

Using TinyOS Components for the Design of an Adaptive Ubiquitous System

Omer Sinan Kaya, Ozlem Durmaz Incel, Stefan Dulman, Roland Gemesi,
Pierre Jansen, Paul Havinga
The Department of Computer Science
University of Twente
PO-Box 217, 7500 AE Enschede
The Netherlands
{o.s.kaya, durmazo, dulman, gemesir, jansen, havinga}@cs.utwente.nl

I. ABSTRACT

This work is an initiative attempt toward component-based software engineering in ubiquitous computing systems. Software components cooperate in a distributed manner to meet a demand, and adapt their software bindings during run-time depending on the context information. There are two main research topics investigated in this study. The first topic is how to build an architecture, consisting of software components, that supports adaptation and self-configuration. We explain why that component is needed, what the requirements are and how it will be designed. Besides component-based design, we build our architecture on top of Publish/Subscribe (P/S) model. We like to reuse the TinyOS components and as a second topic, we investigate the problems that will be experienced when converting these components to our P/S oriented environment. Our experiences during this research pointed out that; buffer exchange and flexible interface name representation are not suitable for ubiquitous systems.

II. INTRODUCTION

Ubiquitous Computing is the calm technology as envisaged by Mark Weiser in [1] that recedes into the background of our daily lives. With the recent advances in the existing technology, this vision is about to become reality. Wireless communication has found widespread use in daily life and people are seeking for information access anytime, anywhere. Mobile phones made a big impact in the market after many commercial companies began to give mobile phone services. The next challenge is to have pervasive, wearable, unobtrusive, disappearing, or invisible computers. This introduces a paradigm shift from personal computing to ubiquitous computing, challenging the research community to study new building blocks and integrated infrastructures, as well as emerging applications and interaction styles.

This calm model needs software systems to work autonomously without user intervention. Becoming part of the daily life requires a system to adapt to environmental changes. A device should continually monitor the environment and itself

to change its behavior. For instance, a digital user agenda can fetch content depending on the user location (for example, at home and work or while traveling). While at home, it can bring a user's tasks inside the family (for example, shopping or bill payments); at work it can remind the user of meetings. Therefore, a device needs to recognize where exactly and in which context it is.

The basic parts of such a smart environment will be small nodes with sensing and wireless communications capabilities which are able to organize flexibly into a network for data collection and delivery. Building such a network introduces significant challenges, especially at the architectural, network protocol and application level because of limited memory, processing, and energy resources. To reach these goals, we have built our event-driven Real-Time Operating System (RTOS) named Ambient-RT [2] for embedded small computers. Despite these constraints, Ambient-RT has powerful features like, real-time scheduling and support for a modular data driven architecture.

The question that remains is how to build an architecture that supports adaptation and self-configuration. Our solution is to use component-based system model [3] for embedded devices. Component-based systems allow different software entities to come together to form an application either at compilation time or during run-time. On the former case, the focus is on the reusability of the software components to create new product populations [4]. The latter approach focuses on meeting real-time characteristics. One use case may be many components cooperating in a distributed manner to meet a demand; or another use case may be adapting software bindings during run-time, depending on the context information.

To support adaptation and self-configuration, we have split the architecture into cooperating entities. In this paper, we start by giving the requirements of how to design each entity or component, why that component is needed. We put our focus on resource management and resource representation which is the decision mechanism for adaptation and self-configuration.

Besides from component model, we use the Publish/Subscribe (P/S) model [5] for supporting heterogeneity and mobility of ubiquitous environment. Programming in

⁰This work has been partially funded by the projects EYES (IST-2001-34734), STW-PROGRESS Featherlight (TES.6388) and Smart Surroundings (<http://smart-surroundings.org>).

P/S environment is quite different from other component-based systems such as TinyOS [11]. TinyOS is a popular event-driven component-based operating system designed for resource-poor sensor nodes. Up to now, researchers have put many efforts on development using this platform. That is why, we do not want to rebuild every software and we want to use TinyOS software in our environment. The next question that we address is: "what are the compatibility problems between the popular TinyOS components and the new P/S based environment?". The problems include resource representation, emulating sequential program flow and message ownership with buffer exchange technique.

In Section III, we explain existing P/S models and our difference as well as how network abstractions are done in TinyOS environment. Section IV introduces the view and the required entities of the architecture with P/S functionally added. In Section V, we explain the basics of the TinyOS operating system. In Section VI, we discuss the compatibility issues, requirements and challenges when translating from the TinyOS applications to our environment. In section VII, we give the concluding remarks, and future research issues.

III. PUBLISH/SUBSCRIBE MODEL AND SOFTWARE ABSTRACTIONS

Most of the adaptive networking protocols try to export a set of parameters that tune the protocol behavior in a custom approach [6]. To achieve local (nodal) and global (network-wide) adaptability and energy efficiency, there is a need to define standard means of communication among software components.

In [7], authors mention about emerging networking abstractions. In their approach, certain Application Programming Interface (API) or abstractions define the means of interacting with the network. When a programmer wants to use the component that implements the abstraction, he or she wires this component to the application and uses it straightforwardly without caring about the content. Their software abstraction analysis shows that each time they try to introduce new hardware, they need to change the abstraction API for certain platform specific features. In addition, application layer abstractions are so specialized that it is not easy to replace one protocol with another.

Instead of defining certain static abstraction API, we use the flexibility offered by P/S systems. P/S systems allow full decoupling of software in space, time and synchronization [5]. In P/S based systems, data providers and subscribers do not know one another. Data providers and subscribers may publish and receive the same data at different places and times. Therefore, there is no need for synchronization. Unlike the static sensor network applications, ubiquitous computing system nodes are mobile and heterogeneous. Therefore, data publishers and subscribers may not exist in the network at the same time. A third-party software entity, named data broker or agent, caches the requests and delivers them when peers become available as in [8]. We use P/S method in Ambient-RT to achieve fine granularity. For example, if an application

wants to send a message, it publishes the message with certain networking parameters using "Routing" as the subject name. If there is any routing protocol component that has subscribed with the same subject name ("Routing"), data broker delivers the published message to the routing protocol. In response to this request, routing protocol publishes the result. This dynamic behavior allows us to replace one protocol with another at run-time and P/S remains the standard means of communication among software components. Any protocol can access the published data of any other protocol and can make certain inferences from the data for energy efficiency and adaptation.

We base our architecture on top of Publish/Subscribe model. It is a model of computation that we find suitable for networks of resource-poor nodes such as Wireless Sensor Networks (WSN) and WSN capable ubiquitous computing networks. Requests between mobile components are not addressed to a certain module. Instead, they are published with a name that identifies the functionality. Any software module, which has already subscribed to it, will process the request.

IV. ARCHITECTURE

In this research, we are mainly concentrating on the design of an architecture that supports adaptation and self-configuration. We extend our P/S model with these features. To meet these goals, we have split the architecture into cooperating entities. In this section, we are giving the requirements of how to design each entity and why it is needed. We put our focus on resource management which is the decision mechanism for adaptation and self-configuration.

A. Data Manager Entity

The data manager or broker is responsible for dissemination of published data in the network and inside the node.

B. Service Discovery Entity

The Service discovery entity is responsible for keeping track of resources (resource monitoring) in the network.

C. Resource Management Entity

There is a Resource Management Entity (RME) inside the operating system that reacts to the changes in the environment. The RME activates or tunes different protocols and interacts with service discovery for resources in other nodes. The RME in each node cooperates with the other nodes' RME's to form a global resource management framework represented in Figure 1. The fundamental design principles for a Resource Manager are as follow:

1) *Resource Allocation*: This is the basic task of the RME. Given a set of requests and resources, the RME should do the suitable resource allocation to the requests.

2) *Resource Representation*: Resource Representation is used to represent resources in terms of the services they provide and as well as their types in the system. This representation should be rich enough to allow the requesters to get the best matchmaking [9]. We give our initial method for resource representation in section VI-A in detail.

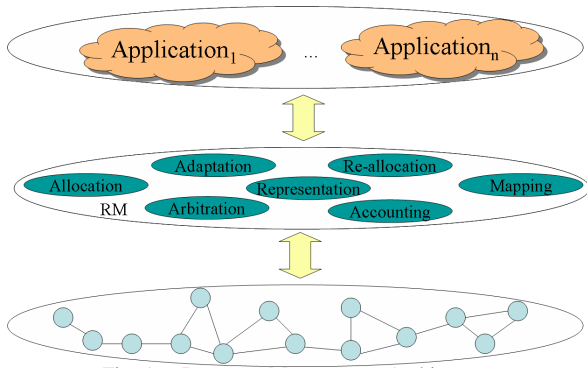


Fig. 1. Resource Management Architecture

3) *Mapping*: Resource mapping or matchmaking is the selection of the resources which the RME should consider when a specific request occurs. For example, given a complex request that needs many nodes to cooperate, the RME should be able to decide which nodes to activate.

4) *Adaptation*: Adaptation is the ability of the RME to adjust the application, node and network operation against dynamic, impulsive and unstable behavior of resources. For example, the networking conditions for ubiquitous computing environment are usually not easily predictable. Because of congestion, the bandwidth resource may vary. The RME should be able to adapt the system behavior, in case a change in the resource availability occurs.

5) *Arbitration*: Arbitration is making sure that, at a minimum, resources are not used beyond their capacities [9]. For example, the RME should be aware of the battery resource availability before taking an energy demanding action. For example if a node intends to route another node's packet, the RME entity should control the battery level before doing the energy-costly listen-receive-send action sequence.

6) *Reallocation*: Sometimes, to meet a specific resource demanding request, the RME can take away some or all the resources from an active request.

7) *Heterogeneity*: The classical resource management architectures usually concentrate on the heterogeneity of the resources. Although this much heterogeneity is not the case for WSN enabled ubiquitous systems, the types of the nodes in the network or roles of the nodes may vary. The nodes' available resources or their needs according to their role may vary from one node to another.

8) *Local Decisions vs. Global Behavior*: The RME should be aware of the actions that affect the global network behavior. For example, the nodes' energy usage affects the lifetime or survivability of the network [10].

V. TINYOS COMPONENT MODEL

TinyOS [11] is a popular component-based, event-driven operating system designed for resource-poor sensor-equipped small devices. It has attracted much attention by the researchers all around the world.

In a typical application scenario, a TinyOS programmer starts with a high-level design. In this design, the programmer writes the specifications of the components and the components' interactions among themselves. TinyOS gives the

```

module TimerM {
  provides interface Timer[uint8_t id];
  provides interface StdControl;
  uses {
    interface Leds;
    interface Clock;
    interface PowerManagement;
  }
}
implementation {
  uint32_t mState;
  ...
  command result_t StdControl.init() {
    ...
  }
  ...
  async event result_t Clock.fire() {
    ...
  }
}

```

Fig. 2. An Example TinyOS Module

configuration and *module* names to the software component. Thus, we use them interchangeably to refer to a software component.

The specifications of module interactions translate into TinyOS *Interfaces*. It includes a set of function names, parameters and their directions (command or event). Interfaces are bidirectional communication channels between two modules or components. Functions named as *commands* form one direction of the channel, and *events* form the opposite direction. Commands are used to invoke requests and events are the notification messages sent to the caller as a response.

A module is usually dependent on certain interfaces. It needs certain functionality from other modules and the programmer declares these definitions at the beginning of the module file using *provides* and *uses* primitives. Functions that any module neither uses, nor provides are internal to the module and they contribute to building of the module itself.

Figure 2 gives an example code listing from the TimerM module. It needs an application, which wants to use this module, to provide Leds, Clock and PowerManagement interfaces to TimerM module. Similarly, the TimerM module provides Timers and StdControl interfaces to the application.

After building the module, the programmer needs to connect the provided and used interfaces of all modules to one another to reach to the complete application using *configuration* files. It is also possible for a configuration file to provide or use interfaces. So, configurations serve as containers of multiple modules. The programmer may bind many configuration files to one another to build higher level configurations for sophisticated applications.

In the most basic form, the programmer connects each provided interface of each module to a used interface of another module or configuration. In the code listing shown in Figure 3 SenseToInt module's TimerControl Interface is connected to TimerC configuration's TimerControl Interface.

VI. COMPATIBILITY ISSUES

In this section, we give the compatibility issues between our environment and TinyOS components. Currently, we are using our simulation framework ¹ environment for this study. We

¹<http://wwwes.cs.utwente.nl/ewnsim/>

```

configuration SenseToRfm {
}
implementation
{
  components Main, SenseToInt, IntToRfm, TimerC
  components DemoSensorC as Sensor;

  Main.StdControl -> SenseToInt;
  Main.StdControl -> IntToRfm;

  SenseToInt.Timer -> TimerC.Timer[unique("Timer")];
  SenseToInt.TimerControl -> TimerC;
  SenseToInt.ADC -> Sensor;
  SenseToInt.ADCControl -> Sensor;
  SenseToInt.IntOutput -> IntToRfm;
}

```

Fig. 3. An Example TinyOS Configuration

have identified the following problems during code generation for the environment.

A. Resource Representation

TinyOS supports a flexible interface representation during the software configuration. A programmer can give an interface name of his or her choice as in Figure 3. We use "given name" and "flexible interface name" definitions interchangeably. Here, "component name . interface name" representation is used to connect the interface of the component to another component's interface. In the example, SenseToInt component's TimerControl and ADCControl interfaces are of the same type (StdControl). TimerControl name is given to the interface that controls the Timer component and ADCControl name is given to the interface that controls the Sensing unit.

One problem with the flexible interface representation is that when we move from TinyOS component model to our P/S model: the name, a publisher module uses for publishing data, may not match a subscriber's subscribed name. A publisher may be using Timer.Timercontrol; whereas, a subscriber can use Timer.StdControl. In existing P/S systems published data name needs to be unique and well-defined before programming. It is not possible to search for an interface name because functionalities provided by the interfaces may differ.

For example, a simple application such as Sense (Figure 3) in the TinyOS repository uses multiple instances of the StdControl Interface to control different units of the system. In this code listing, these interfaces are named as TimerControl and ADCControl using the flexible interface representation although they both provide the same interface for different purposes. SenseToInt module uses the TimerControl Interface to configure Timer and the ADCControl Interface to control the sensing unit (for example: a temperature or a photo sensor).

As the publisher does not have any idea about to which interfaces other modules in the system subscribed, it will send a message using standard interface name StdControl. Let's assume that every component in the system subscribed to the standard interface name. If a subscriber needs service from multiple instances of the same interface and when it receives the published message, to which interface to deliver the message remains a fuzzy problem.

We studied two different approaches to solve this problem.

1) *Standard Interface Name Representation*: To avoid confusion we followed the following approach: if a module

provides multiple instances of the same interface, the module subscribes to the name provided by the programmer rather than the standard interface name. We assume that a subscriber knows (as it is the case in TinyOS) which interface name to publish data. Although this approach fixes the problem, it affects component generalization adversely. A publisher needs to know which interface name subscriber subscribed and whether the subscriber has many interfaces of the same type. Therefore, the publisher needs to know about the subscriber.

This approach was acceptable for a while. We were able to run CntToInt, CnttoIntRfmM applications using this algorithm. The former application sets a timer to fire regularly and the latter one will broadcast a network message with the counter value at each Timer fire event. These applications show that most of the important system components or configurations such as GenericComm (network communication interface including MAC, routing and physical layer components) and Timer interfaces are able to run using new environment.

Nevertheless, we had compatibility problems when we tried to run applications that use the sensing interface. Tracing and debugging showed that our approach was inappropriate because the flexible interface representation gives the programmer the freedom to name his or her interfaces.

2) *Subject Based Interface Name Representation*: Using standard interface names do not solve all the problems. Interfaces are just contracts between the sender and the receiver for calling conventions. However, any component that exports such an interface may have different functionality inside.

We argue that the use of flexible name representation is more code readability concerned. A programmer normally uses the flexible interface representation when a component provides or uses multiple instances of the same interface to distinguish one from another. On the other hand, in current implementation of Timer, a single interface of type StdControl is named as TimerControl and during the configuration it is possible to make the bindings shown in Figure 4.

It is confusing to follow which interface is connected to which interface. During compilation time with testing of the existing interfaces for different names, it is possible to find the matching interface. Nevertheless, if we try to find it during run-time, it is impossible to find the correct one unless we specify a standard calling convention for the interfaces.

As expressed before, one main concern of this research is to enable code reuse for the run-time systems to provide real-time guarantees. Therefore, we needed to solve this problem for existing TinyOS components. As we cannot port each software manually, we needed to find a way to automate translation for

```

Main.TimerControl -> Timer.StdControl
Main.TimerControl -> Timer
Main.TimerControl -> Timer.TimerControl
Main.StdControl -> Timer.StdControl
Main.StdControl -> Timer
Main.StdControl -> Timer.TimerControl
StdControl -> Main.TimerControl
StdControl -> Main.StdControl
TimerControl -> Main.StdControl
TimerControl -> Main.TimerControl

```

Fig. 4. Ambiguous Configuration

```

configuration InjectMsg{
    provides interface ReceiveMsg as RadioReceiveMsg;
    provides interface ReceiveMsg as UARTReceiveMsg;
}
implementation
{
    components Nido;

    RadioReceiveMsg = Nido.RadioReceiveMsg;
    UARTReceiveMsg = Nido.UARTReceiveMsg;
}

```

Fig. 5. InjectMsg Configuration

software.

During translation, we enforce components to the following policy:

- 1) If a component provides a single instance of an interface, it has to subscribe to the component's standard interface name rather than the given name.
- 2) Similar to Aspect Oriented Programming (AOP) [12]; we require a publisher and a subscriber to give a subject name to the functionality type they provide and request (for example: MAC, Routing). They negotiate the subjects before processing a request.

Let's examine the Inject module of TinyOS simulator TOSSIM in Figure 5. TOSSIM uses the Inject module to inject network messages into the simulation environment either for Radio or Universal Asynchronous Receiver-Transmitter (UART) delivery to the simulated node.

Both of the interfaces are of the same type and named as RadioReceiveMsg and UARTReceiveMsg respectively. The InjectMsg module subscribes to the standard interface name of InjectMsg.ReceiveMsg.Receive ("Module Name + Standard Interface Name + Function Name" format).

When it receives a message, it directs the request to the relevant interface according to the subject. Any publisher that wants to use the radio interface must use RadioReceiveMsg as a subject. Similarly, any publisher that wants to use the UART interface must use UARTReceiveMsg as a subject.

B. Emulating Sequential Program Flow

In many imperative programming languages, a program flow starts at the main function and after a series of function calls the program ends. Although TinyOS is a component-based operating system, the compilation or binding of components results in a sequential program code in C language. Despite TinyOS events are asynchronous by nature, command calls are synchronous. As a result, the TinyOS operating system blocks a command caller until the result of the command call is retrieved. Afterwards, the caller can change the path of the program flow depending on the result as if it is a function call.

To enable code reuse for our run-time system, we need to model sequential program flow for commands using RPC like schemes. Additionally, we need to support bidirectional function calls for our component-based systems. A sophisticated command call may set a parameter of the caller before returning a result as in Figure 6. As A, which starts the sequence, needs the result of the function call and B makes another call to A, A should be able to process the incoming request while in blocked condition. A can spawn another task

to serve incoming requests. This may seem like a suitable solution but having too many tasks in the system will put significant loads on the CPU and memory. Besides, the system will suffer from synchronization problems.

We solved the problem by using a better scheduling policy for serving the request. In this policy the operating system blocks the caller while queuing execution requests. If a component receives a message in the blocked condition, the component will check whether it is a response to the previous blocking function call or it is a request to execute another function. If it is the former case, component continues its normal program flow. If it is the latter case, the component serves the request using the program stack of the blocked function call. Therefore, blockers will be ordered according to their execution requests in the stack. Once the result for the last execution request is retrieved, each blocking function call is removed one by one from the stack. Nevertheless, this policy might have some problems with stack overflows. Additionally, to cope with synchronization problems, components serve one request at a time while queuing other requests. In order for one component to serve multiple requests from different components, this algorithm needs to be extended using multiple stacks.

C. Buffer Exchange in Event-Driven Systems

One common technique in networking protocols is to exchange buffers among one another. The purpose of this exchange is to avoid copying and saving memory space. Generally, a message structure is passed onto a higher level protocol after advancing the pointers. However, our study showed that in an event-driven system, this approach is not suitable. There is a need to define clearly which component owns which message.

To clarify the problem, suppose that a message is received from the physical layer and we have multiple protocols that need to use this information. In an event-driven system each component will be sent a notification of the event along with the buffer.

For example, the RME and Routing protocols are interested in physical and MAC layer parameters of incoming messages. The RME would like to check certain characteristics of the messages such as received signal strength, CRC or other MAC

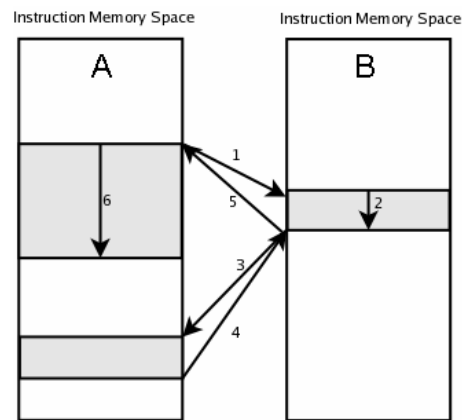


Fig. 6. Sophisticated RPC Call

related parameters; whereas, routing layer would like to make decisions on forwarding. In a buffer exchanging system, a physical layer entity will first deliver the message to the RME and gather an empty buffer for incoming new messages. Although the physical layer entity does not own the message anymore, it will also do the same for the Routing entity. Therefore, there will be multiple software entities working on the same buffer. Thus, it will lead to an inconsistency in the system.

As explained above, for the sake of the system, the operating system (OS) has to take some other approach. We have identified the following options.

- 1) The OS sends a separate copy to each component although the operation is CPU and memory intensive.
- 2) The OS schedules each task to access a resource in such a way that they do not overlap for read or write access [13].
- 3) Before gaining access to a variable, a task has to negotiate for the access type. If every task wants to gain access to the variable read only - instead of copying - OS keeps a single copy and grants the access by passing pointers. The problem with this approach is to propagate access rights among components. It is especially difficult when the requesting entity is some other node accessing via the network. A node needs to keep track of neighbors for variable access. This approach is too energy consuming.

Although it is not efficient, for the ease of implementation we have decided to separate message ownership and make copies at the current implementation. At the receiving end, every component is responsible for its own message buffers. When we make the translation to Ambient-RT, we will use the built-in scheduling policy as in option two.

VII. CONCLUSION

This work is an initiative attempt toward component-based software engineering in ubiquitous computing and wireless sensor network environments while maintaining code reusability. After identifying the system components and their requirements, we implemented a code generation tool and experimented with TinyOS components in our P/S based simulation environment. The next step is to build the resource management support at each node and at global scale.

At the current implementation, we have borrowed interface definitions from TinyOS where they are hardcoded and do not change during the lifetime of the application until the node is reprogrammed. We want to have systems that remain in use as long as possible (in untethered mode). Therefore, we need to query the interfaces in the system to know which ones are available as in COM [14] since the nodes in the environment can be heterogeneous and may not have the interfaces that a node requires.

Our approach is a subject based P/S mechanism. We expect that we need formal techniques to avoid programming errors due to subject selection. We also expect that simple syntactic interfaces are not sufficient and that we will need to take functional behavior into account, such as QoS parameters and

synchronization behavior [15]. Finally, we expect the need for contract-based design [16] to guarantee stability of the system.

REFERENCES

- [1] M. Weiser, "The Computer for the Twenty-First Century," *Scientific American*, pp. 94–10, Sept. 1991.
- [2] T. Hofmeijer, S. Dulman, P. G. Jansen, and P. J. M. Havinga, "AmbientRT - real time system software support for data centric sensor networks," in *2nd Int. Conf. on Intelligent Sensors, Sensor Networks and Information Processing*. Melbourne, Australia: IEEE Computer Society, Washington, DC, Dec 2004, pp. 61–66.
- [3] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins, "A survey of configurable, component-based operating systems for embedded applications," *IEEE Micro*, vol. 21, no. 3, pp. 54–68, 2001.
- [4] R. van Ommering, "Building product populations with software components," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM Press, 2002, pp. 255–265.
- [5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.
- [6] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *SensSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, pp. 95–107.
- [7] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The emergence of networking abstractions and techniques in tinyOS," in *First Symposium on networked system design and implementation (NSDI04)*, San Francisco, California, USA, 2004, pp. 1–14.
- [8] G. Cugola and H.-A. Jacobsen, "Using publish/subscribe middleware for mobile systems," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 4, pp. 25–33, 2002.
- [9] K. Gajos, L. Weisman, and H. Shrobe, "Design principles for resource management systems for intelligent spaces," in *Proceedings of The Second International Workshop on Self-Adaptive Software*, Budapest, Hungary, 2001, to appear.
- [10] G. Mainland, D. C. Parkes, and M. Welsh, "Decentralized adaptive resource allocation for sensor networks," in *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.
- [11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *ASPLOS*, 2000, pp. 93–104.
- [12] E. Hilsdale, J. Hugunin, M. Kersten, G. Kiczales, and J. Palm, "Aspect-oriented programming in java with aspectj?" March 2001.
- [13] P. G. Jansen, S. J. Mullender, P. J. M. Havinga, and J. Scholten, "Lightweight EDF scheduling with deadline inheritance," Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, Technical report TR-CTIT-03-23, May 2003.
- [14] Microsoft, "DCOM technical overview," Microsoft Corporation, Microsoft Windows NT Server White Paper, 1996.
- [15] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorga, F. Long, J. Robert, R. Seacord, and K. Wallnaau, "Technical concepts of component-based software engineering," CMU, SEI TR-008, 2000.
- [16] D. K. Hammer and M. Chaudron, "Component-based software engineering for resource-constraint systems: What are the needs?" in *Proc. 6th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, Rome, 2001.
- [17] F. Luders, K.-K. Lau, and S.-M. Ho, "Specification of software components," M. L. Ivica Crnkovic, Ed. ARTECH HOUSE BOOKS, 2002.