Every node possesses a communication stack that allows the node to exchange messages with other nodes in the network. We will assume that this stack handles communication across different communication media and network boundaries transparently, a possible stack that contains this functionality is presented in [13]. Based on this stack, a message Broker component handles the delivery of messages between services. Incoming messages are dispatched to the targeted instances and outgoing messages are forwarded to the corresponding remote hosts. The Broker contains a routing table that specifies these actions and that can be modified by a special component of the middleware, the *Broker Management*. It allows inspecting, adding, changing and removing entries in the Broker table. The services executed on a node are controlled by the *Lifecycle Management* component. It allows the installation of new services[3], the starting and stopping of running services, and the de-installation of services.

2.2 Model Driven Development

In order to support the automatic adaptation of applications to the characteristics of a given network, we developed the ϵSOA platform which is described in more detail in [12]. The ϵSOA platform uses a model driven development ap-

[2]These types can be taken from a domain specific taxonomy.

[3]The installation of a new service requires the dynamic loading of service implementations which are shipped over the network. Some operating systems support this functionality out of the box, e.g., Contiki[5], other operating systems, such as the popular TinyOS[14] do not. The authors of [6] show a concept that allows to add this functionality to TinyOS and similar operating systems. In this paper, we assume that a dynamic code loading mechanism is available on the nodes. If such a mechanism cannot be implemented, a possible workaround is to install a software library with commonly used services on each node. The optimizations presented in this paper can still be applied to such a system by modeling the availability of services on the different nodes via the constraint system explained in the following sections.

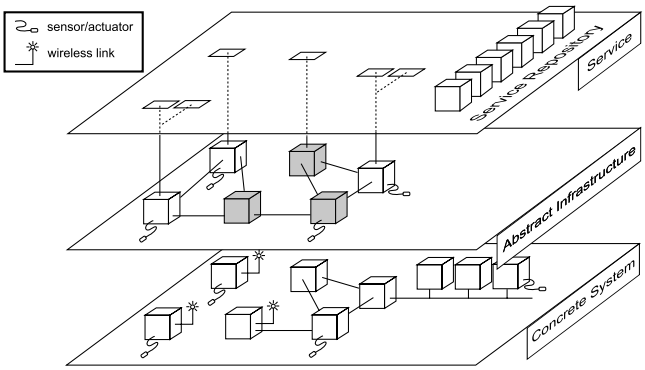

Figure 3: Abstraction Layers Used For Modelling Embedded Networks

proach. It is based on a model of the hardware available in a given network and a model of the applications and their functional and non-functional requirements. Examples of such requirements are memory and CPU demands, network bandwidth requirements, etc. These two models are combined by a global optimizer, which configures and programs the embedded network in a way that ensures an efficient execution of all running applications - or issues a warning that the available hardware is not capable of running the desired applications.

The model is based on abstraction layers, which are shown in Figure 3. Based on a given *Concrete System*, an *Abstract Infrastructure* model is built. Every node from the concrete system is represented as a node in the Abstract Infrastructure. The special characteristics of a node, such as the available memory, the CPU power, energy resources, etc, are annotated as properties at the node. Analogously, all communication links between devices are represented by links in the Abstract Infrastructure, which are annotated with characteristics such as bandwidth, reliability, etc. Note that these links represent single-hop unicast communication capabilities. If a wireless communication medium is used, a link is added to all nodes that are in communication range of a specific node. Analogously, a link is added between all nodes that communicate via a shared bus channel, etc. What characteristics are available and relevant in a given system depends on the underlying hard- and software and the intended application field. The ϵSOA platform can be extended with domain specific models that allow the addition of new properties. Note that the system model may not only comprise static properties, but also properties that are acquired at runtime and that are subject to change, such as energy resources, reliability metrics, etc.

The next level of abstraction is the *Service* layer that provides an overview of all services and applications running on the nodes in the network. Additionally it contains a service repository with new services that can be downloaded and installed on the nodes on demand. If a user wants to create a new application, he selects a set of services from the running services and/or the repository and composes these service to an application by connecting the in- and outputs of the services. Each of these services may possess a set of requirements, e.g., memory, cpu and bandwidth requirements, or hardware requirements, such as specific sensor or actor devices. These requirements have two purposes: they ensure that a service is only installed at nodes that possess all hardware requirements needed for the service, and they help avoiding overload situations which may occur if too many resource intensive services are installed at a single node. After the selection and composition of the services, the ϵSOA middleware will handle the installation of the new application in the network. An important step in this operation is the determination of a good placement of the services, i.e., the creation of a mapping between the services and the nodes these services should be executed on. We will present algorithms for the calculation of such a placement in Section 4. Based on this placement, the required services are installed at the nodes and the Broker tables are configured to allow the message exchange between the services.

3. METRICS

A prerequisite for the calculation of a service placement are metrics that allow to quantify the quality of a placement and allow comparing different placements. The available metrics depend on the information available in the system model. For the calculation of some of the metrics mentioned in this section, information from the routing layer is required to determine the routes used in the physical network for the transmission of data streams (the streams only specify the start and end point of the transmissions, not the hops in between). This information can be either supplied by a heuristic that calculates the shortest path between nodes in the physical network, or be queried through the cross layer interface.

At the current stage, we do not support timing constraints during the calculation of metrics. For some metrics, such as the CPU utilization, it is not only important how large the demand of a service for this resource is, but also *when* it is requested. If the underlying resources can only be used exclusively, simultaneous demands will result in delays and increase the time needed to execute a service. We are currently investigating how the execution model on the nodes and the timing requirements of the services can be incorporated to improve the calculation of the metrics described in the following paragraphs.

Many of the metrics described below require information from the system model regarding the capabilities of the available hardware and the requirements of the applications. In many cases, this information will be available immediately because it is contained in the hardware specifications or given by the application developer. If this is not the case, most information can also be collected by observing the service execution on the nodes. In this scenario the system will be launched with a placement based on a very simple metric, e.g., the hop count, and can be optimized when additional information is available through monitoring.

Currently we have implemented 6 metrics. Just like new properties in the system model, new metrics can be added easily to the system to allow a customization for specific application fields. For the utilization metrics, we will describe how the utilization coefficient for every node is calculated. These coefficients are combined to receive the overall utilization based on the maximum, mean or a specific percentile of the coefficients.

Hop Count

A simple metric that is always available is the hop-count. It is calculated by summing up the number of hops involved

for the transmission of all data streams flowing through the system. This very simple metric works fairly well for the optimization of the network utilization if the data streams used by the applications have similar data volumes.

Data Volume

If information about the expected volume of data-streams is available, the hop-count metric can be refined to calculate the data volume metric. This metric is based on the summed data volume transmitted over all links, i.e., the data rate of each stream multiplied with the number of hops needed for routing the stream. If an application comprises services that produce low data volume streams out of high data volume streams, e.g., a control service like the one presented in the air condition example that requires periodic measurements but only rarely issues commands to an actor service, this metric will ensure that the data consuming service is placed as close to the data producing services as possible (preferably on the same node). This metric closely resembles the heuristics used in systems like TinyDB, which "push" services as close to the stream sources as possible.

Link Utilization

The data volume metric can be further extended to calculate the overall link utilization metric, if additional information about the bandwidth of the links is available. For each link the summed data rates of all streams flowing through a link is divided by the link's bandwidth. If this coefficient is greater than one, the link is marked as overloaded[4].

Network Utilization

In many cases, logical links to different nodes are using the same physical communication medium, e.g., ethernet links using switches or wireless links interfering with each other. To avoid overload situations under these circumstances, an additional utilization metric, the network utilization is calculated for each physical communication medium available at each node. This is done by aggregating the data volumes for all logical links using the same physical medium, e.g., all ZigBee links.

Memory Utilization

The memory utilization metric can be calculated if information about the memory demand of services is available. In many cases this information can be determined by inspecting the service code. If this is not the case, the user has to specify it manually in the system model or it has to be determined at runtime by observing the memory usage of the running service.

CPU Utilization

The CPU utilization is calculated based on CPU cycles. The calculation of this metric requires information about the CPU capacity of each node, and an estimation of the required CPU cylces for the execution of each service.

[4]Placements containing overloaded resources are not immediately discarded because the user can opt to install the applications based on these placements anyway. This can be a reasonable decision of all placements result in overload situations and the middleware possesses features to compensate link congestions at runtime, e.g., by dynamically reducing the data acquisition rates at the sensor devices

Combined Metrics

All the individual metrics mentioned above can be combined with a weighting function to create an overall rating for a placement.

4. ALGORITHMS

The task of a placement algorithm is to determine an optimal placement, i.e., a placement with as little costs as possible, based on a user supplied weighting function for the metrics presented in the previous section and the system model containing information about the hardware characteristics and the application requirements. The optimization problem of distributing services to nodes can be easily mapped to the bin packing problem: the task is to distribute n services with resource demands $d_1 \dots d_n$ to m nodes with resource capacities $c_1 \dots c_m$ in a way that avoids overload situations. The problem is therefore NP hard. For small networks (< 10 nodes) and a small number of services (< 10 services), a solution based on a simple enumeration of all possible combinations is possible. For larger problem instances, more efficient solutions have to be applied.

We are analyzing well-known optimization techniques, such as Ant Colony Optimization, Simulated Annelaing and Genetic Programming w.r.t. their suitability for solving this optimization problem. We will present first results with these approaches in the following sections, a detailed analysis is currently work in progress. The optimization techniques are intended to be used on a central management node in the network that possesses global knowledge about the network topology, hardware characteristics and service requirements. This is typical the case for management nodes which control the application execution in an embedded network, or a subnet of a larger network. The algorithms aim at finding a global solution to the optimization problem, i.e., will move already installed services in the network if a new application should be installed and requires already occupied resources. These reorganizations come at a cost, because services have to be migrated between nodes and the corresponding applications will cease to work during the migration process. To provide a good trade-off between the migration costs and the long time savings of a new placement, the algorithms create a list of placements containing different levels of reorganization, which can be used by the user to select an appropriate placement. This is done by running the placement optimization multiple times with different restrictions for the placement of services, e.g., restricting all installed services to the node they are executed on will result in a scenario with no reorganization.

We also designed a distributed greedy heuristic to solve this problem, which we will present at the end of this section. This heuristic is beneficial in environments where central knowledge about the network strucutre is not available or too expensive to maintain, e.g., due to memory constraints. It requires only very little memory on the nodes, however the resulting placements can have a lower quality than the placements created with the centralized optimization algorithms.

4.1 Ant Colony Optimization

It is very difficult to apply the Ant Colony Optimization algorithm to the problem of mapping services to nodes. The reason for this is that the service placement problem exhibits

no optimal substructure in many cases. If one service is assigned to a node, this decission may influences other service assignments, because the assigned service will increase the resource utilization on the node and the used communication links. As a consequence, adding a single new service to an optimal placement may require a massive reorganization of the already assigned services in order to meet all resource constraints. Mapped to the Ant Colony Optimization algorithm this leads to the following problem: even if we found a fairly good "path", i.e., a placement with low costs, for a subset of our services we cannot reuse this information in subsequent runs. The assignment of other services can change the resource utilization on the nodes used in this subset and therefore render the solution invalid.

4.2 Simulated Annealing

The crucial part for the implementation of the Simulated Annealing algorithm is the specification of a suitable neighborhood function. Based on a given placement of services, this function should return a new placement that is a "neighbor" of the given placement. We have experimented with different functions: moving a single service to a neighbor node, moving a single service up to x nodes, moving a single service to a random node, moving multiple services at once, etc. The experiments were conducted on networks of different size and different topology (randomly generated networks and grid-like networks). Moving a single service to a random node in the network was the neighborhood function that yielded the best overall results. The neighborhood functions that move a single service to a neighboring node tend to get stuck in local optima, because it requires very many steps to move a service through the network. As a consequence the running time of the Simulated Annealing has to be extended considerably in order to achieve comparable results to the solution with random node selections. Moving multiple services at once resulted in worse results compared to the movement of a single service. This is most likely due to the very large number of possible neighbors, especially in larger networks. We also experimented with different temperature functions and initial placements, but these only had minor impact on the overall execution time of the annealing algorithm and no measurable impact on the quality of the resulting placements. These preliminary test also showed a drawback of the Simulated Annealing approach: in a typical network there are a lot of sub-optimal solutions which have costs very close to the optimal solution and Simulated Annealing will very likely choose one of these sub-optimal solutions.

4.3 Genetic Programming

As our first tests show, Genetic Programming seems to be the most suitable strategy. As mutation function we chose the neighborhood function already used in the Simulated Annealing approach, i.e., to mutate a genome we move one service to a randomly chosen new node. If elitist selection is applied, i.e., the currently best genome is always preserved in the gene pool, Genetic Programming is capable of finding the optimal solution even if there are a lot of sub-optimal solutions with small cost differences. We are currently experimenting with different crossing algorithms, e.g., we create a new placement out of a subset of service to node assignments from one genome and the remaining assignments from the other genome.

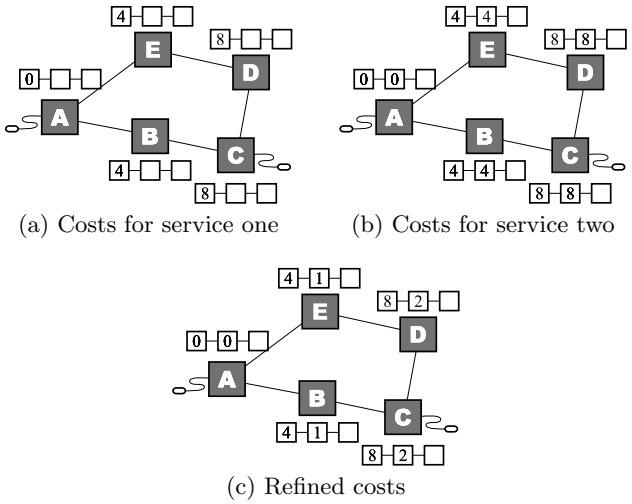

Figure 4: Scenarios for Placement Heuristic

4.4 Distributed Heuristic

The distributed heuristic operates solely on local knowledge available at each node. It provides only a local optimization, i.e., it will determine a good placement if an additional application should be installed in the system, but it will not re-arrange running applications in order to achieve a globally optimal solution. It is intended to be used if the network has to react to sudden changes in the underlying topology, e.g., when a node enters or leaves the network, and no global optimization is possible. Due to the distributed execution of the heuristic, no central controller is required that possess a global view of the embedded network.

The heuristic is based on a simple broadcast mechanism which is illustrated with the example shown in Figure 4(a). Assume we want to place a simple application consisting of a chain of 3 services. In order to keep the example concise, we will use as metric solely the transmitted data volume. Assume that the data stream between the two leftmost services of the application is 4, and the data rate for the stream between the two rightmost services is 1. Further we assume that the third service of the application is restricted to run on node E.

Initially the description if the application is distributed to all nodes. After that, each node iteratively calculates the costs for getting the output stream of each service in the application chain and broadcast this value to all nodes in the network. In the first round, the node containing the required sensor device (node A) will therefore announce a cost value of 0 for the first service. Based on this announcement, nodes B and E can determine that they can provide the data stream with a cost of 4 (a stream with data rate 4 has to be transmitted over 1 link), whereas nodes C and D can provide the data with costs 8 (stream with data rate 4 routed over 2 links). In the next iteration, each node calculates the costs for hosting service two of the chain. This results in the values shown in Figure 4(b): 0 for node A, 4 for node B and E and a value of 8 for nodes C and D (because the execution of service incorporates no costs in our metric, the costs for executing a service are the costs for receiving its required input data). The new information is now distributed in the

network. Services B, C, D and E now discover, that it is cheaper to let node A host the second service. This is due to the very low data rate of 1 of the output stream of the second service. As a consequence, the nodes update their costs to a value of 1 (the costs for transmitting a stream of volume 1 over one hop), and 2 respectively (see Figure 4(c)). The third service can only be executed on node E, what yields a total cost for the execution of the application of 2.

The worst case network traffic created by this heuristic is a broadcast from each node for every position of the application. So this approach is only feasible for small applications. In other scenarios, it is advisable to use the Simulated Annealing based optimization.

5. RELATED WORK

There are other projects that use a service oriented application model for the development of embedded networks. Examples are the the DPWS[4] based SIRENA[7] and SOCRADES[3] projects and the OASiS[8], MORE[10], and RUNES[2] projects. These systems could benefit from the optimized placement of services based on application requirements and network characteristics shown in this paper.

Regarding related work with respect to optimal placement of services/aggregators most work deals with sensor networks that perform monitoring tasks [9, 15, 11, 1]. In such systems, applications/queries can be organized in a tree-like structure. In contrast to this related work, this paper presents a solution for sensor-actuator networks. It allows optimizing applications that are not centered around a dedicated sink node and it allows a global optimization of embedded networks that takes into account interferences between multiple simultaneously executed applications.

6. SUMMARY AND ONGOING WORK

In this paper we motivated the problem of adaptive execution of applications in heterogeneous embedded networks comprising nodes with different capabilities and different communication channels. In our approach, the placement of services is optimized based on application requirements and the characteristics of the underlying hardware. We presented a set of metrics that allow quantifying the quality of service placements, and stated the optimization problem that has to be solved in order to compute an optimal placement. We outlined preliminary results for well-known optimization techniques and a heuristic that can be executed without central knowledge about the network structure.

Besides the ongoing evaluation of the optimization techniques, we also plan to to extend the model driven development approach. A crucial part of the application execution in embedded networks is the communication between services. Some services have very loose QoS requirements and may tolerate packet loss, e.g., readings from a sensor device sending data every second, whereas other services demand the reliable and timely transmission of messages, e.g., a fire alarm. We are currently investigating how these requirements can be specified at the application level and what mechanisms are required in order to guarantee QoS requirements across heterogeneous network infrastructures comprising multiple different network technologies and communication protocols. Another direction for research are fault tolerant systems. The algorithms presented in this paper can be used to provide some very basic failure recovery mechanisms, e.g., to calculate a new placement if a node fails and to re-configure the application to not use this node. However, this approach is not feasible for systems that have to recover from errors very fast. We are investigating which mechanisms can be used to minimize the application downtime, e.g., by pre-calculating alternative service placements or by installing redundant services based on the modeled reliability requirements.

7. REFERENCES

[1] B. J. Bonfils. Adaptive and decentralized operator placement for in-network query processing. In *In IPSN*, pages 47–62, 2003.

[2] P. Costa, G. Coulson, C. Mascolo, G. P. Piccoand, and S. Zachariadis. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In *PIMRC'05*, 2005.

[3] L. de Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio. SOCRADES: A Web Service Based Shop Floor Integration Infrastructure. *IOT'08*, pages 50–67, 2008.

[4] Devices Profile for Web Services. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, 2004.

[6] M. Gauger, P. J. Marrt'on, and C. Niedermeier. TinyModules: Code Module Exchange in TinyOS. In *INSS'09*, 2009.

[7] F. Jammes and H. Smit. Service-oriented Paradigms in Industrial Automation. In *IEEE Transactions on Industrial Informatics*, volume 1, pages 62–70, 2005.

[8] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits. OASiS: A Programming Framework for Service-Oriented Sensor Networks. In *COMSWARE'06*, 2007.

[9] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *TODS*, 30(1):122–173, 2005.

[10] MORE – Network-centric Middleware for Group communication and Resource Sharing across Heterogeneous Embedded Systems. http://www.ist-more.org/.

[11] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *In ICDE*, 2006.

[12] A. Scholz, C. Buckl, S. Sommer, A. Kemper, A. Knoll, J. Heuer, and A. Schmitt. ϵSOA – service oriented architectures adapted for embedded networks. In *INDIN'09*, 2009.

[13] A. Scholz, I. Gaponova, S. Sommer, A. Kemper, A. Knoll, C. Buckl, J. Heuer, and A. Schmitt. Efficient communication in control-oriented embedded networks. In *ETFA'09*, 2009.

[14] TinyOS. http://www.tinyos.net/.

[15] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.