Workflow Support for Wireless Sensor and Actor Networks

A Position Paper

Pablo Ezequiel Guerrero* Daniel Jacobi† Alejandro Buchmann

Dept. of Computer Science
Technische Universität Darmstadt
D-64283 Darmstadt, Germany
{guerrero, jacobi, buchmann}@dvs1.informatik.tu-darmstadt.de

ABSTRACT

As initial challenges of wireless sensor and actor networks (WSANs) are overcome, their application possibilities evolve. For these applications to move mainstream, efficient programming methods are required which can be used by domain experts. So far, the question of how can WSANs be efficiently programmed remains unanswered. In this paper we examine proposed middleware approaches, and show that they have focused on data extraction rather than in-network actuation. We thus propose the usage of *workflows* as a means to define the logic that orchestrates the network activity, and introduce a language to express WSAN interactions. At this time, a concrete system is not given, but the paper discusses the relevant aspects towards one, and poses many questions for future research.

1. INTRODUCTION

Ever since the idea of merging fundamental *sensing*, *processing* and *wireless communication* capabilities into tiny devices emerged [26, 27], an enormous progress has been made to get large numbers of low-cost, battery-powered nodes to carry out collective tasks [2]. The tight collaboration between experts from different areas leveraged a technology spread over many domains. Indeed, the last 5 years have seen a number of large experimental deployments of *wireless sensor network* (WSN) applications.

One of the earliest deployments was on Great Duck Island [18]. The study, in cooperation with researchers in the Life Sciences, aimed at unobtrusively learning about seabirds and their environment, by installing sensor nodes in and around their burrows during nesting periods. Another deployment, in this case together with glaciologists, aimed at investigating the behavior of glaciers by inserting sensor nodes in them [19]. A third deployment, now involving multi-hop communication, was carried out in dutch potato fields [13]. There, crops must be protected against fungal diseases, which are strongly associated to the climatological conditions within the field. The deployments in this first group faced many issues in common such as network longevity, remote administration and unobtrusive monitoring. Their application logic, however, is quite simple: the (mostly) raw observed data must be pushed out of the network. Note that the parameters of this logic, i.e., the sampling rates, are normally specified by the domain experts.

In a second group of applications, a continuous observation of the environment is not required. In contrast, the slightly more complex goal is to detect an event of interest and observe the phenomena afterwards. One such deployment was that carried out with volcanologists at Ecuador's Volcán Reventador [37]. Each node waits until its seismometer readings exceed a certain threshold, and then notifies a base station, which triggers a data collection phase. Another example in this group are structural health monitoring systems like Wisden [40] or the one deployed on the Golden Gate Bridge [12]. The event detection is followed by local data storage and posterior progressive coding (compression) for efficient data transmission. The parameters of the application logic, i.e., the sampling rates and the thresholds, are also given by the experts in the domain, who may vary them to adjust or refine the experiment.

A third type of applications is considerably more complicated. The habitat monitoring system deployed at the Coastal Redwood Forests of California [17] allows complex queries to be injected into the network, whose results are aggregated as they are streamed back to a base station. In the industrial scenario of the CoBIs project [33], nodes attached to chemical drums cooperate without using any external infrastructure to check for hazardous situations and violations of safety regulations. Again, the logic behind these applications is precisely stated by domain experts, now in the shape of SQL-like queries or inference rules, respectively.

From this categorization, it is clear that the research focus has been on data extraction and event detection. While we observe that many operations have been effectively moved into the network, the decision on how, when and where to perform certain actuation is only taken off the network, either by a *human*, or with the help of a *decision support system*. This is logical, since only after data had been consolidated in a central server was it possible to reason about it and decide what to do next. However, a WSN can be easily extended with other nodes further capable of *actuating*, forming a *wireless sensor and actor network* (WSAN) [1]. As an example, WSANs are needed for the control of autonomous vehicles (AVs) and the coordination of swarms thereof. Individual actuators are as diverse as sensors; in

*Supported by the DFG Graduiertenkolleg 492, *Enabling Technologies for Electronic Commerce*.

†Supported by the DFG Graduiertenkolleg 1362, *Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments*.

4th International Workshop on Data Management for Sensor Networks.
DMSN'07, September 23-28, 2007, Vienna, Austria.

essence they can either open or close a switch, or set a value in some way [11]. With these, the loop of event-sensing, decision and acting can be closed, and even lead to a reduced need for unnecessary, slow and error-prone human intervention in the process. It is thus natural to foresee that a next step is to support *in-network actuation*. In this way, the whole WSAN loop can be shifted to the network. Intuitively, this approach presents a number of benefits, namely:

- faster reaction to the event, as the decision is taken closer to the point of interest,
- enhanced reliability, due to the smaller chance of losing messages in the loop sequence, and
- energy savings (i.e. extended network lifetime) for the reduced amount of messages exchanged between event sources, sinks and actuation nodes.

These benefits, however, don't come for free. To achieve them, there exist several challenges that a WSAN platform must overcome. One of such challenges, which constitutes the central theme of our work, is how to define the logic that orchestrates the WSAN activity. As can be noted in the previous application categories, it is the domain experts who have the knowledge of the behavior of the target environment and what is needed to do with it as a response. They can provide engineers with very detailed information of what is expected to happen, either from what they have observed, learned or suspect and want to corroborate. In this position paper we argue that this behavior, which we have called *application logic*, can be naturally expressed through a *workflow* with relative ease.

The rest of this paper is organized as follows. In the next section we examine the existing middleware approaches' suitability for programming WSAN interactions. Section 3 proceeds with a description of workflows and how they can be applied to express WSAN application logic. In Section 4, we identify further platform possibilities and propose future directions to realize a WSAN workflow middleware. Finally, in Section 5 we draw the conclusions.

2. MIDDLEWARE APPROACHES

The aforementioned applications dictate requirements of increasing complexity, e.g., long life span, fast response, fault-tolerance, secure communication, flexibility or reconfigurability, etc. Due to the technical constraints of the sensing platforms, developing such applications using a language like nesC and TinyOS [10, 6], or even reutilizing them in slightly different environments, poses a highly complex task. Fortunately, research in the recent past has produced the necessary low level building blocks like MAC protocols, routing, localization or time synchronization. Composing a system with these, i.e., putting the correct pieces together for a particular end application, nevertheless, is still a difficult engineering endeavor. In order to alleviate this problem, *middleware* has been proposed as a means to ease the development of applications [30, 41, 34, 9, 8].

A major task of the middleware is to raise the level of abstraction for programming distributed applications. From a programmers' point of view, there are two types of middleware designs: node-centric and network-centric.

Using the first design, programmers have to deal with how to do things in a node, e.g., get the temperature or process a packet and handle a response. One of the earliest systems categorized as node-centric middleware is Maté [14]. It employs a virtual machine (VM) architecture to enable portability across several sensor platforms. Maté's instruction set is concise (instructions have only 1 byte), while programs are broken into 24 instructions' *capsules*. This approach presents a number of drawbacks for WSAN's application logic. First, Maté's programs are epidemically flooded through the network, impeding a selective decision on which nodes should do what. This issue is faced by the Agilla [5] mobile agent middleware, which is built on top of Maté and extends it with explicit *migrate* and *move* operations. Mobile agents are focused on working in a local manner as well. Second, although Maté hides certain complexity from the developer such as routing details or asynchrony, the level of abstraction is still quite low: bytecodes. In this regard, SwissQM [22] expands work on VMs to provide compilers for higher level languages such as XQuery, SQL or Java. Finally, it is noted that VM's overhead for bytecode interpretation (which is translated to energy consumption) is not necessarily traded off for platform *interoperability*, highly desired in heterogeneous environments.

By viewing the network as its most important part, many approaches for WSNs follow the second, network-centric, design. A first known subgroup, which exposes a high level of abstraction, are *sensor database systems*. In these systems, an SQL-like query language is used to extract data from the network nodes. Cougar [4] is one of the first models of this kind. In Cougar, every sensor type is modeled as an abstract data type (ADT), whose public interface consists of the supported signal processing functions. Every ADT object in the database represents one physical sensor in the real-world. Cougar extends the traditional central approach with the ability to gather only data needed for a user query, instead of extracting all the data from the physical environment. By following this central approach, however, Cougar inherits also its drawbacks: it stores the data in a fixed node and answers user queries from it. This leads to reduced energy resources at nodes in the area around this fixed node. In addition, Cougar utilizes streams and long running queries, without considering the energy constrains of the sensors.

In contrast to Cougar, TAG [16] and TinyDB [17] are the first attempts to consider these energy constrains and incorporate a distributed execution of a query. TAG pursues reducing the power consumption on nodes that have to deliver data to a central server. For this, it partially moves computing from the server to the network: it uses *aggregation queries* to aggregate queried values in the network and keep the amount of transferred data over all nodes small and similarly distributed. A drawback of TAG is that it uses only aggregation queries, which are flooded through the network regardless if a node has data for the query or not. TinyDB solves both problems of TAG. It creates an index over all constant attributes and uses this to send a query only to nodes that can participate in its execution. TinyDB is the first system where the user does not need to write low level code to query data, because it provides an interface to send SQL-queries. Even though TinyDB provides basic support for actuation queries, its main aim is to get data out of the network. In general, these systems have almost no possibilities to send data back into the network, as is required for WSAN's to control actuators.

Another subgroup of network-centric middleware targets

clustering or grouping nodes. *Hood* [38] and *Abstract Regions* (based on Hood) [36] focus on building local clusters of neighboring nodes to exchange data. Hood restricts the clusters to one-hop neighbors, while Abstract Regions can recruit nodes in a n-hop distance or, with some location information, in a specific radius. To communicate with each other, trees or meshes are used. As every node builds its own neighborhood, a single node can be a member of several neighborhoods and in each neighborhood data is exchanged, aggregated and processed.

The approach used in *Generic Role Assignment* (GRA) [29] is to define different roles for the network nodes. Every node receives the same specification of roles and program images. At runtime, each node picks a role out of the specification depending on several conditions. For instance, a *'cluster head'* role in a routing algorithm could depend on the number of neighboring nodes. *Scopes* [32] combines many advantages of the previous systems. It can group different nodes by properties over the whole network, different groups can execute different applications and it is also capable of creating nested groups. The applications can be sent to the groups while the network is already deployed in the field and so can easily change the tasks to execute. Nevertheless, Scopes presents two drawbacks. The first is that the approach can be inefficient when executing short tasks. The second, which depends on the used routing scheme, is that it may be impossible to perform in-network aggregation due to the complexity in incorporating it into its algorithms.

A latest subgroup of network-centric middleware design is based on the idea of macroprogramming. *Regiment* [24] is one such macroprogramming language. It relies on *Region Streams* [25] to get routing tasks done. With Regiment, the network as a whole is modeled in one macro-program. The tasks a single node has to process are extracted from this code. To get data from one part of the network, some properties are specified to identify a region of nodes and the data sent back from this region is represented by a Region Stream. They further offer additional executable operators, for example, to aggregate raw data. In the *state-centric programming* approach of Pieces [15], the concept of node clusters is also present. Programmers, however, focus on dividing the global state of physical phenomena into a hierarchical set of independently updatable pieces. These pieces are encapsulated by *principals*, who update their state by interacting with other principals through *port agents*. Although we agree with many ideas behind Pieces, the lack of a more detailed description of the programming primitives (and possibly an implementation) makes it difficult to conclude whether it is suitable for in-network actuation. Another approach along the same lines, but for which a concrete implementation exists, is Kairos [7]. Here, programmers are offered three new language constructs, namely *node*, *one-hop neighbors* and *remote data access*. These are used to program the global behavior of a distributed application implicitly. This single centralized program is later compiled into a node-level version for each node. Its authors showed that the easiness in writing Kairos' programs comes at the cost of a lower performance, although within 2 times that of the original ones (i.e., distributed explicitly).

We finalize this examination with two macroprogramming approaches that illustrate the effectiveness of programming large numbers of nodes using *data flows*. In the *Abstract Task Graph* approach [28], programs consists of two parts. The first, *declarative* part specifies the program's tasks and constraints on their placement and communication. The second, *imperative* part contains the node-level implementation of the task in a traditional computer language. Similarly, in Cosmos [3], programs are composed of *functional components* which provide computing primitives, and *interaction assignments*, which specify the dataflow through the former. While we consider that analyzing an application's data flow is necessary, we also think that attention must be paid to the control flow.

The previous (by no means exhaustive) analysis provides us with enough evidence that, even though many are heading in the right direction, the existing approaches fall short for WSAN interactions. In the next section, therefore, we propose what we believe can be an effective programming paradigm that produces efficient WSAN programs.

3. THE CASE FOR WORKFLOWS

The use of workflows to express WSAN application logic embraces two major benefits. First, by keeping the workflow programming abstractions *simple* [31], we can bring WSAN programming closer to domain experts. The application logic can be defined using workflow modeling tools, whereas the WSAN runs an infrastructure which senses and generates the data that causes the state transitions in such workflow and executes their associated actions. A second advantage is that general purpose specification formalisms can be used for *formal correctness reasoning* and doing *model checking*, for instance through Petri nets [23] or state and activity charts [39].

The term workflow has been overused in the literature to the point of requiring a further description when used. For this purpose, we introduce the elements that make up our workflow programs following a bottom-up approach. The first element defines the application's event *filters*, which are expressions that match with a given event or not.

DEFINITION 3.1. *Let $\mathcal{F}$ be the set of all possible event filters, and F be a proper subset of it.*

The second element needed defines the application's actions, which can be seen as executable *code*.

DEFINITION 3.2. *Similarly, let $\mathcal{C}$ be the set of possible action codes, and C be a proper subset of it.*

We now introduce the network's nodes.

DEFINITION 3.3. *Let $\mathcal{N}$ be the set of network nodes.*

Note that $\mathcal{N}$ is purposely defined as an infinite set so that new nodes can always join the system.

DEFINITION 3.4. *Let $R = \{R_1, R_2, \ldots, R_n\}$ be the finite set of node roles, where $R \neq \varnothing$, such that $\forall R_i \in R : R_i \subseteq \mathcal{N}$.*

Roles are functions that *select* subsets of nodes, e.g., with node ids, proximity, with hop counts or sensor properties.

We now characterize a workflow program as a set S of logical *states* an application can be in, and a specification of *control flow* transitions T between them (cf. Definition 3.5). There is an initial state designated s_0, and a non-empty subset of S with the final states called S_F. The state transitions, T, have two annotations. The first annotation, E, is composed of a *filter*, which describes occurrences of the event

DEFINITION 3.5. *A* workflow program *is a tuple* $WP = (S, s_0, S_F, R, T)$,
where $S = \{s_0, s_1, s_2, \ldots, s_m\}$ *is the finite set of application states,* $S \neq \varnothing$, $s_0 \in S$ *is the initial application state,*
$S_F \subseteq S$ *is the set of final application states,* $S_F \neq \varnothing$,
$E = (F \times R)$ *is a shorthand for event annotations,* $A = (C \times R)$ *is a shorthand for action annotations,*
$T \subseteq ((S \times S) \times E \times A)$, *is an annotated transition between two states,*
such that $\forall s_i \in (S \setminus S_F)\ \exists\,(t, s_j, e, a) \in (T \times S \times E \times A) : t = ((s_i, s_j), e, a)$, *and*
$\forall s_j \in (S \setminus \{s_0\})\ \exists\,(t, s_i, e, a) \in (T \times (S \setminus S_F) \times E \times A) : t = ((s_i, s_j), e, a)$.

that can match with it for the transition to be followed, and a *role*, which at runtime delimits the set of nodes that will participate in the detection of the event. In turn, the second annotation, A, is composed of an action *code*, that must be executed as a response when the transition is chosen, and another *role*, also defining which nodes will participate in the execution. Finally, the first constraint simply enforces that every state, except those that are final, must have an outgoing transition, whereas the second constraint makes every state, except the initial, have an incoming transition.

A generic transition between two application states s_1 and s_2 is illustrated in Figure 1 a). Such transition is triggered when an *event* matching the filter f_e is detected by the nodes in role r_e. The associated *action* is taken by executing c_a at nodes in role r_a. Note that the event annotation of a transition can be omitted, in which case the transition is triggered unconditionally. The action can also be omitted, indeed executing no action code.

The definition of a workflow is flexible enough to accommodate innumerable WSAN programs. We now describe the semantic of common compositions from a control flow perspective. In Figure 1 b), a typical *iteration* is shown. The case when a state has multiple outgoing transitions, allowing *choices* to be followed, is depicted in Figure 1 c). Each of these choices need not be disjoint, i.e. the filters f_{e1}, f_{e2}, etc., can overlap. In such case, the control thread is indeed forked with those transitions whose filter match the detected event. Note, furthermore, that the resulting threads could execute sequentially (in no particular order) or in parallel. Finally, the syntax allows multiple transitions to converge into a single state, as depicted in Figure 1 d).

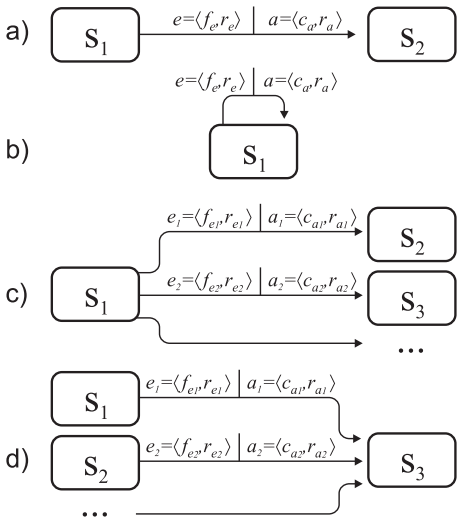

Figure 1: Workflow compositions

Given our formal language syntax, we proceed with a simple algorithm used to execute workflow programs for WSANs, depicted in Algorithm 1. For this purpose, an entity called *workflow manager* is introduced, whose duty is to loop tasking the nodes to sense and actuate through the corresponding workflow states. Given that control can be split into parallel threads, the workflow program's state is really described with a set, namely *CurrentState*. A thread dies whenever it reaches a final state. Additionally, when two or more threads converge simultaneously into a single state, they merge. At any point, $|CurrentState|$ equals the number of parallel threads. When every thread has finished, the workflow execution concludes.

Algorithm 1 Workflow Manager Execution Algorithm

```
1: CurrentState ← {s0}
2: while CurrentState ≠ ∅ do
3:    for all t = ((s, d), e, a) ∈ T such that
                 s ∈ CurrentState do
4:       Given e = (fe, re),
           task nodes in role re to detect event fe
5:    end for
6:    for all t = ((s, d), e, a) ∈ T such that
                 event e = (f, r) was detected do
7:       Given a = (ca, ra),
           task nodes in role ra to actuate with ca
8:       CurrentState ← CurrentState \ {s}
9:       if d ∉ SF then
10:         CurrentState ← CurrentState ∪ {d}
11:      end if
12:   end for
13: end while
```

4. DISCUSSION AND DIRECTIONS

The preliminary state of this work led us to intentionally avoid providing a particular syntax for the formal specification of a workflow program given in the previous section, e.g., for the definition of roles, and event and action annotations. In this section we discuss how can this syntax be concretized. There exists a tradeoff between a language's expressiveness and the spectrum of possible applications for which it can be used. Approaches from Section 2 could fit.

The event annotations, for instance, could range from TinyDB's queries [17], through Agilla's mobile agents [5], to Regiment's streams [25]. Previous work on publish/subscribe abstractions has shown event-based communication to be suitable for low-power, unreliable devices. Defining events that can be evaluated inside the network leads to near optimal energy consumption, provides deployment flexibility and leads to fault-tolerance in the event detection even when some nodes fail. These properties would allow the workflow execution to proceed even under adverse conditions.

State transitions can be further triggered due to *temporal* events, i.e., at an absolute date or relative to another event (when the state was entered, a threshold was exceeded, etc). Logical combinations of temporal and spatial constraints should also be valid. An action's code could not only cause a particular set of nodes to actuate but also set or update *state variables* which would be kept in a shared space. This could be later read by an event or another action.

The workflow language can be extended to optionally include *conditions* in the transitions. In this way, transitions can be seen as full Event-Condition-Action rules. A condition is a boolean expression that evaluates to `true` or `false`, whereas the algebra typically supports the operators `sequence`, `and`, `or` and `not`. The usage of conditions can be practical for developers, leading to a more clear design, as events become simpler (not overloaded with the extra guards), at the cost of adding complexity to the language. The major difference between events and conditions is that events are pushed towards the workflow manager, while condition's results are usually pulled by it. Finally, conditions could make use of the state variables in the shared space.

For the definition of an event or an action's role, which selects network nodes for a particular event detection or action execution, approaches like GRA [29] or Scopes [32] are suitable. Here, it will be important to distinguish between a *static* role (i.e., one evaluated only once, either at deployment-time or at run-time) and a *dynamic* role (i.e., re-evaluated with a certain policy).

The workflow compositions shown in Figure 1 can be further extended with numerous patterns as observed in the work from von der Aalst *et al.* [35]. These patterns allow to explicitly describe whether a single or multiple choices can be followed when a state has multiple outgoing transitions; whether a thread must wait until other finish when a state has multiple incoming transitions; etc. This will clearly extend the workflow language syntax.

The problem of how to coordinate sensors and actor interactions has been introduced in [20]. The described interaction, however, is rather simplistic. It consists of an aggregate and actuate loop. In contrast, we are looking at providing means to describe an arbitrary application logic. Our algorithm is naïve in that it assumes a centralized workflow manager, who decides which transitions to follow in an atomic fashion given input events that arrive sequentially. This would be adequate, e.g., for workflows involving control of a single AV. In a distributed system, however, designating a single node for this task results in a system with a single point of failure, and leads to uneven energy consumption at that node and around it. On the other extreme, letting every node assume this manager role requires synchronization, a problem that is related to distributed shared memory models [21] and requires future work.

A number of questions issues is raised by such middleware:

1. *Workflow updates and composition*: due to the distributed nature of a running workflow program, how can it be partially updated to accommodate program changes? How can a state be replaced or dynamically composed with an inner workflow?

2. *Multiprogramming*: how can the platform, once running a particular workflow program, accept a second one? How can similarities between these two be exploited so that several workflows can be executed simultaneously in an energy-efficient manner?

3. *In-network storage*: how can state be distributed across nodes for efficiency and reliability reasons? how can it be kept updated?

4. *Security*: given a concrete middleware architecture, how can security be built in so that it is ensured that the network is resilient to attacks?

5. *Logging*: in case a human operator monitors the network, how can the decisions taken by the workflow middleware be distributedly logged at a particular sink? How can the system inform about the confirmed state transitions to support a debugging process (and possibly find out where the system got blocked)?

Ultimately, for an end-to-end software solution, various CASE tools must be available to domain experts. Many workflow development GUIs exist, but they would need to be adjusted to reflect the particular semantics of our language.

5. CONCLUSIONS

In this paper we have shown that a next step in order to move forward with WSANs' application logic is to provide effective means to orchestrate the network's sensing and actuation. We have examined existing middleware approaches, pointing out that they fall short due to their focus on data extraction and not on in-network actuation. We are addressing WSANs in the context of autonomous vehicles. We have proposed the use of workflows to describe this application logic, and described a sensing and actuation language that offers the basic workflow operators. The paper raises many questions requiring further research. Initial tradeoffs for a middleware platform supporting workflow interactions were identified and discussed. We expect to continue our work in the area by instantiating the language with concrete examples, use simulations to sort out the most relevant platform issues and then move on with a concrete implementation.

6. REFERENCES

[1] I. Akyildiz and I. Kasimoglu. Wireless Sensor and Actor Networks: Research Challenges. *Ad Hoc Networks*, 2(4):351–367, October 2004.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: a Survey. *Computer Networks*, 38:393–422, 2002.

[3] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming Heterogeneous Sensor Networks using Cosmos. In *EuroSys'07*, March 2007.

[4] P. Bonnet, J. Gehrke, and P. Seshadri. Querying the Physical World. *IEEE Personal Communications*, 7:10–15, October 2000.

[5] C. Fok, G. Roman, and C. Lu. Agilla: A Mobile Agent Middleware for Sensor Networks. Technical Report WUCSE-2006-16, Wa. Univ. in St. Louis, March 2006.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI'03*, pages 1–11, New York, NY, USA, June 2003. ACM Press.

[7] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming Wireless Sensor Networks using Kairos. In *1st DCOSS*, pages 126–140, June 2005.

[8] S. Hadim and N. Mohamed. Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online*, 7(3), March 2006.

[9] K. Henricksen and R. Robinson. A Survey of Middleware for Sensor Networks: State-of-the-Art and Future Directions. In *MidSens'06*, pages 60–65, New York, USA, 2006. ACM Press.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS-IX*, pages 93–104, December 2000.

[11] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley & Sons, June 2005.

[12] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *6th IPSN*, pages 254–263, New York, NY, USA, 2007.

[13] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *14th WPDRTS*, April 2006.

[14] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS-X*, pages 85–95, New York, NY, USA, October 2002. ACM Press.

[15] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-Centric Programming for Sensor-Actuator Network Systems. *IEEE Pervasive Computing*, 02(4):50–62, October 2003.

[16] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-hoc Sensor Networks. *5th USENIX OSDI*, 36(SI):131–146, 2002.

[17] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: an Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005.

[18] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *1st ACM Intl. Workshop on Wireless Sensor Networks and Applications*, pages 88–97, New York, NY, USA, September 2002.

[19] K. Martinez, P. Padhy, A. Elsaify, G. Zou, A. Riddoch, J. K. Hart, and H. L. R. Ong. Deploying a Sensor Network in an Extreme Environment. In *IEEE SUTC*, volume 1, pages 186–193, June 2006.

[20] T. Melodia, D. Pompili, V. C. Gungor, and I. Akyildiz. A Distributed Coordination Framework for Wireless Sensor and Actor Networks. In *6th MobiHoc*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[21] D. Mosberger. Memory Consistency Models. *Operating Systems Review*, 27(1):18–26, January 1993.

[22] R. Müller, D. Kossmann, and G. Alonso. A Virtual Machine for Sensor Networks. In *EuroSys'07*, Mar. 07.

[23] T. Murata. Petri nets: Properties, Analysis and Applications. *Procs. IEEE*, 77(4):541–580, April 1989.

[24] R. Newton, Arvind, and M. Welsh. Building up to Macroprogramming: an Intermediate Language for Sensor Networks. In *4th IPSN*, pages 6–13, 2005.

[25] R. Newton and M. Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *1st DMSN*, pages 78–87, Toronto, Canada, August 2004. ACM Press.

[26] K. Pister, J. Kahn, and B. Boser. Smart Dust: Autonomous Sensing and Communication in a Cubic Millimeter. `http://robotics.eecs.berkeley.edu/~pister/SmartDust/`, 1999.

[27] G. Pottie and W. Kaiser. Wireless Integrated Network Sensors. *Comm. of the ACM*, 43(5):51–58, 2000.

[28] V. Prasanna, J. Reich, A. Bakshi, and D. Larner. The Abstract Task Graph: a Methodology for Architecture-Independent Programming of Networked Sensor Systems. *EESR'05 Workshop*, pages 19–24, June 2005.

[29] K. Römer, C. Frank, P. Marrón, and C. Becker. Generic Role Assignment for Wireless Sensor Networks. In *11th ACM SIGOPS European Workshop*, pages 7–12, Leuven, Belgium, September 2004.

[30] K. Römer, O. Kasten, and F. Mattern. Middleware Challenges for Wireless Sensor Networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):59–61, October 2002.

[31] A. Sharp and P. McDermott. *Workflow Modeling: Tools for Process Improvement and Application Development*. Artech House Computing Library, Norwood, MA, April 2001.

[32] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann. Scoping in Wireless Sensor Networks: A Position Paper. In *2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 167–171, October 2004.

[33] M. Strohbach, G. Kortuem, and H. Gellersen. Cooperative Artefacts - A Framework for Embedding Knowledge in Real World Objects. In *Smart Object Systems Workshop at UbiComp 2005*, September 2005.

[34] K. Terfloth and J. Schiller. Driving Forces Behind Middleware Concepts for Wireless Sensor Networks. In *Workshop on Real-World WSNs*, June 2005.

[35] W. van der Aalst, A. Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[36] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *1st USENIX/ACM NSDI*, pages 29–42, March 2004.

[37] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[38] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *2nd MobySys*, pages 99–110, June 2004.

[39] D. Wodtke and G. Weikum. A Formal Foundation for Distributed Workflow Execution Based on State Charts. In *6th Intl. Conference on Database Theory*, volume 1186 of *LNCS*, pages 230–246, January 1997.

[40] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A Wireless Sensor Network for Structural Monitoring. In *2nd SenSys*, pages 13–24, November 2004.

[41] Y. Yu, B. Krishnamachari, and V. Prasanna. Issues in Designing Middleware for Wireless Sensor Networks. *IEEE Network*, 18(1):15–21, January 2004.