Generic Model and Architecture for Cooperating Objects in Sensor Network Environments

Pedro José Marrón, Daniel Minder, Andreas Lachenmann, Olga Saukh and Kurt Rothermel
University of Stuttgart, Institute of Parallel and Distributed Systems (IPVS)
Universitätsstr. 38, D-70569 Stuttgart, Germany
{marron|minder|lachenmann|saukh|rothermel}@informatik.uni-stuttgart.de

***Abstract*— The complexity and heterogeneity of cooperating object applications in ubiquitous environments or of applications in the sensor network domain require the use of generic models and architectures. These architectures should provide support for the following three key issues: flexible installation, management and reconfiguration of components in the system; optimization strategies whose implementation usually involves the proper management of cross-layer information; and proper adaptation techniques that allow for the self-configuration of nodes and components in the system with minimal human intervention. In this paper, we present one possible instance of such a generic model and architecture and show its applicability using Sustainable Bridges, a sensor network application that requires the analysis of complex sensor data to achieve its goal of effectively monitoring bridges for the detection of structural defects.**

I. Introduction

The continuous miniaturization process of computing devices combined with the proliferation of sensor networks, have led to an increase on the number of devices that are able to sense their environment, process it and communicate their results. The cooperation and coordination tasks of applications running in such environments present the application and system developer with new challenges that need to be resolved [1].

In the `Embedded WiSeNTs` project [2], twelve European universities have joined forces to study environments composed of a large number of cooperating objects that interact with each other to accomplish a common task. These objects might be composed primarily of sensors, building the traditional sensor networks found in the literature, be embedded in their surroundings, in what is usually called pervasive or ubiquitous environments, or be inmersed in a combination of both worlds. The difference in mentality of these communities has lead to the development of two types of approaches: **data-centric** and **service-centric** solutions.

In general, **data-centric** approaches are chosen in environments where the naming of data and the use of data types within the network play a more important role than the specific device that might be responsible for its processing. Therefore, there is a dissociation of data and network device which can be used to dynamically select the appropriate location where data processing is performed. Therefore, **data-centric** approaches are best suited for database-like operations like aggregation and data dissemination.

In the literature, there are two different kinds of **data-centric** processing techniques. The first one uses the query/response (or request/reply) paradigm, so that the network of cooperating objects only sends responses to specific queries issued by the user [3]. The second technique assumes that queries are "stored" in the network and are provided with an associated lifetime. During their lifetime, each device is responsible for the processing of the stored query and sends messages to the issuer of the query (also called sink) whenever the condition specified in the query is met [4]. Therefore, both pull-based and push-based approaches can be used in data-centric environments.

Although the absolute position of devices within the network do not play an important role from the perspective of the issuer of the query, good topology management techniques need to be used in order to maximize the lifetime of individual devices.

In contrast, **service-centric** approaches are mostly concerned with the definition of the interface or *API* in order to provide certain functionality for the user. Depending on the specific fields there are other additional characteristics that need to be mentioned. For example, in the field of pervasive computing, the miniaturization of devices as well as resource-limitation play an important role, whereas in classic client-server architectures no such restrictions apply.

In such environments, the transport mechanisms are hidden from the user applications (such as in traditional networked environments), but a certain cooperation among the nodes in the network allows for the processing of data. The difference to **data-centric** approaches lies in the kind of programming techniques needed to interact with the network. In a **service-centric** environment, the application developer is supposed to have and use a clear specification of services offered by the network.

The complexity that arises from the interaction of computing devices in such settings have led the researchers in the `Embedded WiSeNTs` project to define cooperating objects in such a way that the breadth of challenges (and hopefully some solutions) can be easily inferred. For this purpose, we propose a generic model and architecture that can be used in arbitrary environments where cooperating objects interact.

The goal of this paper is, therefore, three-fold: (1) Provide a more formal definition of cooperating objects; (2) identify the key characteristics that software developed for cooperating

objects needs to have; (3) provide a generic network model and object architecture that would allow for the easy development and deployment of software in sensor network environments.

The remainder of this paper is structured as follows: Section II provides a definition of cooperating object and derives some requirements for a generic model and architecture. Section III explains our network model that defines possible interactions among cooperating objects. The specifics of the object architecture are left for section IV, whereas section V provides an example usage of our model using the Sustainable Bridges application. Finally, section VI gives some insight on related work and section VII concludes this paper and discusses future work.

II. Definition of Cooperating Objects

As already specified in some internal documents of the `Embedded WiSeNTs` project, a **cooperating object** is a collection of:

- *sensors*,
- *controllers (information processors)*,
- *actuators* or
- *cooperating objects*

that communicate with each other in order to accomplish a common task in a more or less autonomic way.

More precisely, *sensors* are devices that act as inputs to the cooperating objects and are able to gather and retrieve information either from other cooperating objects or from the environment they are inmersed in.

Controllers are devices that act as data or information processors and cooperate with *sensors* and *actuators* in order to be able to interact with their environment. Furthermore, *controllers* are equipped with a storage device that allows them to perform their tasks. The amount of "effort" devoted by a particular controller to either information processing or storage tasks is determined on an individual basis. 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 act as output producers and are able to interact and modify their environment using, for example, some kind of electromechanical device.

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

Finally, the inclusion of other *cooperating objects* as part of the definition of cooperating object itself indicates that these objects can combine their *sensors*, *controllers* and *actuators* in a hierarchical way and are, therefore, able to create arbitrarily complex structures.

In order to illustrate this definition more precisely, imagine that a cooperating object is used to collect 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".

For the specific implementation of cooperating objects, there is nothing in the definition above that forces all three entities (*sensors*, *controllers* and *actuators*) to be physically independent devices. In fact, in the case of sensor networks, where the primary focus is set on gathering data from the environment and not so much on acting on it, *actuators* are usually relegated to a second plane and sensors and controllers are put together in a single device. Therefore, hardware for sensor networks usually looks like the MICA family of Fig. 1, where the integration of sensing devices and controllers is done on a single board.

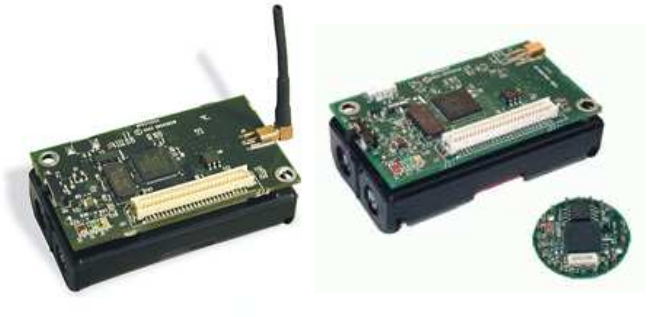


Fig. 1. MICA Family from Crossbow Technology Inc.

Cooperating objects need certain system software that takes care of basic functionality such as communication, event handling and generation, as well as the scheduling of the installed components. For the purposes of this paper, we adopt the definition of **component** used in TinyOS [5], the standard system software found on the MICA-family of Fig. 1 and extend it to fit our needs. In TinyOS, components are modular pieces of software that, by means of interface specifications, can be wired together to implement a complex application. Components offer and require certain functionality and are able to generate or handle events. In our generic architecture (proposed in section III), we assume the presence of adaptation components which control the installed components in the system based on cross-layer information such as roles.

A **role** defines the function of a node based on properties such as hardware capabilities, network neighborhood, location etc. The types of cooperating objects defined above (sensors, controllers and actuators) are some examples of role assignments. Other examples for roles are `SOURCE`, `AGGREGATOR`, and `SINK` for aggregation applications, `CLUSTERHEAD`, `GATEWAY`, and `SLAVE` for clustering applications. In previous work [6] we describe a generic specification language and an algorithm for efficient role assignment.

Requirements for a Generic Model and Architecture

Using the definition we have just described, it seems clear that in typical cooperating object applications, the network itself, that is, the collection of cooperating objects involved in solving the problem at hand, is heterogeneous. An application developer will have to deal with sensors, controllers, actuators, etc. and probably will need to deal with the complexity of having hybrid network topologies, where some of the cooperating objects interact with each other using wireless technology, whereas others might be connected to an infrastructure.

Moreover, the applications themselves are heterogeneous [7], [8], so that their requirements change drastically from one another. In some cases, due to the fact that applications are installed for extended periods of time, their requirements might change over time and the system software needs to be quickly adapted to the new application requirements.

Finally, depending on the environment where the application is deployed, the system itself might change rapidly. Parameters like mobility, network density, etc. play a crucial role for the selection of the appropriate algorithm to solve efficiently the task at hand, but these parameters are, under some environments, highly dynamic.

To ease the development of sensor network applications, a generic framework is, therefore, necessary. Such a framework has to support the *data-centric model* of sensor network applications and their need for *reconfiguration* and flexibility. However, sensor networks are heterogeneous and new applications and hardware platforms continuously evolve. Thus, a generic framework has to be *extensible* and *flexible* to manage new application requirements. It should provide mechanisms for the *parametrization of generic components* so that they can meet the requirements of specific applications. If this is not sufficient, new *application-specific components* have to be installed on the sensor nodes. The code of these new components has to be distributed efficiently in the network to avoid wasting energy.

Finally, applications react differently to changes in their environment, e.g., changes in the mobility of nodes. They also have different optimization parameters, e.g., energy or latency. The framework must then be able to *adapt* to these conditions and support optimizations, especially because of the resource limitations found in sensor networks. One approach is to perform cross-layer optimizations by allowing components to interact closely.

Therefore, in order to provide a generic model and architecture, we need to provide: a network model, that describes a collection of cooperating objects and interactions among them, and an object architecture that describes the internal characteristics of each device that composes each cooperating object and allows itself to *configure* its components, provide *cross-layer optimizations* and *adapt* to changes in its environment.

III. Network Model

For the description of the network and its components as defined in section II, our network model is best described as a tuple $\mathcal{M} = (\mathcal{G}, \mathcal{F}_\mathcal{N}, \mathcal{I}_\mathcal{N}, \mathcal{F}_\mathcal{E}, \mathcal{I}_\mathcal{E}, \mathcal{P})$, where:

- $\mathcal{G} = (N, E)$ is a communication graph that represents the physical connectivity of devices in the network in the usual way;
- $\mathcal{F}_\mathcal{N}$ is the set of functions that define and map the properties of each node in $\mathcal{G}$;
- $\mathcal{I}_\mathcal{N}$ is the set of domains for all functions $F_i \in \mathcal{F}_\mathcal{N}$;
- $\mathcal{F}_\mathcal{E}$ is the set of functions that define and map the properties of each communication link in $\mathcal{G}$;
- $\mathcal{I}_\mathcal{E}$ is the set of domains for all functions $F_j \in \mathcal{F}_\mathcal{E}$; and
- $\mathcal{P}$ is the set of primitives that represent emergent properties of the network.

A **basic cooperating object** is a graph consisting of only one physical device (node) $n_i \in N$ and no communication links. It is defined as: $\mathcal{C} = (\{n_i\}, \emptyset)$, where n_i is of type *sensor*, *controller*, or *actuator*.

A subgraph $\mathcal{C} = (N', E')$ of $\mathcal{G}$ with $N' \subseteq N$ and $E' = \{(a, b) \in E : a, b \in N'\}$ is said to be a **cooperating object** if $\mathcal{C}$ is connected, that is, $\forall N_1, N_2 \subset N'$ with $N_1, N_2 \neq \emptyset, N_1 \cup N_2 = N', N_1 \cap N_2 = \emptyset : \exists (a, b) \in E' : (a \in N_1 \wedge b \in N_2) \vee (a \in N_2 \wedge b \in N_1)$. Note that $\mathcal{C}$ always contains all existing communication links of $\mathcal{G}$ for all nodes N'. Figure 2 shows an example of a network with several cooperating objects.

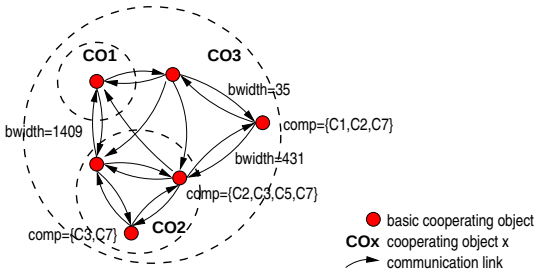


Fig. 2. Sample Network Model

$\mathcal{F}_\mathcal{N}$ is a set of (possibly multivalued) functions of the form $F_i : N \times N \rightarrow I_i$, where $F_i \in \mathcal{F}_\mathcal{N}$, N is the set of nodes of $\mathcal{G}$, and $I_i \in \mathcal{I}_\mathcal{N}$ is the domain of function F_i. Analogously, $\mathcal{F}_\mathcal{E}$ is a set of (possibly multivalued) functions of the form $F_j : N \times E \rightarrow I_j$, where $F_j \in \mathcal{F}_\mathcal{E}$, E is the set of communication links of $\mathcal{G}$, and $I_j \in \mathcal{I}_\mathcal{E}$ is the domain of function F_j. The first argument of the functions denotes the node the information is stored on, and the second argument denotes the entity the information is about.

Finally, let $\mathcal{P}$ be a set of primitives $P_i \in \mathcal{P}$ that define properties of basic cooperating objects and their links as a whole. Typical primitives are defined using logic expressions, as shown in the examples below.

The specific set of functions defined in each case depend greatly on the application, but there are standard functions both in $\mathcal{F}_\mathcal{N}$ and $\mathcal{F}_\mathcal{E}$ that need to be defined by all applications:

- $F_{roles} : N \times N \rightarrow I_{roles}$ is a multivalued function that assigns one or more roles to each node in the network. The type of a physical device as defined above, is then simply a specific role assignment to a node. The membership to a cooperating object can be expressed as a role, too.

- $F_{comp} : N \times N \rightarrow I_{comp}$ is a multivalued function that assigns one or more components to each node. The set of components defines the functionality of the node and, therefore, the type of work it can perform. Fig. 2 shows several cooperating objects that store information about components. For example, the basic cooperating object furthest to the right has components C_1, C_2 and C_7 installed.
- $F_{data} : N \times N \rightarrow I_{data}$ is a multivalued function that assigns one or more data items to each node in the network. This information is maintained and updated by each algorithm in order to provide a means for exchanging information among components in a standard way.
- $F_{pol} : N \times N \rightarrow I_{pol}$ is a multivalued function that assigns one or more policies to each node. These policies are used for adaptation purposes, as explained in the next section.
- $F_{bwidth} : N \times E \rightarrow I_{bwidth}$ is a function that assigns the link capacity to each edge in the network. For example, in Fig. 2, the basic cooperating object furthest to the right has a communication link with bandwidth 431 to CO_2.

As an example of a primitive $P \in \mathcal{P}$, consider the definition of $P_{role\_conn}(r, k, n_i)$ as the set of basic cooperating objects with a given role $r \in I_{role}$ found in at most k-hops from basic cooperating object n_i. See [9] for a formal definition.

Primitives can also be used to obtain information about the composition of a cooperating object. For examples, the primitive $P_{BCO}(co)$ determines the set of nodes that belong to a cooperating object co, and $P_{FC}(co, F_i)$ applies function $F_i \in (\mathcal{F}_{\mathcal{N}} \cup \mathcal{F}_{\mathcal{E}})$ to all nodes $n \in N$ that belong to cooperating object co.

Applying the definitions described in this model, it is possible to obtain a "global view" of the network and to know what is installed in each cooperating object, what kind of objects are found in the network and how they operate with each other. Depending on the specific location where data, algorithms and policies are stored and executed, it is possible to define different processing techniques using the same formalisms. For example, the processing of data centralized in single controller, or distributed among several controllers and/or cooperating objects can be specified in our model by storing the values of certain functions in its corresponding location.

IV. Object Architecture

In order to support the generic requirements described in section II (flexible reconfiguration, optimization and adaptation capabilities), as well as to fit the model defined in the previous section, we need support from the internal configuration of the different cooperating objects available in the network. For this purpose, our proposed architecture, which we call `TinyCubus` [9], is composed of three parts: the Tiny Cross-Layer Framework, the Tiny Configuration Engine and the Tiny Data Management Framework.

A. Tiny Cross-Layer Framework

The Tiny Cross-Layer Framework provides a generic interface to support the parametrization of components that use cross-layer interactions. As described in [10], strict layering is not practical for wireless sensor networks, and thus for cooperating objects, because certain optimizations might not be applicable. Therefore, the purpose of this framework is to manage a copy of cross-layer information among cooperating objects in a *state repository*.

This *state repository* allows for the clean separation of the data itself and the components that publish or subscribe to it. Using the more formal definition of previous sections, the *state repository* physically stores some values of functions in $\mathcal{F}_{\mathcal{N}} \cup \mathcal{F}_{\mathcal{E}}$, so that they can be used by other cooperating objects.

Name	Type	Publishers	Subscribers	Data
$roles$	I_{roles}	(system)	req:C_3	$n_1 \rightarrow \{r_1\}$ $n_2 \rightarrow \{r_1, r_2\}$
$comp$	I_{comp}	(system)	(system)	$n_1 \rightarrow \{C_1, C_2, C_7\}$
pol	I_{pol}	(system)	(system)	$n_1 \rightarrow$ $(S_1, (10, 27, 35))$
$temp$	$float$	C_1,C_5	req:C_4,C_5	$n_3 \rightarrow 24.01$
$bwidth$	int	C_2	req:C_5 opt:C_3	$(n_1, n_3) \rightarrow 42$

TABLE I
SAMPLE STATE REPOSITORY OF NODE n_1

Table I shows the contents of a sample state repository where cross-layer information is kept. The system keeps information about the *name* of the data item, its *type*, a list of *publishers* of each data item, a list of optional and required *subscribers* to it, and the value of the function itself. Required subscribers are components that cannot properly function if the data item they are subscribed to is no longer available, whereas optional subscribers might benefit from a particular data item, but do not need it.

Finally, the *state repository* also stores some derived information, such as topology data, neighboring cooperating objects, etc. that belong to the set of primitives $\mathcal{P}$ defined above as part of the network model.

B. Tiny Configuration Engine

In some cases the separation of code and data as provided by the Tiny Cross-Layer Framework might not be enough for some applications. Installing new components, or swapping certain functions is necessary, for example, when new functionality such as a new processing or aggregation function for 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 arbitrary 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 (F_{roles} in our model). It also publishes topology information using the state repository that describes

the neighborhood of cooperating objects, the status of communication links and the availability of certain components in other neighboring nodes.

Additionally, the configuration engine needs to provide enough capabilities for the efficient reconfiguration of a cooperating object, which involves the implementation of bootstrapping code and the ability to load and install components on the fly.

The configuration engine may benefit from cross-layer information such as the specific roles available in the network to provide more efficient implementations of code distribution algorithms and component installation techniques, as shown in [9].

C. Tiny Data Management Framework

The Tiny Data Management Framework is an Adaptation Framework that also provides a set of data management and system components. 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 current information contained in the system.

The cube of Fig. 3, called 'Cubus', combines optimization parameters ($O_1, O_2, \ldots$), such as energy, communication latency and bandwidth; application requirements ($A_1, A_2, \ldots$), such as reliability or consistency level; and system parameters ($S_1, S_2, \ldots$), such as mobility or node density. 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.

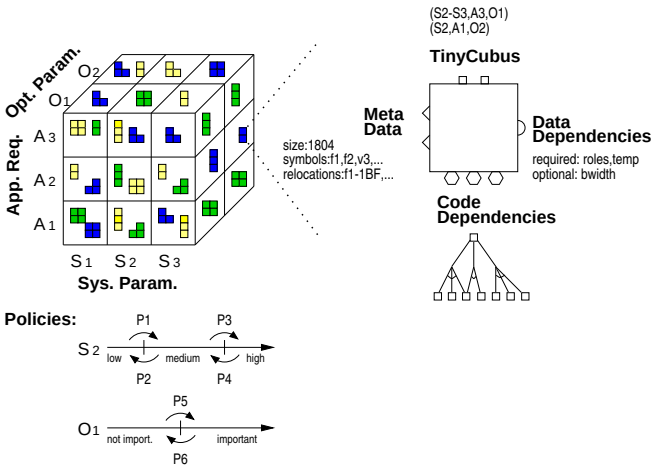


Fig. 3. Sample Adaptation Engine

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.

In order to accomplish this optimization process, we need two different parts: the Adaptation Framework itself, shown on the left-hand side of Fig. 3, and the set of components it manages, shown on the right-hand side.

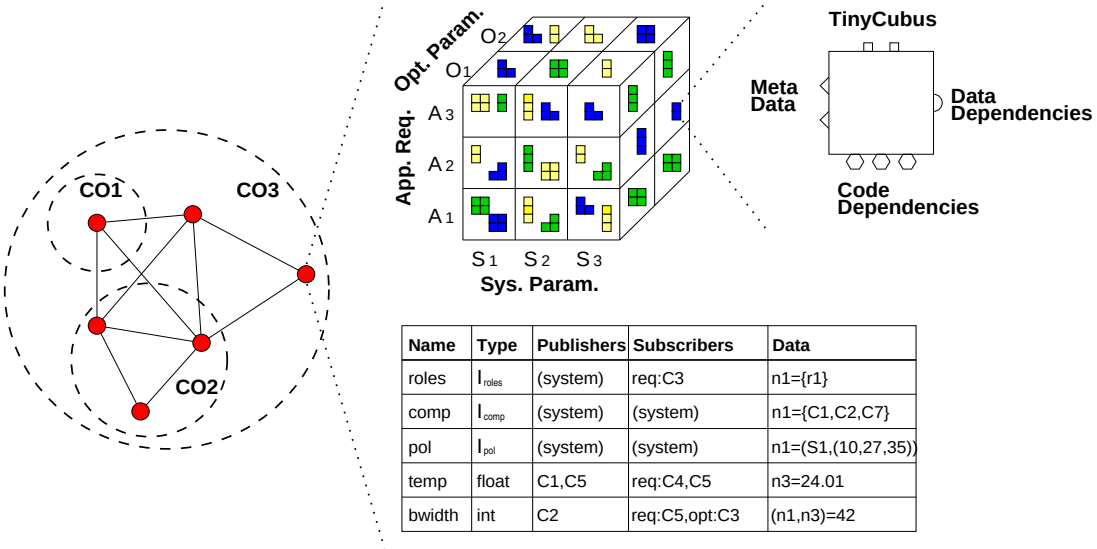
The adaptation framework contains three entities: a set of parameters, a set of policies and the adaptation components themselves. The set of parameters is used to provide a classification of the available components and, as depicted in Fig. 3, form a three-dimensional space (cube) where components can be mapped to. This mapping is performed using experimental evaluation of each component in combination with the appropriate parameters. This way, we know which components and/or combination of components perform best for a given *system parameter*, *optimization parameter* or *application requirements* parameter.

The second entity found in the adaptation framework, the available policies, are used to adapt and exchange components. These policies are rules with certain threshold values that indicate the operations that need to be performed to trigger changes in the configuration of objects and, therefore, in the set of components installed in a cooperating object. Fig. 3 shows two different kinds of policies for parameters S_2 and O_1. For S_2, there are policies P_1, P_2, P_3 and P_4 that specify the position of two threshold values. These thresholds define three areas ("low", "medium" and "high") and the different policies specify the set of operations that need to be performed with parameter S_2 changes from one area to the next. For example, if S_2 was "low" and is now "medium", the operations defined in P_1 are executed.

Finally, the third entity found in the adaptation framework are the system components that implement the policies and parameter checks needed to accomplish adaptation. For some of the defined parameters, this implies the necessity of having a *system monitor* that checks certain parameters (some of which are stored in the state repository) at regular intervals so as to trigger the right adaptation policy when needed.

The right-hand side of Fig. 3 shows the interfaces that need to be specified by the components that are available for adaptation. These components are obviously the most important part of the adaptation framework since they are the ones that provide the functionality, algorithms, etc. that need to be adapted. The following pieces of information need to be provided to the adaptation framework by each component that wants to be adaptable: a set of code dependencies, a set of data dependencies, a set of meta-data items and a mapping to the adaptation Cubus.

The first element, the set of code dependencies is specified best by a set of interfaces and a dependency graph. As shown in Fig. 3, this defines a graph of dependent components that also need to be installed, uninstalled, modified, etc. if the component is adapted. Of course, the dependencies to other components in the system can be extracted automatically by



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. 4. Architecture of the Cooperating Object Model

a compiler and stored as part of the component definition.

Secondly, the set of data dependencies indicates which pieces of data provided by other components are needed by the component. For example, Fig. 3 shows that the component on the right-hand side requires information about the roles available in the network, temperature values and bandwidth information. As for the set of code dependencies, data dependencies can be extracted at compile-time by analyzing the code in the component and at run-time by looking at the subscription information contained in the state repository of the Tiny Cross-Layer Framework.

Third, the set of meta-data items describes the internal properties of the component, such as its code size, the names and types of symbols contained in the component and a relocation table that is needed to place the component at arbitrary locations within a cooperating object. This information is needed because if components need to be installed, uninstalled, etc., the adaptation framework needs to be able to relocate them dynamically based on the current set of installed components.

Finally, there is some information that needs to be provided by the component regarding the mapping to the adaptation framework. These are data items such as the specifics of the classification within the three dimensions of the Cubus, threshold values for policies and changes, and even certain policies that need to be taken into account by the adaptation components.

Note that the entities described in the object architecture and the concepts presented as part of the network model are tightly coupled. Fig. 4 shows the relationship between the concepts of section III and the architecture described in this section. In this picture, we can see our network of cooperating objects on the left with 6 basic cooperating objects and three compound ones. The right-most device has certain information stored in it: the adaptation components with the right set of parameters and components needed for the proper functioning of the device, and a series of cross-layer data provided by some of the available components.

V. Sample Application: Sustainable Bridges

Let us now use an example to describe how the model and architecture would look like for a specific application: Sustainable Bridges.

The goal of the Sustainable Bridges project [11] is to provide cost-effective monitoring of bridges using static sensor nodes in order to detect structural defects as soon as they appear. A wide range of sensor data is needed to achieve this goal, e.g., temperature, relative humidity, vibrations characteristics, as well as noise detection and localization mechanisms to determine the position of cracks. In order to perform this localization, nodes sample noise emitted by the bridge at a rate of 40 kHz and, by using triangulation methods, the position of the possible defect is determined. This process requires the clocks of adjacent sensors to be synchronized within 60 μs of each other. Finally, sensors are required to have a lifetime of at least 3 years so that batteries can be replaced during the regularly scheduled bridge inspections.

Fig. 5 shows the topology of the network and the different cooperating objects needed to monitor the bridge. CO_1 and CO_2 are responsible for the monitoring of the columns of the bridge and contain several devices that cooperate with each other to reach consensus about sensor information. CO_3 and CO_4 are responsible for the monitoring of the bridge "edges", and all other devices ensure the connectivity of the network.

In each of these devices and cooperating objects, and based on the description of the project given above, we need the following five major components (more thoroughly described in [12]): cluster management, event localization, time synchronization, data aggregation and acoustic emission analysis components.

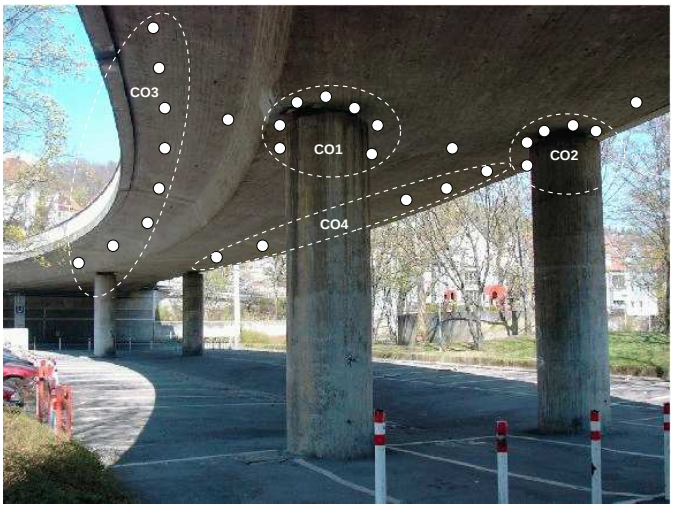


Fig. 5. Idealized Network Structure of the Sustainable Bridges Application

The *Cluster management* component for sensor data fusion is needed by each of the cooperating objects of Fig. 5 that monitor critical parts of the bridge. In this setting, each basic cooperating object (or device) is responsible for the sensing of potential structural defects using acoustic emission analysis and, if a potential problem is detected, each device communicates with its cluster head to find out whether or not other devices in the same cooperating object have also reached the same conclusion. If so, a possible defect event is generated and propagated through the network.

The *Event localization* component determines using triangulation mechanisms and acoustic emission data from the bridge, the position of cracks and defects on the structure. For this component, usage of clustering information is critical so that the triangulation mechanisms achieve a degree of accuracy (within a couple of meters) that would allow a person to know the location of a possible defect.

The *Time synchronization* component allows for the comparison of complex time series (acoustic emission waves) gathered by the acoustic emission sensors. Unless the different devices and cooperating objects are synchronized with each other, the same event detected by several sensors independently cannot be correctly compared since the acoustic waves are shifted on the time axis. Furthermore, higher time synchronization is required within cooperating objects, whereas this requirement is not so crucial for cooperating objects further apart.

The *Data aggregation* component is able to summarize data retrieved by the sensors in the bridge on-the-fly using the topological information stored in each device about the network.

The *Acoustic emission analysis* components work with different degrees of accuracy and complexity on the acoustic waves produced by the sensors in order to determine the presence, magnitude and complexity of potential structural defects on the bridge.

These components store some cross-layer information in the state repository of each cooperating object. This data is either produced or consumed by one or more of the components we have just described. Besides generic information needed by all applications such as the *roles* of each device (for example cluster heads), the *topology and routing information* that defines the connectivity of the network, and *node vital information* such as the battery level or link reliability, the Sustainable Bridges application has the following application-specific data:

Acoustic emission data that identifies the time, magnitude and characteristics of potential cracks detected in the structure, as well as history of past detections. *Temporal data* such as the current time, accuracy of the last synchronization round and time to the next wave synchronization. *Dependencies among components* based on subscriptions to data. For example, the topology information needs to be used by the acoustic emission analysis component to find out the set of neighbors it needs to contact in order to analyze a possible defect. The data aggregation component needs information about roles, the topology of the network and time synchronization data in order to be able to compare different acoustic waves.

Finally, the adaptation engine contains information such as the *dependencies* among components, and *policies* (with their corresponding *threshold values*) needed to determine when it is necessary to perform a certain type of analysis. Based on this information, the adaptation framework might decide that certain low-cost low-accuracy analysis can be performed at the sensor itself, whereas if a certain threshold is reached, more complex analysis might need to be performed at the cluster head or at a central computer located outside the network. Since acoustic emission waves are too complex to be sent efficiently to the central computer for analysis, it is more desirable to trigger the installation of the right analysis component at the location of the bridge that needs it.

VI. Related Work

SensorWare [13] and Impala [14] aim at providing functionality to distribute new applications in sensor networks. For this purpose, they create abstractions between the operating system and the application, although both differ slightly from each other. SensorWare does not support adaptation and cross-layer interactions, as it is the case in our generic architecture and does not provide models of the network.

In Impala, new code is only transmitted on demand if there is a new version available on a neighboring node. Furthermore, if certain parameters change and an adaptation rule is satisfied, the system can switch to another protocol. However, this adaptation mechanism only supports simple adaptation rules. Although it uses cross-layer data, Impala does not have a generic, structured mechanism to share it and so, is not easily extensible.

The MobileMan project [15] is a system that aims at creating a cross-layer architecture similar to ours. However, MobileMan is not targeted towards sensor networks and assumes environments typical of mobile ad-hoc networks, which are, in the general case, not so limited in terms of resources. In

addition, MobileMan focuses on data sharing between layers of the network protocol stack and, therefore, does not include the configuration and adaptation capabilities found in our architecture.

EmStar [16] is a software environment for Linux-based sensor nodes that, like MobileMan, assumes the presence of higher-end nodes as part of the sensor network. EmStar also contains some standard components for routing, time synchronization, etc., but it is not able to provide the adaptation mechanisms available in our architecture.

Finally, regarding the modeling of cooperating object networks, it is worth mentioning that there is no available literature that attempts to combine a network model with a local architecture that supports it. For example, [17], [18], [19] only deal with models for the simulation of sensor networks and obviate the need for models that also incorporate actuators and more general cooperating objects. Although [20], [21] try to provide taxonomies and models for more generic sensor networks, they only consider the modeling of network characteristics and never consider the actual characteristics of the nodes themselves or the type and amount of software (components) installed in each system which, for cooperating objects in general and sensor networks in particular, plays a crucial role, since resource-limitation is one of the intrinsic characteristics of such systems.

VII. Conclusion and Future Work

Given the complexity of the definition of cooperating objects and the heterogeneity of cooperating object applications, let them be just sensor network applications or more complex sensor-actor systems, there is a need for a generic model and architecture that allows us to tackle the complexity of such systems. In this paper, we have presented such a generic model and architecture that can be used for cooperating object applications in service-oriented environments as well as in data-centric environments, such as sensor networks. Furthermore, we have shown the integral parts of our proposed model and architecture by using Sustainable Bridges, a complex sensor network application.

In terms of 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.

VIII. Acknowledgements

This work has been partially supported by the European Union as part of the 6^{th} Framework Programme Project: `Embedded WiSeNTs` – FP6-004400.

References

[1] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: Scalable coordination in sensor networks," in *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*. ACM Press, 1999, pp. 263–270.

[2] Embedded WiSeNts - Project FP6-004400. [Online]. Available: http://www.embedded-wisents.org/

[3] J. Gehrke and S. Madden, "Query processing in sensor networks," *Pervasive Computing*, vol. 3, no. 1, pp. 46–55, 2004.

[4] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A tiny aggregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, 2002.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104.

[6] K. Römer, C. Frank, P. J. Marrón, and C. Becker, "Generic role assignment for wireless sensor networks," in *Proc of the 11th ACM SIGOPS European Workshop*, 2004, pp. 7–12.

[7] J. Kahn, R. Katz, and K. Pister, "Emerging challenges: Mobile networking for smart dust," *Journal of Communications and Networks*, vol. 2, no. 3, pp. 188–196, September 2000.

[8] K. Römer and F. Mattern, "The design space of wireless sensor networks," *IEEE Wireless Communications*, vol. 11, no. 6, pp. 54–61, December 2004.

[9] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel, "TinyCubus: A flexible and adaptive framework for sensor networks," in *Proc. of the 2nd European Workshop on Wireless Sensor Networks*, E. Çayırcı, Şebnem Baydere, and P. Havinga, Eds., Istanbul, Turkey, January 2005, pp. 278–289.

[10] A. J. Goldsmith and S. B. Wicker, "Design challenges for energy-constrained ad hoc wireless networks," *IEEE Wireless Communications*, vol. 9, no. 4, pp. 8–27, 2002.

[11] Sustainable bridges website. [Online]. Available: http://www.sustainablebridges.net

[12] P. J. Marrón, O. Saukh, M. Krüger, and C. Grosse, "Sensor network issues in the sustainable bridges project," in *2nd European Workshop on Wireless Sensor Networks (European Project Session)*, Istanbul, Turkey, January 2005.

[13] A. Boulis, C.-C. Han, and M. B. Srivastava, "Design and implementation of a framework for efficient and programmable sensor networks," in *Proc. of the 1st Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys 2003)*, 2003.

[14] T. Liu and M. Martonosi, "Impala: A middleware system for managing autonomic, parallel sensor systems," in *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2003, pp. 107–118.

[15] M. Conti, G. Maselli, G. Turi, and S. Giodano, "Cross-layering in mobile ad hoc network design," *IEEE Computer*, vol. 37, no. 2, pp. 48–51, 2004.

[16] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "EmStar: A software environment for developing and deploying wireless sensor networks," in *Proc. of USENIX 2004*, 2004, pp. 283–296.

[17] M. Varshney and R. Bagrodia, "Detailed models for sensor network simulations and their impact on network performance," in *MSWIM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*. ACM Press, 2004, pp. 70–77.

[18] S. Park, A. Savvides, and M. B. Srivastava, "SensorSim: A simulation framework for sensor networks," in *MSWIM '00: Proc. of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*. ACM Press, 2000, pp. 104–111.

[19] ——, "Simulating networks of wireless sensors," in *WSC '01: Proceedings of the 33nd conference on Winter simulation*. IEEE Computer Society, 2001, pp. 1330–1338.

[20] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, "A taxonomy of wireless micro-sensor network models," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 2, pp. 28–36, 2002.

[21] B. A. Kadrovach and G. B. Lamont, "A particle swarm model for swarm-based networked sensor systems," in *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. ACM Press, 2002, pp. 918–924.