Adaptive System Software Support for Cooperating Objects

Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel

IPVS, Universität Stuttgart
Universitätsstr. 38
D-70569 Stuttgart, Germany
{marron, gauger, lachenmann, minder, saukh, rothermel}@informatik.uni-stuttgart.de

Abstract. Efficient system software support is essential for cooperating object applications in order to cope with the complexity and heterogeneity of typical scenarios in this domain. In this paper, we argue that adaptation capabilities should be an integral part of such system software and present the TinyCubus framework as one possible solution that provides the features required of system software for cooperating objects.

1 Introduction

Ubiquitous computing and sensor/actuator network applications are mainly characterized by their need to interact heavily with their environment. Cooperation among the embedded systems used to support these types of applications is the only way to perform complex tasks with the limited capabilities of each individual node. The *cooperating objects* model is an abstraction that represents systems that combine sensors, controllers and actuators to perform a variety of tasks autonomically in a distributed environment. In this paper we present the cooperating objects model and argue for the need of adaptive system software support for its implementation in ubiquitous computing-type environments.

The remainder of this paper is structured as follows. In the next section we explain the concept of cooperating objects in more detail and present our arguments for the need of adaptive system software support. In section 3 we briefly present TinyCubus, a flexible and adaptive cross-layer framework for TinyOS-based sensor networks that provides mechanisms supporting the implementation of cooperating objects. Finally, section 4 concludes this paper.

2 Cooperating Objects

2.1 Definition

Following the definition provided by the Embedded WiSeNTs consortium [1], a **cooperating object** is a collection of *sensors*, *controllers (information processors)*, *actuators* and other *cooperating objects*. The individual components of a

cooperating object communicate with each other in order to perform a common task in a more or less autonomic way.

Sensors are devices that act as suppliers of input to the cooperating objects and are able to gather and retrieve information either from other cooperating objects or from the environment they are immersed in.

Controllers act as data or information processors and cooperate with *sensors* and *actuators* for interacting with their environment. *Controllers* are also equipped with a storage device that allows them to use previously collected data for performing their tasks. The amount of "effort" a particular controller devotes to either information processing or storage tasks is determined individually. This way, the cooperating object network might be composed of controllers that provide information processing capabilities, whereas others might specialize in storing data efficiently.

Actuators are devices that produce output and are able to actively interact with and modify their environment using, for example, some kind of electromechanical device.

Being able to include other cooperating objects as part of the definition of a cooperating object allows the system to combine *sensors*, *controllers* and *actuators* hierarchically in arbitrarily complex structures. At the same time, the definition does not force the three entities to be realized as physically independent devices. In fact, in the case of classical sensor networks, *actuators* are often relegated to a second layer whereas sensors and controllers are put together in a single device.

Obviously, if *sensors*, *controllers* and *actuators* need to interact with each other in a distributed environment, each device needs to be equipped with communication capabilities. Depending on the type of cooperating object network, the communication might be based on wired or wireless technology.

To illustrate the definition of cooperating objects, imagine a cooperating object being used for collecting temperature gradients of flammable liquid within an industrial plant. When the gradient achieves certain pre-defined thresholds, safety pipe valves must be opened to minimize the risks of an explosion. In this scenario, we have two cooperating objects: one that continuously measures temperatures and another one that actuates in the environment by manipulating valves. The first one is an example of a classical sensor network with embedded controllers, whereas the second one would be traditionally described as an "actuators and controllers network".

2.2 The Need for Adaptive System Software

Cooperating objects need the support of system software that takes care of basic functionality such as communication, event handling and event generation, as well as the scheduling of the installed components. This simplifies the work of the application developer and, depending on the specific solution, this basic functionality can be provided either directly by the operating system or with the help of a middleware solution.

While this kind of support is already provided by many established system software implementations, we argue below that another critical factor, namely the ability to adapt the application and system software to varying environmental conditions and changing user preferences, is not well addressed by prevalent middleware and operating system solutions for cooperating objects.

Important properties of the environment of cooperating objects might change over time. In some cases, this necessitates adapting the software of the sensors, controllers or actuators to accommodate for these changes. One can imagine scenarios where it is difficult or impossible to include code appropriate for all possible environmental conditions. It might not even be desirable to do so as the program memory of embedded systems is typically extremely constrained.

Analogous to potential changes in the environment of the cooperating objects, the user preferences might also change over time. One example is a user deciding to analyze a certain subset of the sensor data in more detail. Again, it is difficult to accomodate for all such changes in advance.

In many cases, it is not possible to have an a priori description of the detailed topology of wireless sensor and actuator networks, since it is often desirable to deploy the nodes of the network in an ad-hoc manner. Moreover, existing topologies can change at any time due to the mobility of nodes, changing communication links or failing nodes. This makes it extremely difficult to program the cooperating objects with software suitable for all possible situations. This is aggravated by the heterogeneity expected for typical cooperating objects applications where not only the communication topology but also the availability of individual types of objects changes with time.

To cope with changing environmental conditions, changing user preferences and changing network topologies and to ease the development of applications for cooperating objects, support by adaptive system software is necessary. It should support the *reconfiguration* of applications, also providing mechanisms for the *parametrization of generic components* in order to adapt them to specific application requirements. If this does not suffice, an efficient mechanism to install new *application-specific components* must be available on the cooperating objects so that an explicit initiation of reconfigurations does not need to be triggered externally. Instead, the system software should provide mechanisms for *automatic adaptation*, for example, based on a predefined set of rules.

Cooperating objects operate in a heterogeneous environment and new applications and hardware platforms continuously evolve. For this reason, the system software has to be *extensible* and *flexible* to cope with new platform and application requirements.

Finally, applications react differently to changes in their environment, e.g., changes in the mobility of nodes. They also differ concerning their optimization parameters, for example whether they favor low latency or low energy consumption. The system software must be able to *adapt* the system operation to these requirements and support the optimization requests of the applications, especially because of the resource limitations found in many cooperating objects.

One possible approach is to perform cross-layer optimizations thereby allowing components to interact closely.

3 TinyCubus

`TinyCubus`, a flexible and adaptive cross-layer framework for TinyOS-based sensor networks, aims to support the main building blocks of an adaptive system software for cooperating objects as elaborated in the previous section. Specifically, these are flexible reconfiguration, optimization and adaptation capabilities.

`TinyCubus` consists of a data management framework, a cross-layer framework, and a configuration engine. The *data management framework* allows the dynamic selection and adaptation of system and data management components. The *cross-layer framework* supports data sharing and other forms of interaction between components in order to achieve cross-layer optimizations. The *configuration engine* allows code to be distributed reliably and efficiently by taking into account the topology of sensors and their assigned functionality.

The overall architecture of `TinyCubus` mirrors the requirements imposed by the applications and the underlying hardware. As shown in figure 1, `TinyCubus` is implemented on top of TinyOS [2] using the nesC programming language [3], which allows for the definition of components that contain functionality and algorithms. We use TinyOS primarily as a hardware abstraction layer. For TinyOS, `TinyCubus` is the only application running in the system. All other applications register their requirements and components with `TinyCubus` and are executed by the framework.

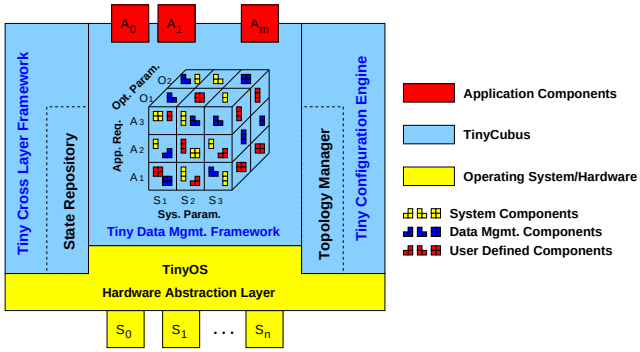
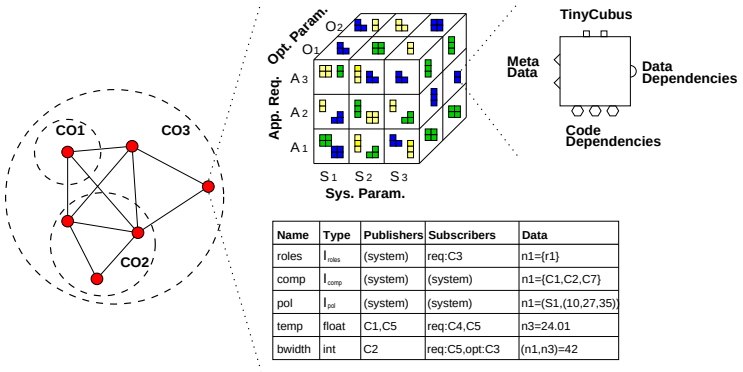


Fig. 1. Architectural components in `TinyCubus`

Figure 2 illustrates the architecture of a cooperating objects model using `TinyCubus`. The left part shows a network of cooperating objects whereas the right part of the figure illustrates the operation of `TinyCubus` on one example node. The individual parts shown in the right part are explained in the description of the architecture in the following paragraphs.



Name	Type	Publishers	Subscribers	Data
roles	I_{roles}	(system)	req:C3	n1={r1}
comp	I_{comp}	(system)	(system)	n1={C1,C2,C7}
pol	I_{pol}	(system)	(system)	n1={S1,(10,27,35)}
temp	float	C1,C5	req:C4,C5	n3=24.01
bwidth	int	C2	req:C5,opt:C3	(n1,n3)=42

Fig. 2. Architecture of the Cooperating Objects Model

3.1 Tiny Data Management Framework

The Tiny Data Management Framework provides a set of data management and system components, selected on the basis of the typically data-driven nature of sensor network applications. For each type of standard data management component such as replication/caching, prefetching/hoarding, aggregation, as well as each type of system component, such as time synchronization and broadcast strategies, it is expected that several implementations of each component type exist. The Tiny Data Management Framework is then responsible for the selection of the appropriate implementation based on the information obtained from the system.

The cube shown in figures 1 and 2, called 'Cubus', combines *optimization parameters*, such as energy, communication latency, and bandwidth; *application requirements*, such as reliability; and *system parameters*, such as mobility. For each component type, algorithms are classified according to these three dimensions. For example, a tree based routing algorithm is energy-efficient, but cannot be used in highly mobile scenarios with high reliability requirements. The component implementing the algorithm is tagged with the combination of parameters and requirements for which the algorithm is most efficient. Eventually, for each combination a component will be available for each type of data management and system components.

The Tiny Data Management Framework selects the best suited set of components based on current system parameters, application requirements, and optimization parameters. This adaptation has to be performed throughout the lifetime of the system and is a crucial part of the optimization process.

3.2 Tiny Cross-Layer Framework

The Tiny Cross-Layer Framework provides a generic interface to support parameterization of components using cross-layer interactions. Strict layering (i.e.,

each layer only interacts with its immediately neighboring layers) is not practical for wireless sensor networks [4] because it might not be possible to apply certain desirable optimizations. For example, if some of the application components as well as the link layer component need information about the network neighborhood, this information can be gathered by one of the components in the system and provided to all others.

If layers or components interact with each other, there is the danger of loosing desirable architectural properties such as modularity. Therefore, in our architecture the cross-layer framework acts as a mediator between components. Cross-layer data is not directly accessed from other components but stored in a state repository. The lower right part of figure 2 shows an example of such a state repository with data identified by its name and stored together with its type, a list of providers and a list of subscribers.

Other examples of cross-layer interactions are callbacks to higher-level functions, such as the one provided by the application developer. TinyOS already provides some support with its separation of interfaces from implementing components. However, the TinyOS concept for callbacks is not sophisticated enough for our purposes, since the wiring of components is static. With `TinyCubus`, components are selected dynamically and can be exchanged at runtime. Therefore, both the usage of a component and callbacks cannot be static; they have to be directed to the new component if the data management framework selects a different component or the configuration engine installs a replacement for it. `TinyCubus` extends the functionality provided by TinyOS to allow for the dereferencing and resolution of interfaces and components.

3.3 Tiny Configuration Engine

In some cases, parameterization, as provided by the Tiny Cross-Layer Framework, is not enough. Installing new components, or swapping certain functions is necessary, for example, when new functionality such as a new processing or aggregation function for the sensed data is required by the application. The Tiny Configuration Engine addresses this problem by distributing and installing code in the network. Its goal is to support the configuration of both system and application components with the assistance of the topology manager.

The topology manager is responsible for the self-configuration of the network and the assignment of specific roles to each node. A role defines the function of a node based on properties such as hardware capabilities, network neighborhood, location etc. A generic specification language and a distributed and efficient role assignment algorithm is used to assign roles to the nodes.

Since in most cases the network is heterogeneous, the assignment of roles to nodes is extremely important: only those nodes that actually need a component have to receive and install it. This information can be used by the configuration engine, for example, to distribute code efficiently in the network.

4 Conclusion

Given the complexity and heterogeneity of cooperating object applications, let them be just sensor network applications or more complex sensor-actor systems, there is a clear need for support by the underlying middleware or system software. In this paper, we have argued that the ability to adapt to changing environmental conditions, user preferences and network topologies should be an integral part of such system software. Furthermore, we have described the `TinyCubus` framework, one possible architectural and algorithmic solution to the requirements imposed by cooperating objects environments.

Regarding future work, there is a need to provide a clear classification of cooperating object applications in general and sensor network applications in particular that will allow us to better show the applicability of our model to a wide variety of application domains. In addition, the use of actuators will eventually need the modeling of real-time applications and control-loops that fall a little short on our current view of component dependencies and are, therefore, hard to model using the described architecture.

References

1. Embedded WiSeNts: Embedded WiSeNts - Project FP6-004400 (`http://www.embedded-wisents.org/`)
2. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. (2000) 93–104
3. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation. (2003) 1–11
4. Goldsmith, A.J., Wicker, S.B.: Design challenges for energy-constrained ad hoc wireless networks. IEEE Wireless Communications **9** (2002) 8–27