Embedded WiSeNts

IST-004400

System Architectures and Programming Models

EW Identifier:	EW-D314-INRIA-001-07-SysArch
Date:	2005-10-13
Author(s) and company:	S. Santini and K. Roemer (Swiss Federal Institute of Technology Zurich, Switzerland), P. Couderc (Institut National de Recherche en Informatique et en Automatique, France), P. Marron and D. Minder (University Stuttgart, Germany), T. Voigt (Swedish Insitute of Computer Science, Sweden), A. Vitaletti (Consorzio Interuniversitario Nazionale per l'Informatica, Italy)
Work package/task:	WP 3 / T3.1.4
Document status:	Final
Confidentiality:	Public
Keywords:	Programming Models, System Architectures, sensor networks, Cooperative objects
Abstract:	This document discusses the structure and the topics of the study regarding the programming models for sensor networks and Cooperating Objects. It motivates the importance of programming abstractions for the success of these technologies, and presents the existing approaches in use that will be surveyed in the study.

Table of Contents

1 Executive Summary

This study provides a survey about the current state of the art of programming models and system architecture for Cooperating Objects and motivates their importance for a successful development of these technologies. Section 2 provides a brief introduction to the topics and motivates the need of designing suitable programming abstractions for Cooperating Objects. In Section 3, the most relevant existing programming abstractions are surveyed and classified. The main reason for the development of these abstractions is to allow a programmer to design applications in terms of global goals and to specify interactions between high level entities (such as *agents* or *roles*), instead of explicit managing the cooperation between individual sensors, devices or services. For example, the database abstraction allows to consider a whole sensors network as a logical database, and performing network-wide queries over the set of sensors. The various paradigm are surveyed and a set of criterions allowing to easily review their strengths and weaknesses are presented.
Section 4 presents the existing system architectures for Cooperating Objects at two different levels: first, system architectures of individual nodes, which includes the structure of the operating system running at node level and its facilities; second, system architectures supporting the cooperation of different nodes, such as communication models.
Finally, in section 5, the document points out some of the limitations of current approaches, and proposes some research perspectives. In particular, programming paradigms should provide more support to ease of programming, heterogeneity, as well as scalability issues. Regarding system architectures, real-time aspects, which are not currently well addressed, will become increasingly important for Cooperating Objects. Dynamic maintenance (such as code deployment and runtime update support) is another important issue to address in future systems. At last, an effort is required to better integrate the various paradigms and systems into a unified framework.

2 Introduction

Key to the successful and widespread deployment of cooperating objects and sensor network technologies is the provision of appropriate programming abstractions and the establishment of efficient system architectures able to deal with the complexity of such systems. Programming abstractions shield the programmer from the "nasty system details" and allow the developer to think in terms of the concrete application problem rather than in terms of the system. This is also true for traditional distributed systems, where numerous software frameworks and middleware architectures are crucial to perform an integrated computing task. Such frameworks and middleware are based on programming models such as distributed objects or events. These conventional and successful programming abstractions for distributed systems can, however, not be simply applied to Cooperating Objects and sensor networks, due to some substantial differences existing among the latter and the former systems.
We will refer to a *programming model* as "a set of abstractions and paradigms designed to support the use of computing, communication and sensing resources in an application" and to a *system architecture* as "the structure and organization of a computing system, as a set of functional modules and their interactions".

The notion of *Cooperating Objects* refers to devices ranging from sensors and smart tags to personal computing devices such as cell phones, PDAs, or even digital cameras. Within the Embedded WiSeNts project a *Cooperating Object* is formally defined as "a collection of:

- sensors,
- controllers (information processors),
- actuators or
- Cooperating Objects

that communicate with each other and are able to achieve, more or less autonomically, a common goal". For the purpose to achieve a given global goal, cooperating objects can organize themselves in a particular set up and, like in traditional distributed systems, coordinate and cooperate in order to perform a form of distributed computing. However, traditional distributed computing applications show many important differences from those considered in the context of cooperating objects and sensor networks: Cooperating Objects are in fact closely related to real world objects and/or to the physical space, which neither are considered in traditional distributed computing. This close integration with the real world raises several issues for the design of feasible architectures and programming abstractions for Cooperating Objects. Since real world objects/entities may be mobile, resource availability and context of operation change often. The system must be able to accommodate dynamic (re-)configuration, as well as being able to expose to the programmer context changes. The latter point is an aspect which differentiates Cooperating Objects from mobile computing: mobile computing approaches often try to provide the impression of a standard (static) computing environment, in order to shield the programmer from mobility constraints. In addition, typical application domains of Cooperating Objects are different from those of distributed and mobile computing: in ubiquitous and pervasive computing, applications often have to deal with the notion of *context*, which is directly related to data about the physical environment and does not appear as an issue in distributed

computing. Another aspect that differentiate Cooperating Objects and in particular sensor networks to traditional distributed system, is that a service or function may often be provided anonymously by one (or several) sensor(s): the identity of the device providing the function is thus often unimportant.
Because of these differences, the design and development of dedicated system architectures and programming models appears as a key issue to enable the success of Cooperating Objects as a mainstream technology. Therefore, we will in the next sections provide an analysis of the requirements with which feasible system architectures and programming models must be able to comply, we will survey the most significative existing approaches and evaluate them, underlining the issues that in our opinion still require attention from the research community.

3 Programming Models

In order to design feasible abstractions and paradigms, a first issue to address is which objects will be used in the computing environment. While entities like a simple sensor or a complex cellular phone share the common properties of a Cooperating Object, it is unclear whether common programming models would apply to such different entities. In the following section we will provide a brief overview of the most significative common attributes of Cooperating Objects and underline the requirements that these attributes pose on programming models. In Section 3.1 we will then survey the state of the art, providing examples and references for further readings. Finally, in Section 3.3, a summary of the existing approaches will be provided and the issues requiring further research will be pointed out.

3.1 Requirements

Cooperating Objects and sensor network applications pose specific requirements on the design of programming abstractions and paradigms. Fundamental work has been done already to identify general significative characteristics of Cooperating Objects and to derive from them some common requirements with which suitable programming models have to comply [ECPS02], [ASSC02].
Considering the existing work and our experience in this domain, we conclude that an adequate programming abstraction for Cooperating Objects must essentially take into consideration the following aspects:

1. **Ease of Programming and Expressiveness.** Future applications scenarios for sensor networks and Cooperating Objects typically envisage developers that are not necessarily expert programmers, as for example biologists, supply-chain managers or hospital operators. Moreover, since networks of Cooperating Objects are envisioned to include hundreds or thousands of single devices, an application developer should not be bound to program Cooperating Objects individually. On the contrary, should be provided with the ability to specify a *global task* for the network, letting to the underlying system layers the task of translating this global goal in concrete actions the single devices have to carry out.
   On the other hand it is also extremely important to keep as high as possible the level of expressiveness of the system "programming language", by providing the programmer with a set of operators and instructions which must be large and diverse enough to enable him/her to exploit the whole capabilities and functionalities the network is endowed with.
   On the application level this requires the programmer being somehow aware of the *global features*

the system can offer and being provided with adequate *programming primitives* to access these features without necessarily knowing neither the characteristics of the system nor the way a specific task is achieved by assigning concrete roles to the single devices. In an environmental monitoring application, for example, a programmer may probably prefer to issue a query like *"Detect all bats in the cave"* rather than *"Detect and discriminate all ultrasonic-pulses coming from the area delimited by the given reference points"*. In this case, the phenomena the programmer is interested in should be reinterpreted in measurable quantities and detected by multiple cooperating entities. in this case, the programmer will not need to be aware of this "translation" process, because of the higher level primitives he/she is provided with.
With regard to usability and expressiveness aspects, we can conclude that a good programming abstractions for Cooperating Objects must:

- Provide "easy-to-use" and expressive programming primitives
- Allow the developer to program the system as a whole (global task specification)

2. **System Diversity.** We defined a Cooperating Object as "a collection of sensors, controllers (information processors), actuators or Cooperating Objects that communicate with each other and are able to achieve, more or less autonomically, a common goal". Thus, a Cooperating Object may be as simple as a single sensor endowed with some computing and communication capabilities or as complex as a sophisticated mobile unit equipped with a wide set of sensors and actuators, a powerful processor and some kind of long lasting power supply. An application developer is, however, not interested in knowing the concrete ability of each single device the system consists of, but only on the results she could gain from using the system as a whole. In this sense the heterogeneity of the system should be as much as possible hidden from the programmer, allowing her to define a *global network task* and eventually letting the underlying system layers or even the single devices to cope with the translation of this global tasks into *single device tasks*. It is worth to note that the aspects single system devices may differ, include, among others: number and the type of sensors, computing power, communication range, communication technology, type and durability of power supply, packaging and physical dimensions.
   An adequate programming abstraction must thus be able to cope with heterogenous devices and should, in particular:

   - Furnish a set of programming primitives for task assignment independently of single devices capabilities

3. **System Dynamic.** When deploying a system composed of wireless devices potentially unevenly distributed over a wide geographical area, keeping track of the exact state of the system in terms of topology, lifetime, availability and connectivity, appears as a challenging task. In fact, not only the initial state of the system may be partially unknown, but due to the extreme dynamics the system may experience, its state may strongly vary in both space and time domains. Due to node mobility or environmental factors, for example, the topology of the network may change over time in an undetermistic manner, thus bringing uncertainty on network coverage and connectivity. In a sensor network single nodes may run out of power, be temporarily or permanently unavailable for unforeseen reasons (environmental factors, conscious or unconscious human interaction) or additional nodes may be redeployed to replace crashed ones. Since an application programmer would and should not track

or control the ever changing state of the system, it is a role of the run-time system to adequately adapt the assignment of network resources.
The abstraction designer is therefore compelled to consider that the system must:

- Be able to deal with unknown/unstable topology
- Be able to deal with unknown/unstable network size
- Be robust against temporarily or permanently device crashes
- Cope with unstable connectivity

At this point, we would like to point that system dynamic may also vary depending on the concrete application and/or deployment [1]. In application scenarios that include mobile Cooperating Objects, for example, the uncertainty about the topology of the system is typically much more significant than in some typical building monitoring scenarios, where the sensing devices may be fixed on walls or furniture.

4. **Environmental Dynamic.** Being embedded into the real world, sensing devices may experience extreme dynamics with respect to the phenomena being observed, both due to the unpredictable nature of the phenomena and due to their wide varying intensity and space-time extension. Even a single sensed quantity, for example, may span in a very wide interval and exhibit an extremely irregular behavior or, on the contrary, show no relevant changes for long periods of time and/or on wide geographical areas. The detection of real-world phenomena may also be a complex process, which may require the use of many different sensors and an aggregation of the measured values.
An application programmer is, however, presumably not interested in knowing how the occurrence of the phenomena can be detected, but just need to be able to correctly specify which phenomena she is interested in and eventually which actions to perform in consequence of their detection. The occurrence of a specific phenomenon represent thus for the application an *event*, to which the system should be able to react by either just reporting the measured variables or performing a more complex action. An abstraction designer must thus:

   - Provide programming primitives to address a wide variety of real-world *events*
   - Provide programming primitives to react to real-world *events*

5. **Resource Constraints.** We already pointed out that distinct Cooperating Objects may show a considerable diversity in terms of kind and number of usable sensors, available power supply and computing and communication capabilities. Therefore, the measure of which resource constraints influence the design of programming paradigms and abstractions for Cooperating Objects depends on the specific Objects being considered. When designing algorithms for sensor systems where nodes are typically powered with small batteries, have poor computing and storing capabilities, strict bounded communication range and a limited set of sensing and/or actuating devices, saving resources becomes a key factor for a feasible and successful design. As we already pointed out when discussing *System Diversity*, a suitable programming abstraction should be able to hide device heterogeneity to the programmer, and should thus also be able to:

[1]See also study 3.1.1, "Applications and Application Scenarios"

- Cope with power constraints
- Cope with computing constraints
- Cope with hardware constraints

6. **Scalability.** Since networks of Cooperating Objects may include hundreds or thousands of single devices, a suitable programming abstraction must be able to cope with network sizes that range from some units to thousands of devices.
   Depending on the application context and the geographical area that the network is deployed on, the density of devices may also vary. Scalability must thus be ensured also for varying (and possibly unknown) network density. For programming a network of Cooperating Objects, the developer must thus be provided with a programming model that is able to:

   - Ensure scalability to varying network size
   - Ensure scalability to varying network density

7. **Deployment and Maintenance.** Physical deployment and maintenance of networks of Cooperating Objects may have different challenges that depend on the concrete application scenario. However, since devices may be in most cases be physically inaccessible, hardware repairs and the number of software updates must be kept as low as possible, both because of the resulting transmission (thus, energy) costs and due to the problems related to ensuring update coherence throughout the network. An application developer will also need to be provided with suitable debugging tools, whose design and realization are issues of growing interest in the research community. Due to the manifold of devices a network of Cooperating Objects may be composed of and due to the extreme dynamics the system may experience, fault isolation appears as an extremely complex task and therefore reliable and resource efficient tools must be provided as a fundamental system feature.
   In order to guarantee efficient maintenance, a suitable programming model for cooperating objects must thus:

   - Limit the number of code updates
   - Support a resource-efficient application-level debugging

The complexity and diversity of Cooperating Objects pose strong constraints on the design and development of a suitable programming abstraction. Most of the approaches being investigated by the research community focus on specific application scenarios and are thus typically able to comply with only a subset of the above listed requirements. In order to give the reader an overview on the state of the art, we supply in Section 3.2 a brief survey on the ongoing work on programming models for Cooperating Objects.

3.2 State of the Art

In the last couple of years, a growing interest in the research area of Cooperating Objects brought to a number of designs for programming abstractions specifically targeted to these systems. We analyzed the research literature and ongoing work and came up to the conclusion that the nowadays most relevant existing programming abstractions for Cooperating Objects sensor networks can be classified in the following categories:

1. Database View
2. Event Detection
3. Virtual Markets
4. Virtual Machines
5. Mobile Code and Mobile Agents
6. Role-Based Abstraction
7. Group-Based Approach
8. Spatial Programming
9. Shared Information Space
10. Other Approaches (Service Discovery, Client-Server Approach, Distributed Objects)

In the following sections, we will analyse the main characteristics of the above listed general approaches and we will give a brief description of one or two representative sample-implementations for each given category.

3.2.1 Database View

A *database* may be commonly defined as a collection of data elements (facts) stored in a centralized or distributed memory in a systematic way, such that a computer program can automatically retrieve them to answer user-defined questions (queries). A system composed by a manifold of entities like simple sensor nodes, complex devices or just common everyday objects, each one of them endowed with sensing capabilities, may be regarded as a distributed database. In this system, stored data consist in sensor readings and where users can issue *SQL*-like queries to have the system performing a certain sensing task or delivering required data. In this perspective the system appears just as a collection of sensors, whose readings need to be adequately and automatically stored. Thus, a database approach abstracts away much of the complexity of a system collecting a manifold of different devices, allowing users to see the system like a common database and querying it in a simple, user-friendly query language.
Unfortunately, traditional data-retrieving and processing techniques from the database community cannot be applied directly to Cooperating Objects, since the traditional assumptions about reliability, availability and requirements of data sources cannot be extended to simple sensors. In fact, when comparing *sensor-based* data sources and *traditional* database sources some relevant differences can be identified. First of all, sensor nodes in a Sensor Network have (typically) limited processors and battery resources, they do not deliver data at reliable rates and the data may often be corrupted. Secondly, since sensors typically produce data continuously or at pre-defined time intervals, near real-time processing may be required, both because storing raw sensor streams may be extremely expensive and because sensor data may represent real-world events the user would like to be aware of and eventually respond to. Thirdly, sensor nodes are typically connected in an ad-hoc manner and must share common protocols and algorithms to collect, transmit and process data. The use of a multi-hop transmission strategy may, for example, allow the

in-network processing and data aggregation strategies as query-answers flow through the network to reach a central sink [Mad02].
Thus the traditional database approach needs to be readjusted to cope with the new requirements of Co-operating Objects. The benefits of this kind approach overwhelm however the drawback of a new design and brought several researchers to the development of query-like interfaces to sensor networks like *Cougar* ([BGS00], [YG02]), *IrisNet* ([iri], [NKG+02]) and *TinyDB* ([MFHH02], [MFHH03], [Mad02]). Section 4.1.4 provides some implementation details about the *Cougar* approach, while the

TinyDB. As a representative example of a query-like interface to sensor networks we examine *TinyDB*, which is a query processing system for extracting information from a network of tiny wireless sensors developed at the Intel Research Laboratory Berkeley in conjunction with the UC Berkeley. "Given a query specifying your data interests, *TinyDB* collects that data from motes in the environment, filters it, aggregates it together, and routes it out to a PC. *TinyDB* does this via power-efficient in-network processing algorithms" [tin].
The *TinyDB* query processor runs on top of the TinyOS [HSW+00] operating system (see for details Section 4.1.2) and each sensor node within the network needs to be endowed with an instance of the processor before deployment. *TinyDB* supports a single virtual database table sensors, where each column corresponds to a specific type of sensor (e.g., temperature, light) or other source of input data (e.g., sensor node identifier, remaining battery power). Reading out the sensors at a node can be regarded as appending a new row to sensors table. The query language is a subset of *SQL* with some extensions. In order to understand the way *TinyDB* works, consider the following query example. Several rooms are equipped with multiple sensor nodes each. Each node is equipped with sensors to measure the acoustic volume. The table sensors contains three columns room (i.e., the room number the sensor is in), floor (i.e., the floor on which the room is located), and volume. We can determine rooms on the 6th floor where the average volume exceeds the threshold 10 with the following query:

```
SELECT AVG(volume), room FROM sensors
   WHERE floor = 6
   GROUP BY room
   HAVING AVG(volume) > 10
   EPOCH DURATION 30s
```

The query first selects rows from sensors at the 6th floor (`WHERE floor = 6`). The selected rows are grouped by the room number (`GROUP BY room`). Then, the average volume of each of the resulting groups is calculated (`AVG(volume)`). Only groups with an average volume above 10 (`HAVING AVG(volume)` > 10) are kept. For each of the remaining groups, a pair of average volume and the respective room number (`SELECTAVG(volume), room`) is returned. The query is re-executed every 30 seconds (`EPOCH DURATION 30s`), resulting in a stream of query results. *TinyDB* uses a decentralized approach, where each sensor node has its own query processor that preprocesses and aggregates sensor data on its way from the sensor node to the user. Executing a query involves the following steps: Firstly, a spanning tree of the network rooted at the user device is constructed and maintained as the network topology changes, using a controlled flooding approach. Secondly, a query is broadcasted to all the nodes in the network by sending it along the tree from the root towards the leafs. Thirdly, nodes fulfilling the query criteria select the requested data and send them back to the sink. Data can eventually be aggregated as they flow through the network [2]. More details about the *TinyDB* query-processor are provided in Section 4.1.4 and are available, amongst others, in [MFHH02], [MFHH03], [Mad02], [tin].

In the context of Cooperating Objects, the database view is also used in the *PerSEND* system to support proximate collaborations between PDAs. In this model, a federated view of a database is maintained from the data available on each node. The database model is relational, and the system proposes an *SQL*-like interface to the applications. The database view is dynamic in the sense that it directly reflects a physical context. This context is represented by the set of near-by objects. As objects moves, the context evolves and the data associated to the objects are added or deleted from the database view. This system relies on a decentralized architecture, using only peer to peer communications (one-hop) over short distance wireless interfaces.
An important aspect of this database approach is that the system supports the notion of continuous queries. This means that the dynamics of query results can be managed at the system level, instead of the application one. For example, consider a continuous query for the evaluation of the maximum offer in a bidding: the query result continuously reflect the best bid, the data dependency between the best bid and current offers is managed at the system level.

3.2.2 Event Detection

Events are a natural way to both represent and trigger state changes in the real world and in distributed systems, giving rise to model applications as producers, consumers, filters, and aggregators of events. Regardless of the specific scenario, "interesting events " may represent node-internal occurrences (timeouts, message sending or receiving) or specific sensing results. Thus, the application can specify interest in certain state changes of the real world (*basic events*) and upon detecting such an event, a sensing-device sends a so-called *event notification* towards interested applications. The application can also specify certain patterns of events (*compound events*), such that the application is only notified if occurred events match this pattern [R̈04], [LMRV00].
The *Event Detection* paradigm is particularly well suited to provide a programming abstraction for sensor network applications. We discuss *DSWare* [LSA03] as a representative example of this category.

DSWare. *DSWare* is a software framework that supports the specification and automated detection of compound events. A compound event specification contains, among others, an event identifier, a detection range specifying the geographical area of interest, a detection duration specifying the time frame of interest, a set of sensor nodes interested in this compound event, a time window W, a confidence function f, a minimum confidence c_{min}, and a set of basic events E. The confidence function f maps E to a scalar value. The compound event is detected and delivered to the interested sensor nodes, if $f(E) \geq c_{min}$ and all basic events occurred within time window W. Consider the example of detecting an explosion event, which requires the occurrence of a light event (i.e., a light flash), a temperature event (i.e., high ambient temperature), and a sound event (i.e., a bang sound) within a subsecond time window W. The confidence function may be defined as:

```
f = 0.6 * B(temp) + 0.3 * B(light) + 0.3 * B(sound)
```

[2]The *TAG* engine may be provided with *TAG* [MFHH02], an aggregation service for in-network processing of data.

The function B maps an event ID to 1 if the respective event has been detected within the time window W, and to 0 otherwise. With $c_{min} = 0.9$, the above confidence function would trigger the explosion event if the temperature event is detected along with one or both of the light and sound events. This confidence function expresses the fact that detection of the temperature event gives us higher confidence in an actual explosion happening than the detection of the light and sound events. *DSWare* includes also various real-time features, such as deadlines for reporting events, and event validity intervals.
For completion, some architectural issues about the *DSWare* abstraction are discussed in Section 4.1.4.
The event detection paradigm is also used for higher level programming model for Cooperating Objects, such as in LINDA-like tuple spaces systems and databases. In tuples spaces, a node interested in an event create a tuple pattern and read the tuple space for this pattern (which is equivalent to subscribing to an event class, corresponding to the pattern). An actuator node produces an event by publishing a tuple. When one or more tuples are matching a given tuple pattern, the corresponding thread is waked up read operation returns) and can process the event.

```
actuator thread:
out<'event'>

listener thread:
rd<'event'>  // handling the event
```

We must note that in this approach event handling is synchronous, while asynchronous event handling may be expressed either through multiple threads (one per event), or through a "catch all" pattern, and then determining the type of event which occurred.
Database approaches which offer *triggers* also support a form of event handling: the trigger is defined by a logical predicate on a query, with the associated code to process when its predicate becomes true. This mechanism is proposed in system as old as *Xerox Parctab*, to trigger pre-programmed operations in a database of location-dependant data.

3.2.3 Virtual Markets

The market-based approach offers a very expressive and intuitive way to model and analyse typical distributed control problems as well as guidelines for the design and implementation of distributed systems. This methodology has also been proposed as a generic programming paradigm for distributed systems and addressed as *Market-Oriented Programming* [Wel96]. This approach regards modules in a distributed system as autonomous agents holding particular knowledge, preferences and abilities and the distributed computation may be implemented as a market price system [MW96]. This abstraction is particularly suited to model systems where different devices need to cooperate and coordinate in order to reach a common goal in a globally efficient way. Under this point of view, the system is seen as a virtual market where agents (i.e., single devices) act as self-interested entities, which regulate their behavior to achieve maximal profit with minimal costs (resource usage) considering globally-known price information (set in order to achieve a globally efficient behavior).
Mainland et al. [MKL$^+$04] applied the basic ideas of *Market-Oriented Programming* and defined the *Market-Based Macroprogramming* (*MBM*) paradigm, a promising programming abstraction for sensor networks.

Market-Based Macroprogramming. In the *MBM* approach, sensor nodes are seen as agents that perform actions to produce goods in return for (virtual) payments. Goods prices are globally-advertised throughout the network and single nodes decide to perform only those action that maximize their (local) utility function, whose value depends on both the node's internal state and the payment the node will (virtually) get to perform those actions. By dynamically tuning goods' prices, a user can force nodes to perform desired actions and network re-tasking is accomplished by adjusting prices rather than injecting new code on sensor nodes. *MBM* also allows multiple users to share the network by offering different payments for node actions, providing this way a sort of "free market". In order to preserve network resources, the energy budget of a node has to be taken in account when computing the utility function value.

In the context of the cooperation of personal communication devices, some studies are investigating the use of economic-like regulation systems to enforce a global objective. In the *IST Secure* project, which aims at trust management in uncertain environments (such as the cooperation between newly discovered PDAs), a general *trust model* based on the notion of reputation is proposed. Each possible action involved in the cooperation is associated with a set of possible*outcomes*. Each node maintains an *evidences store* to log the interactions with other nodes and interesting events. Based on the history of interactions and observed outcomes, nodes are able to dynamically build a notion of *reputation*. Essentially, positive outcomes lead to "good" reputation. If an action is required in an interaction, a benefit/risk analysis is performed, and a decision is taken based on the active policy for the node. For example, if the risk is beyond a given threshold, or if the benefit is below a given value, the action may be rejected.
Similar mechanisms are investigated in a French study called *Mosaic*, which aims at providing system support for collaborative backup of vulnerable personal devices (PDA, phones, digital camera...). Each device uses other devices to save part of their data, and provide spare space to backup other devices. Fair use of the resources (space and energy) is considered as critical for the success of this kind of applications, and market-like regulation is a promising approach to ensure this goal. The idea is to associate a currency to the resources, and accounting for their use when a device provides backup to another. Backing up data "gives" credits to a device while saving data to another device "uses" credits.

3.2.4 Virtual Machines

The concept of virtual machines is well-known since the early sixties and used to indicate a piece of computer software able to shield applications from the details of an underlying hardware or software platform. A virtual machine offers to applications a suite of *virtual instructions* and attends to map them to the real instruction set actually provided by the underlying real machine. In this way, the virtual machine abstraction can mask differences in the hardware and software laying below the virtual machine itself, thus facilitating code and data mobility.
In a system of complex Cooperating Objects, the interchange of data and other information may be very cumbersome since single devices may show a broad range of different internal architectures and protocols: this problem may be elegantly overcome by providing each different device with an adequate version of the virtual machine, which will hide the single device peculiarities and provide a "unique" virtual hardware and software setting to applications. In this way, applications can easily be written to run on the virtual machine itself instead of having to create separate application versions for each different platform. The virtual machine approach has already been deeply investigated for different applications, [Dic73], [BDR97], [CSS$^+$91], [LY99], but only a few approaches have been proposed and designed explicitly for Cooperating

Objects. A tiny virtual machine specifically designed for sensor networks is *Maté*, developed at the Intel Research Laboratory Berkeley in conjunction with UC Berkeley. We now provide a brief description of the *Maté* and *MagnetOS* [BBD$^+$02] virtual machines, while for further implementation details the reader may refer to Section 4.1.3

Maté. Maté is a byte-code interpreter that allows to concisely describe a wide range of sensor network applications through a small set of common high-level primitives. The design of the Maté virtual machine focused on producing a very concise instruction set, in order to allow complex programs to be very short and thus feasible to be flooded into the network with limited energy-costs. *Maté*-Code is sent through the network in small capsules of 24 instructions, each of which is a single byte long (thus, a single capsule fits in a *TinyOS* packet). *Maté*'s high-level abstraction provides an efficient way to frequently reprogram a sensor network reducing code transmission costs and thus saving precious energy resources.

MagnetOS. Energy-saving issues have also been the guidelines that led to the design of *MagnetOS* [BBD$^+$02], a power-aware adaptive operating system, specifically developed to work both on single nodes as well as across a large ad-hoc network. *MagnetOS* provides a unified system abstraction to applications, that see the entire network as a single unified Java virtual machine. The *MagnetOS* system is able to adapt to changes in resource availability, network topology and applications behavior, it supports efficient power consumption policies and hides network's heterogeneity to the applications.

The use of virtual machine architectures is also widespread for larger Cooperating Objects like PDAs and mobile phones. The most common and well-known one is *Java*. Some specific motivations exist in this context: in devices like mobile phones, the core services (like voice communication) are provided by the native environment, while additional applications/services (like games etc.) are confined in the virtual machine. This prevent additional services to compromise the operation of the core system, which is considered as dependable, without preventing extensibility. In addition, the isolation provided by the VM avoid untrusted code to access sensitive data, such as contact information.
Despite the many attractive features of the *Java* virtual machine, we must highlight some practical limitations which exist for Cooperating Objects: the heterogeneity abstraction is somewhat limited, as platform fragmentation is important (PJava, MIDP profiles, various API scattered in various optional JSRs...). Also, the native environment may still be impacted by malicious or buggy code running in the VM, as it usually shares some resources with the VM: the CPU, memory, energy. While CPU and memory usage can easily be controlled, energy is more difficult.

3.2.5 Mobile Code and Mobile Agents

Mobile Code is a general notion that indicates a software programm transmitted from one entity to another through a network to be executed at the destination. *Remote Evaluation*, *Code-on-demand* and *Mobile Agents* are the three basic paradigms that are encompassed in the notion of *Mobile Code*. *Mobile Agents*, in particular, represent mobile code that autonomously migrates between entities and they are therefore well suited for the implementation of distributed applications.
The notion of *Mobile Agents* may be easily seen as an efficient programming strategy for sensor networks, since sensing tasks may be specified as mobile code that may spread across the network piggybacking collected sensor data. These scripts may be injected into the network at any point and are able to travel

autonomously through the network and distribute themselves where and when necessary.
A possible approach to the definition, implementation and deployment of such scripts is provided by Boulis et al. in [BHS03], where the design and implementation of *SensorWare*, an active sensor framework for sensor networks, is presented.

SensorWare. In *SensorWare*, programs are specified in Tcl [Ous94], a dynamically typed, procedural programming language. The functionality specific to *SensorWare* is implemented as a set of additional procedures in the Tcl interpreter. The most notable extensions are the `query`, `send`, `wait`, and `replicate` commands. `query` takes a sensor name and a command as parameters. One common command is `value` which is used to obtain a sensor reading. `Send` takes a node address and a message as parameters and sends the message to the specified sensor node. Node addresses currently consist of a unique node ID, a script name, and additional identifiers to distinguish copies of the same script. The `replicate` command takes one or more sensor node addresses as parameters and spawns copies of the executing script on the specified remote sensor nodes. Node addresses are either unique node identifiers or broadcast (i.e., all nodes in transmission range). The `replicate` command first checks whether a remote sensor node is already executing the specified script. In this case, there are options to instruct the runtime system to do nothing, to let the existing remote script handle this additional user, or to create another copy of the script. In *SensorWare*, the occurrence of an asynchronous activity (e.g., reception of a message, expiry of a timer) is represented by a corresponding event. The `wait` command expects a set of such event names as parameters and suspends the execution of the script until one of the specified events occurs. The following script is a simplified version of the *TinyDB* query and calculates the maximum volume over all rooms (i.e., over all sensor nodes in the network):

```
set children [replicate]
set num_children [length children]
set num_replies 0
set maxvolume [query volume value]
while {1} {
   wait anyRadioPck
   if {maxvolume < msg_body} {
      set maxvolume msg_body }
   incr num_replies
   if {num_replies = num_children} {
      send parent maxvolume
      exit }
}
```

The script first replicates itself to all nodes in communication range. No copies are created on nodes already running the script. The `replicate` command returns a list of newly infected sensor nodes (children). Then, the number of new children (`num_children`) is calculated, the reply counter (`num_ replies`) is initialized to zero, and the volume at this node is measured (`maxvolume`). In the loop, the wait blocks until a radio message is received. The message body is stored in the variable `msg_body`. Then, `maxvolume` is updated according to the received value and the reply counter is incremented by one. If we received a reply from every child, then `maxvolume` is sent to the parent script and the script exits. Due to the recursive replication of the script to all nodes in the network, the user will eventually end up with a message containing the maximum volume among all nodes of the network. Further details about the *SensorWare* approach are reported in Section 4.1.3.

In the broader context of cooperating personal devices, few systems use the Mobile Agent paradigm, as for example the Sony *Shop Navi* project [NR96]. Beyond security issues, a major problem is the need of a common agent hosting environment which is a difficult constraint given the heterogeneity of the devices. An important advantage of mobile code is the support for dynamic deployment of software (over the air provisioning), which allows dynamic adaptation of the same devices in new situations requiring additional software support.

3.2.6 Role-based Abstractions

Many sensor network applications require some form of self-configuration, where sensor nodes take on specific functions or roles in the network without manual intervention. These roles may be based on varying sensor node properties (e.g., available sensors, location, network neighbors) and may be used to support applications requiring heterogeneous node functionality (e.g., clustering, data aggregation).
The concept of role assignment is thus an implicit part of many networking protocols as well as a common function of middleware platforms for sensor networks. Heinzelman et al. proposed in [HMCP04], *MiLAN*, a middleware able to control the allocation of functions to sensor nodes in order to meet certain quality-of-service requirements specified by the user. In [UWMG05], another high-level role-based programming approach for sensor networks is presented. In this work, a high-level task specification is compiled into a set of node-level programs that must be properly allocated to sensor nodes taking into account the node capabilities. These approaches typically support only very specific role assignment tasks and do not offer a general solution for the role specification problem. Providing such a general solution is the primary effort of Frank et al. [FR05], [RFMB04], whose framework for *generic role assignment* is hier briefly sketched.

Generic Role Assignment. In this approach, a developer can specify user-defined roles and rules for their assignment using a high-level configuration language. Such a *role specification* is a list of role-rule pairs. A role is simply an identifier. For each possible role, an associated rule specifies the conditions for assigning this role. Rules are Boolean expressions that may contain predicates over the local properties of a sensor node and predicates over the properties of well-defined sets of nodes in the neighborhood of a sensor node. *Properties* of individual sensor nodes are available sensors (e.g., temperature) and their characteristics (e.g., resolution); other hardware features (e.g., memory size, processing power, communication bandwidth); remaining battery power; or physical location and orientation. A *distributed role assignment algorithm* assigns roles to sensor nodes, taking into account role specifications and sensor node properties. A separate instance of the role assignment algorithm is executing on each sensor node. Triggered by property and role changes on nodes in the neighborhood, the algorithm evaluates the rules contained in the role specification. If a rule evaluates to true, the associated role is assigned.
Consider the following *coverage* example. A certain area is said to be covered if every physical spot falls within the observation range of at least one sensor node. In dense networks, each physical spot may be covered by many equivalent nodes. The lifetime of the sensor network can be extended by turning off these redundant nodes and by switching them on again when previously active nodes run out of battery power. Essentially, this requires proper assignment of the roles `ON` and `OFF` to sensor nodes. The following

role specification implements this.

```
ON :: {
    temp-sensor == true &&
    battery >= [threshold] &&
    count(2) {
        role == ON &&
        dist(super.pos, pos) <= [sensing-range]
    } <= 1 }
OFF :: else
```

The rule in lines 1-7 specifies the conditions for a node to have `ON` status: it must have a temperature sensor and enough battery power (lines 2 and 3). As a third condition, we require that at most one other `ON` node should exist within this node's sensing range. This is specified by the `count` operator in line 4. It expects a hop-range as its first parameter and returns the number of nodes within this range for which the expression in curly braces evaluates to true. Here we request to evaluate nodes within 2 network hops. Note that the used property names (e.g., `role` in line 5, `pos` in line 6) in the nested expression refer to properties of the specified neighbor nodes. To access properties of the current node instead, the prefix `super` is used (e.g., `super.pos` in line 6). The `dist` operator used in line 6 returns the metric distance between two positions. In the example, it specifies that only nodes located within this node's sensing range should be counted.

3.2.7 Group-based Approach

The *clustering* paradigm is well-known in the field of distributed systems and ad-hoc networks ([RHH01], [Bir93]) and offers a suitable programming abstraction for systems collecting a manifold of complex devices, which cooperate and coordinate to reach a common goal [EGHK99].
In a sensor network, for example, nodes that share some neighborhood relationship can organize themselves in *groups* that constitute single addressable entities for the programmer and within which nodes can efficiently communicate and collectively exploit local resources. The *Hood* and *Abstract Regions* paradigms are the approaches that will now be closer analyzed.

Hood. Whitehouse et al. managed to define the neighborhood concept as a proper programming primitive by designing *Hood*, an abstraction "which allows users to think about algorithms directly in terms of neighborhoods and data sharing instead of decomposing them into messaging protocols, data caches and neighbor lists" [WSCB04]. For a given network task, *Hood* defines the membership criteria and the attributes to be shared within a group and provides an interface that shows the names of the current neighbors as well as the list of the shared attributes, hiding to the applications all the nesting details about neighbors discovering and data sharing, data caching and messaging within a single group. This group-based approach seems a suitable solution for sensor networks, since it scales well for increasing network size, it is robust to node failures and allows dynamic network reconfiguration.

Abstract Regions. Another neighborhood-based approach is also followed in the design of *Abstract Regions*, a set of "general-purpose communication primitives for sensor networks that provide addressing, data sharing and reduction within local region of the network" [Wel03]. Regions are just a collection of nodes and may be defined on the base of geographical, topological or connectivity predicates such as "*all nodes within 10 meters*" or "*all nodes in 1-hop communication distance*", and nodes within a region communicate, share variables and data, and provide aggregation. *Abstract Regions* provide a communication abstraction that simplifies application design by hiding local actions within regions (communication, data dissemination and aggregation) and by allowing applications to explicitly trade off resource consumption and accuracy of global operations [WM04].

3.2.8 Spatial Programming

Accessing network resources using *spatial references*, in the same way as in traditional imperative programming variables are accessed using memory references, is the underlying idea of *Spatial Programming*, a space-aware programming paradigm particularly suitable for distributed embedded systems. In this view, a networked embedded system is seen as a single virtual address space and applications can access network resources by defining a *spatial reference*, i.e. a pair {*space:tag*}, where *space* indicates the expected physical location and *tag* a property of the demanded network resource [BIK+04].
A system that supports the *Spatial Programming* abstraction has been implemented by Borcea et al. using *Smart Messages*, a software architecture that recalls many concepts and constructs typical of mobile agents [BIK+02].
Other systems that exploit the *Spatial Programming* approach are *SIS* [BW99], *Close Encounters* [KST99] and *Ubibus* [BCPB04]. In these works, Cooperating Objects are used as data symbols that cover a given geometrical shape and the physical space is used as a way to structure information and processing. The idea is to annotate existing interactions of physical entities with computing actions. These actions are triggered according to geometrical conditions, in particular physical proximity.

3.2.9 Shared Information Space

Devices that need to cooperate to accomplish a global task, need also to share data and information about their internal states. In traditional centralized systems *Shared Information* is usually stored in a physical central place, accessible for all entities participating in the system.
In systems like sensor networks, sensor nodes need to cooperate and coordinate and thus need to share information in order to efficiently perform high-level sensing tasks. A centralized solution is, however, not suitable for such systems since making single nodes reporting data and state information to a central unit poses an extremely high communication overhead (thus, inefficient energy management) and provides a single point of failure. Since useful information is "spread" among a manifold of single entities, a distributed solution is needed. Koberstein et al. proposed to assume a sensor network to behave like a *swarm*, where nodes cooperate with each other by sharing knowledge about swarm state and external conditions [KLBF04]. This knowledge is "stored" in a *distributed virtual Shared Information Space* (*dvSIS*), an abstract entity that may be represented as union of local instances stores on single sensor nodes. Since a single local instance may contain information that is no-longer valid or may be inconsistent with the local instance of other nodes, requirements on consistency and completeness need to be very strict. Using broadcast, a single node can publish new acquired data in the *dvSIS*, thus other nodes can enhance their

local instances just by listening to broadcasts. Even if it is often desirable that all participating nodes have identical views on this space (i.e., all nodes see the same data items), a more efficient implementation can be provided if nodes are allowed to have slightly different views on the space, without affecting a correct global behavior of the network.

The shared information space paradigm is also useful to coordinate mobile computing and ubiquitous computing applications. Tuple spaces similar to Linda are used in mobile computing with *LIME* [PMR99], and a spatial tuple space enabling spatial programming for ubiquitous computing is proposed in the *SPREAD* system [CB03].

3.2.10 Other Approaches

Among the other approaches presented we would also like even if only briefly to discuss comprehend:

Service Discovery. Systems adopting this approach allow a node to discover services available in the current context, in particular those provided by neighborhood nodes. This abstraction is related to traditional distributed computing approaches, such as client-server interactions or distributed objects method invocations. Typically, an object uses the service discovery service prior to being able to interact with the neighboring objects. Some existing systems based on this approach are *JINI* [Wal99], *Bluetooth SDP* [bt:01], *Cooltown URL beaming* [KBM+00].

Client-Server Approaches. In this simple approach, each Cooperating Object hosts one (or more) server(s) enabling other objects to use its services. The Cooltown system is typical of this approach, where Cooperating Objects (printers, etc.) are running embedded web servers, accessed by other objects (PDAs etc.) running web clients.

Distributed Objects. This approach is similar to the previous one, but services are provided by *objects* (in the object-oriented sense), and interactions between these objects are supported through remote method invocation. Some examples of this approach include JINI and CORBA's ORB [Wal99], [Wea02].

Some other interesting approaches presented in the research community like *EnviroTrack* [ABC+04, BNW+03], *MiLAN* [HMCP04]. *Impala* [LM03] and *TinyCubus* [MLM+05, MMLR05] are discussed in Sections 4.1.4 and 4.1.5.

3.3 Summary and Evaluation

In Section 3.1 we discussed the fundamental attributes of Cooperating Objects and derived from them the requirements with which an adequate programming abstraction should be able to comply in order to allow an end-user to make a proper and efficient use of the system. On the basis of the listed requirements we outlined the characteristics that in our opinion should be considered as the most significant when evaluating the suitability of a programming abstraction for Cooperating Objects. In particular, we underlined that the following aspects should be considered:

- **Ease of programming and Expressiveness.** Does the abstraction provide easy-to-use programming primitives? How expressive is the provided set of programming primitives? Does this set allow to access all the functionalities the network is able to provide? Does the abstraction allow an application developer to program the network as a whole, i.e., to specify global network tasks instead of single devices roles?
- **System Diversity.** Does the abstraction hide device heterogeneity to the programmer?
- **System Dynamic.** Is the abstractions able to deal with unknown/unstable connectivity, topology and/or network size? How does the abstractions deal with temporary or permanent device failures?
- **Environmental Dynamic.** Does the abstraction provide adequate primitives to define, detect and react to real-world events?
- **Resource Constraints.** Is the abstraction able to cope with hardware, computing and power constraints?
- **Scalability.** Is the set of provided programming primitives suitable for growing network size and/or density?
- **Deployment and Maintenance.** Is it possible to perform a resource-efficient debugging at application level?

In Section 3.2 we provided a survey on the past and ongoing work on programming models for Cooperating Objects. The surveyed approaches were sorted in different categories and for each category we presented a concrete implementation example. We thus discussed along the way the most known and successful programming abstractions specifically targeted to Cooperating Objects, and provided an overview on existing running systems like *TinyDB*, *DSWare*, *Maté*, *MagnetOS*, *SensorWare*, *Generic Role Assignment*, *Hood* and *Abstract Regions*, just for citing a few.
Most of the proposed approaches were designed for a specific application scenario or were tailored to some specific design goals and appear thus able to comply with only a subset of the above listed requirements. and even if most of them (e.g., database approach, agent-based approach, event-based approach, virtual markets) are not new, they required significant adaptation for being used for Cooperating Objects and/or sensor networks. These approaches differ with respect to ease of use, expressiveness, scalability, overhead, etc., as we will not outline by means of three representative examples, namely, *TinyDB*, *SensorWare* and *DSWare*.
TinyDB provides the user with a declarative query system which is very easy to use. The database approach hides distribution issues from the user and rather than programming individual objects, the network can be programmed as a single (virtual) entity. On the other hand, the expressiveness of the database approach is limited in various ways. Firstly, adding new aggregation operations is a complex tasks and requires modifications of the query processor on all objects. But more importantly, it is questionable whether more complex tasks can be appropriately supported by a database approach. For example, the system does not explicitly support the detection of spatio-temporal relationships among events in the real-world (e.g., expressing interest in a certain sequence of events in certain regions). The system also suffers from some scalability issues, since it establishes and maintains network-wide structures (e.g., spanning tree of the network, queries are sent to all devices). In contrast, many sensing tasks exhibit very local behavior (e.g.,

tracking a mobile target), where only very few devices are actively involved at any point in time. This suggests that *TinyDB* cannot provide optimal performance for such queries. Additionally, the tree topology used by *TinyDB* is independent of the actual sensing task. It might be more efficient to use application-specific topologies instead. A clearly different approach is the one followed in *SensorWare*, where an imperative programming language is used to task individual nodes. Even rather simple sensing tasks result in complex scripts that have to interface with operating system functionality (e.g., querying sensors) and the network (e.g., sending, receiving, and parsing messages). On the other hand, *SensorWare*'s programming paradigm allows the implementation of almost arbitrary distributed algorithms. Typically, there is no need to change the runtime environment in order to implement particular sensing tasks. However, the low performance of interpreted scripting languages might necessitate the native implementation of complex signal processing functions (e.g., Fast Fourier Transforms, complex filters), thus requiring changes of the runtime environment in some cases. *SensorWare* allows the implementation of highly scalable applications, since the collaboration structures among sensor nodes are up to the application programmer. For example, it is possible to implement activity zones of locally cooperating groups of sensor nodes that follow a tracked target. The *SensorWare* runtime does not maintain any global network structures. One potential problem is the address-centric nature of *SensorWare*, where specific nodes are addressed by unique identifiers, thus potentially leading to robustness issues in highly dynamic environments. Another interesting comparison can be made with respect to *DSWare*, a system that provides compound events as a basic programming abstraction. However, a complete Cooperating Objects application will require a number of additional components besides compound event detection. For example, code is needed to generate basic events from sensor readings, or to act on a detected compound event. *DSWare* does not provide support for this glue code, requiring the user to write low-level code that runs directly on top of the sensor node operating system. This makes the development of any application a complex task, while at the same time providing a maximum of flexibility. *DSWare* supports only a very basic form of compound events: the logical and of event occurrences enhanced by a confidence function. It might be worthwhile to consider more complex compound events, such as explicit support for spatiotemporal relationships among events (e.g., sequences of events, non-occurrence of certain events). Note that more restrictive compound event specifications can avoid the transmission of event notifications and can hence contribute to better energy efficiency and scalability. Without re-discussing all the abstractions surveyed in Section 3.2, we can conclude that the examined approaches exhibit a tradeoff between ease of use and expressiveness. While *TinyDB* is easy to use, it is restricted to a few predefined aggregation functions. More complex queries either require changes in the runtime environment, are inefficient, or cannot be expressed at all. While *SensorWare* and *DSWare* support the efficient implementation of almost arbitrary queries, even simple sensing tasks require significant programming efforts. Narrowing this gap between ease of use and expressiveness while concurrently enabling scalable and energy-efficient applications is one of the major challenges in the design of adequate programming abstractions for Cooperating Objects. It is not yet clear, whether suitable programming models will be inspired by known paradigms as in the presented examples or if completely new approaches need to be defined.

4 System Architectures

As we already mentioned in the introduction, we will refer to a *System Architecture* as "the structure and organization of a computing system, as a set of functional modules and their interactions". In this section we will survey the state of the art of system architectures for Cooperating Objects. We divide this survey into two parts:

- **Node internals.** This part presents several possible abstraction levels in a single node. Operating systems, the simplest approach, provide basic system functionality including an uniform way of accessing the hardware. A *Data Management Middleware* hides the data sources and offers precomputed information to applications, in some cases also in collaboration with other Cooperating Objects. *Virtual Machines* abstract from hardware completely, offering a virtual execution environment to the user. *Adaptive System Software* hides the changing environment of a real-world Cooperating Object from the user.
- **Interaction of nodes.** This part presents system architectures topics related to interaction of nodes. We abstract two main sets of functionalities, namely low-level functionalities including tasks corresponding to physical, link, routing, and transport layers as well as high-level functionalities, including coordination and support, clustering, timing and localization, addressing, lookup, collaboration, failure detection, and security.

4.1 System Architectures: Node Internals

In this section we will survey the state of the art of node internal system architectures. Our survey discusses operating systems, virtual machines, data management middleware and adaptive system software for Cooperating Objects.
The most important requirements are related to the requirements discussed in Section 3.1. The main difference to the architectures designed for traditional systems are the resource-constraints, in particular regarding the memory footprint and limited energy budget of the target system. Therefore, energy-efficiency and a small memory footprint are indispensable features of these architectures. Other important requirements include flexibility to cope with different applications and hardware as well as adaptivity and a small learning curve. Further, portability can be regarded as a desirable feature.

4.1.1 Data-centric and service-centric approach

The field of Cooperating Objects comprises a wide range of applications [3]. Sensor networks scenarios are said to be *data-centric* whereas pervasive or ubiquitous computing scenarios are more *service-centric*. In this section, both approaches are described.
A service is a well-defined and self-contained function that does not depend on the context or the state of other services. The service is executed on the explicit request of a caller which has to know the interface of the service. A response is returned after the completion of the service.
In a data-centric approach, the execution is controlled by the data. For example, on the basis of the type of incoming data, the appropriate function is called which is able to handle this type. Although

[3] See also study 3.1.1)

the user of a data-centric system may communicate with it by functions of a well-defined interface and although the components of a system may internally exchange the data over such interfaces, in contrast to a service-centric approach, the desired functionality is not specified explicitly by the name of the service but implicitly by the passed data.
There is no obvious hierarchical ordering of service-centric and data-centric approaches. On the one hand, a data-centric system could use several services to fulfil its task. But on the other hand, a service-centric approach could fall back on a lower-level data-centric query. Thus, this approach abstracts from the data.

4.1.2 Operating Systems

In this section, we present different operating systems for Cooperating Objects. Unlike general-purpose desktop operating systems such as *Windows* or *Linux*, these operating systems run on devices that are designed for special-purpose tasks. The main tasks of these operating systems is to provide an abstract interface to the underlying hardware and to schedule system resources.
After a short discussion on scaled-down version of *Linux* and *Windows*, we briefly present some operating systems designed for handheld devices such as PDAs and mobile phones. These devices can be regarded as part of a network of Cooperating Objects (the first mobile phones with attached sensors are already available),or they can be used to access wireless sensor networks.
Many of the traditional embedded operating systems are designed for real-time systems with small memory footprints such as robot arms or break systems, whereas most current operating systems targeted for wireless sensor networks are not real-time systems. We first present three of these operating systems in more detail, namely *TinyOS*, *Contiki*, and *Mantis*. These systems span the whole spectrum of concurrency: *TinyOS* does not provide any multithreading, *Contiki* provides multithreading as a library for those applications that explicitly require it, and *Mantis* is a layered multithreaded operating system. Then other sensor node operating systems, *SOS*, *kOS*, *Timber* and *DCOS* are briefly presented.

- **Scaled down versions of desktop operating systems**
  - **Windows.** Microsoft *Windows* is the most common operating system for desktop computers. There are also embedded systems that run *Windows XP*, for example ATMs, set-top boxes and ticket vending machines. *Windows XP Embedded* is a modular cut-down version of XP that allows the designer to choose the modules to be used. This way the size of a system without networking, GUI and device drivers is limited to about four to five MBytes of memory.
  - **Linux.** Since *Linux* is covered under the GPL license [4], anyone can customize *Linux* to his PDA, Palmtop or other mobile or embedded device. Therefore, a multitude of scaled-down *Linux* versions exist. These include **RTLinux** (Real-Time Linux), an extension of the *Linux* kernel that provides real-time guarantees by inserting an additional abstraction layer between the kernel and the hardware, **uClinux**, a scaled-down *Linux* version for system without a memory mapping unit and thus no isolation between kernel and user-space processes, **Montavista Linux** with Linux distributions for ARM, MIPS, and PPC, **ARM-Linux**, and many others.
- **Operating Systems for handheld devices**

[4]General public License, http://www.gnu.org/copyleft/gpl.html

- **Palm OS.** The *Palm OS* is specifically designed for PDAs featuring a small screen, less processing power than desktop PCs and limited memory. In *Palm OS*, the kernel is responsible for thread scheduling, handling hardware interrupts, and other low-level management tasks. Although Palm-applications are single-threaded, the kernel itself uses multiple threads.
- **Symbian OS.** *Symbian OS* is a robust multi-tasking operating system, designed specifically for wireless environments and the constraints of mobile phones. The core kernel's size is less than 200 KBytes. The OS has support for handling low memory situations and a power management model. *Symbian OS* runs on fast, low power, low cost CPU cores such as ARM processors.
- **Windows CE.** In contrast to Windows XP Embedded, Windows CE has a different codebase than Windows XP. Windows CE is particularly designed for small hand-held devices. Windows CE is a preemptive multitasking operating system allowing multiple applications, or processes to run within the system simultaneously. Further, Windows CE provides deterministic interrupt latencies and real-time properties. Windows CE also provides programmable power conserving mechanisms. According to Microsoft the code size is 200 KBytes without graphics, but the code size increases dramatically when graphic and networking is included.

• **Embedded Real-Time Operating Systems**
There exist a large number of embedded real-time operating systems. Here we present a few of them:

- **eCos.** *eCos* [eCo] is an open-source system designed to be highly configurable. *eCos* has extensive configuration possibilities and can be scaled up from a few hundred bytes in size to hundreds of KBytes. *eCos* provides features such as pre-emptable tasks with multiple priority levels, low latency-interrupt handling,multiple scheduling policies, and multiple synchronization methods. The *eCos* development environment contains a set of Gnu-based tools that assist in making application specific configurations of *eCos* for each particular embedded system. *eCos* has compatibility layers for *POSIX* and *uITRON*.
- **QNX.** *QNX* [qnx] is a Unix-like operating system with real-time properties, and is the most prominent example of a successful micro-kernel design. The micro-kernel is surrounded by cooperating processes that provide higher level services such as inter-process and low-level networking communication, process scheduling and interrupt dispatching. *QNX* features a very small kernel of about 12 KBytes. *QNX* is designed for systems running x86, MIPS, PowerPC or ARM CPUs.
- **XMK.** *XMK* (eXtreme Minimal Kernel) [xmk] is an open-source real-time kernel designed to fit very small micro-controllers, yet be scalable up to larger systems. A minimal kernel configuration requires only 340 Bytes of ROM and 18 Bytes of RAM.

• **TinyOS**
TinyOS [HSW$^+$00] is an operating system specially designed for the constraints and requirements of wireless sensor networks. It is currently the most widely used system for academic research in the area of sensor networks. *TinyOS* is available for several platforms, e.g., Mica, Telos, EYES, imote. Additionally, a *TinyOS* simulator called *TOSSIM* is included. We will now briefly survey the most relevant aspects of this tiny operating system:

– **System modularity.** *TinyOS* builds on a component architecture where both applications and operating system consists of single, interlinked components. Thus, there is no strict separation between operating system kernel and application software. The operating system is not a separate program on which separate applications can rely on. Rather the necessary components of *TinyOS* are compiled together with the application to a single executable that contains both operating system and application components.

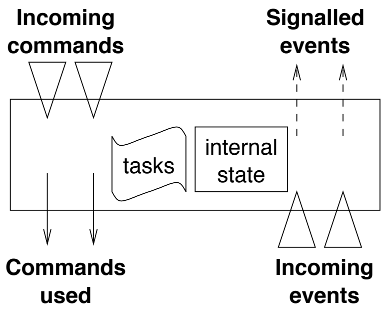

Figure 1: Schematic component of *TinyOS* consisting of tasks, internal state, commands, and events(after [HSW+00])

A component (see Figure 1) consists of fixed-sized state and tasks. Interaction between components is provided via function call interfaces that are sets of commands and events. Commands are used to initiate an action such as the transmission of a message. Events denote the completion of a request such as the completion of the transmission of a packet or an external event such as the reception of a packet. An application is composed by choosing a number of components and "wiring their interfaces together" [LMG+04].

– **Scheduling hierarchy.** *TinyOS* uses a two level scheduling hierarchy that lets high-priority events pre-empt low priority tasks. Events are invoked because of external input such as incoming data, sensor input or a timer. They are allowed to signal other events or call commands. Events and commands must not block the processor with time-consuming operations, but have to return in short time. Longer calculations have to be performed in a task. Tasks are a form of deferred procedure calls enabling postponed processing. This way, events can post tasks for later processing. When the calculation is done, the task can signal an event to inform the component about the result. Both events and tasks must run to completion after being invoked. This precludes the use of blocking statements. Events are implemented using hardware interrupts, and tasks are implemented using a linear FIFO dispatcher. The dispatcher has a queue of tasks, where each task is represented by a pointer to a function. In case the task queue is empty, the system can go into a sleep state and wait for the next interrupt. If this event posts a task, the dispatcher takes it from the task queue and runs it.

– **Concurrency.** *TinyOS* currently does not support multi-threading, blocking or spin loops. Therefore and as a consequence of the command-event model, many operations in *TinyOS* are so-called `split-phase`. A request is issued as a command that immediately returns. The completion of the request is signalled by an event. While this approach enables e.g. implicit error handling, it complicates program design and development since it more or less forces the programmer to implement blocking sequencing in a state-machine style.

  - **Programming.** The entire *TinyOS* system, as well as all applications running under it, is implemented in the *NesC* language [GLv+03]. *NesC* is an extension to the C language that supports event-oriented programming, i.e., the execution of function as a reaction to certain system events which is common for sensor networks. *nesC* introduces several keywords that allow the modelling of *TinyOS* components as described above. Using `command` and `event`, the commands, a component is able to receive, and the events, a component signals, can be defined. The invocation of commands is done via `call` and the triggering of events via `signal`. A set of commands and events can be encapsulated in an interface. The actual implementation is located in a module which can use and provide (keywords `uses` and `provides`) several interfaces. In a `configuration`, components are wired together, connecting interfaces used by components to interfaces provided by others. Every *nesC* application is described by a top-level configuration that wires together the components used. Additionally, the *nesC* compiler provides compile time checks for finding race conditions. Therefore, it finds all asynchronous code, i.e., code that is reachable from at least one interrupt handler. Every use of a shared variable from such asynchronous code is a potential race condition if the programmer does not use the `atomic` statement. This check can be done since the compiler processes the complete code including application and operating system components and has, therefore, knowledge about the interaction between them.

The foremost feature of *TinyOS* is its small code size and memory usage, and its component model that lets the system designer specify the system dependencies at compile time. The main drawback is the event-driven concurrency model which restricts applications to be implemented as explicit state machines. *TinyOS* is open-source software, published under a three-clause BSD license. Because of its widespread use in the wireless sensor networking research community, there is a wealth of implementations of various communication protocols for sensor network available.

- **Contiki**
  Contiki [DGV04] is an operating system designed for networked and memory constrained systems. *Contiki* is in many aspects similar to *TinyOS*, but has additional support for threads and dynamically loadable programs. *Contiki* includes the uIP stack for TCP/IP communication. Important properties of *Contiki* include its execution models, its system architecture, dynamic reprogramming, and portability:

  - **Execution models.** In order to keep its memory footprint small, *Contiki* is based around an event-driven kernel. Unlike *TinyOS*, *Contiki* allows applications to be written in a multi-threaded fashion. Multi-threading is implemented as a library that is optionally linked only with those applications that specifically require a threaded model of execution. The event-driven nature of the kernel makes the system compact and responsive, whereas the multi-threading makes it possible to run programs that perform long-running computations without completely blocking the system. For example, performing user authentication on a mote requires up to 440 seconds [BGR05]. Multi-threading enables the the system to handle incoming packets while performing such a long computation. Additionally, *Contiki* provides a third execution model called `protothreads` [DSV05]. In event-based systems, programs usually have to be implemented as explicit state machines and thus are hard to debug and maintain. `Protothreads` are stack-less thread-like constructs. They allow programs to be written in a sequential fashion and like threads provide conditional blocking on top of the event-driven system. Unlike threads, `protothreads` are extremely lightweight requiring only two bytes of memory per `protothread` and no additional stack.

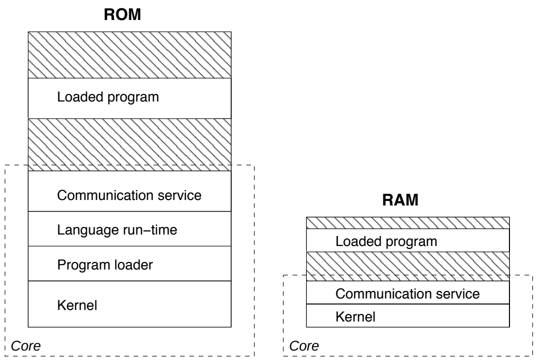

Figure 2: Contiki: partitioning into core and loaded programs [DGV04]

– **System architecture and partitioning.** A *Contiki* system is partitioned into two parts: the `core` and the `loaded programs` (see Figure 2). The partitioning is made at compile time and is specific to the deployment in which *Contiki* is used. Typically, the core consists of the *Contiki* kernel, the program loader, the most commonly used parts of the language, run-time and support libraries, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image that is stored in the devices prior to deployment. The core is generally not modified after deployment, even though it is possible to use a special boot loader to overwrite or patch the core. A *Contiki* system consists of the kernel, libraries, the program loader, and a set of processes. A process may be either an `application program` or a `service`. A `service` is a process that implements functionality that is used by other processes such as protocol stacks and data handling algorithms. Services can be seen as shared libraries which can be replaced during run-time. The kernel consists of a lightweight event scheduler that both dispatches events to running processes and periodically calls polling handlers used to e.g. check for status updates of hardware devices. The kernel also supports two kinds of events, namely asynchronous events which are a form of deferred procedure calls and synchronous events mainly used for interprocess communication. The kernels enqueues asynchronous events in a special event queue and dispatches the events later to the target process.

– **Dynamic reprogramming.** When developing applications for sensor networks, the ability to reprogram the sensor nodes without requiring physical access to the nodes greatly simplifies development and reduces the development time. *Contiki* has support for loading individual programs from the network, which makes it possible to dynamically reprogram the behavior of the network. After a program has been loaded into memory, the program's initialization function is called that may replace or start processes. A thin service layer, conceptually situated next to the kernel, provides service discovery and run-time dynamic service replacement within each sensor node. The ability to load and unload individual applications is important for wireless sensor networks, because an individual application is much smaller than the entire system binary and therefore requires less energy when transmitted through a network. Additionally, the transfer time of an application binary is less than that of an entire system image.

– **Portability.** *Contiki* is designed to be portable across a wide range of different platforms. *Contiki* runs on several platforms, including the ESB nodes from FU Berlin, Amtel AVR and the Intel x86, and the Z80 platform.

Module	**Code size (AVR)**	**Code size (MSP430)**	**RAM usage**
Kernel	1044	810	$10 + + 4e + 2p$
Service layer	128	110	0
Program loader	-	658	8
Multi-threading	678	582	$8 + s$
Timer library	90	60	0
Replicator stub	182	98	4
Replicator	1752	1558	200
Total	3874	3876	$230 + 4e + + 2p + s$

Table 1: Size of compiled Contiki code, in bytes [DGV04].

Contiki's memory requirements of an example sensor data replicator application are shown in Table 1. They depend on the maximum number of processes that the system is configured to have (p), the maximum size of the asynchronous event queue (e) and, if multi-threading is used, the size of the thread stacks (s).

- **Mantis**
One of the main design goal of the *Mantis* system [ABC+03] is ease of use to enable a small learning curve and rapid prototyping while meeting the resource constraints of wireless sensor networks in terms of limited memory and power. The other key goal of *Mantis* is flexibility by providing experienced programmers sophisticated sensor networks features including dynamic reprogramming over the radio and remote debugging of sensor nodes. The architecture of *Mantis* includes the following entities:

  – **Kernel and scheduler.** The goal of the *Mantis* and its kernel is to leverage familiar, traditional OS services in the realm of resource-constrained wireless sensor networks. *Mantis*' design resembles a traditional layered multithreaded design as shown in Figure 3. The *Mantis* kernel is designed similar to a traditional *UNIX*-style scheduler providing a subset of *POSIX*-threads. In particular, the scheduler supports priority-based thread scheduling as well as binary and counting semaphores. The main data structure of the kernel is a table holding one entry per

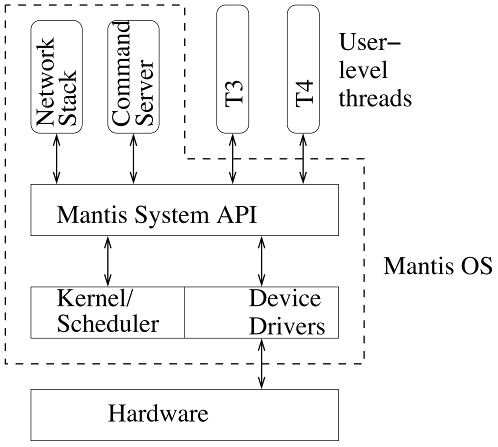

Figure 3: *Mantis* classical OS architecture [ABC+03]

thread. The size of the table cannot be adjusted dynamically but is determined at compile time (12 entries by default). Therefore, there is a fixed level of memory overhead. Each thread table entry has a size of 10 Bytes including e.g. a stack pointer, priority level, and the thread's starting function. The scheduler provides preemptive scheduling with time slicing. The hardware posts timer interrupts that cause the scheduler to initiate a context switch. Other interrupts than these are handled by the corresponding device drivers. On reception of an interrupt, device drivers post a semaphore to activate the corresponding thread. There is a special thread called the idle thread. This threads has the lowest priority and therefore runs when no other thread is runnable. The idle thread can be used to implement power-aware scheduling.

– **Network stack.** The design goal of *Mantis*' networking stack is efficient use of the limited memory as well as flexibility and convenience. The networking stack features a traditional layered design where the different layers can be implemented as several user-level threads or in one thread. The latter solution minimizes memory usage, avoids copying data between different threads and enables cross-layer optimizations. The advantages of the layered solution are modularization and flexibility. *Mantis* standard network stack consists of four layers: application layer, network layer, MAC and physical layer. The latter two are implemented as one user-level thread called base thread. If the network layer is implemented as a separate thread, the network layer thread blocks on a semaphore until the base thread posts the semaphore after e.g. the reception of a packet. The threaded model makes it also possible to activate or deactivate a particular protocol, e.g. a routing protocol. *Mantis* also enables the coexistence of more than one thread for a given task, for example, several routing protocol threads can run at the same time. In such a scenario, incoming packets are directed to the corresponding thread on a per-packet basis.

– **Dynamic reprogramming.** *Mantis* enables reprogramming of both the entire operating system and parts of the program memory by downloading a program image onto EEPROM, from where it can be burned into flash ROM. This capability is implemented as a system call library. The entire *Mantis* kernel uses about 12 KBytes ROM, and approximately 500 Bytes RAM. Thus, the memory requirements of *Mantis* are larger than those of *TinyOS* and *Contiki*. *Mantis* also features what the *Mantis* developers call a multimodal prototyping environment where the same code can be executed on both physical and virtual sensor nodes across X86 and Atmel platforms. In this prototype environment, both types of nodes can coexist and communicate with each other in a hybrid network. This is enabled by preserving a common C API across the platforms. With minor modifications, *Mantis* can be executed as an application running on X86 on both *Windows* and *Linux* since both operating systems support the AVR microcontroller.

	TinyOS	**Contiki**	**Mantis**
Concurrency	events	optional threading & protothreads	threads
Code size (ROM and RAM)	smallest	medium	largest
Ease of learning	hardest	medium	ease of programming major design goal

Table 2: Comparison of *TinyOS*, *Contiki* and *Mantis*

Table 2 compares the important features of *TinyOS*, *Contiki* and *Mantis* related to concurrency, code size and ease of learning. *TinyOS* is highly optimized to achieve a small code size, but hard to program while *Mantis* more traditional structure of a layered multithreaded operating system leads to the largest code size but simplifies programming. *Contiki* by design combines flexility with low memory footprint. While for non-expert programmers *TinyOS* is tricky to program [LC02], *Mantis* simplifies programming. The same is true for *protothreads* available in the *Contiki* operating system. However, *Mantis* large memory footprint makes it impossible to implement it on systems with limited memory such as some PIC micro-controllers to which *TinyOS* has been ported [LR05]. Smaller memory is both more cost-efficient and energy-efficient. For example, micro-controllers featuring less RAM have also been used for sensor nodes that solely rely on energy scavenged from the environment [MM05].

- **Other sensor node operating systems**

  - **SOS.** The *SOS* operating system [HRS+05] is very similar to *Contiki*, in particular the design emphasis on dynamically loadable modules motivated by the need for code updates during deployment of a sensor network. Like *Contiki*, *SOS* consists of dynamically-loaded modules and a small kernel. The kernel implements messaging, dynamic memory and module loading and unloading at runtime. In *SOS*, modules are position independent binaries implementing a specific function. Applications are composed of one or more modules. Unlike *Contiki*, *SOS* does not (yet) provide multi-threading. The code size of the *SOS* core including a facility to distribute programs is about 20 KBytes and RAM usage is more than 2 KBytes. The ROM usage is comparable to *TinyOS* running *Deluge*, a reliable distribution protocol used to distribute e.g. OS images in a sensor network. The RAM usage of *TinyOS* and *Deluge* is, however, only a fourth of the RAM requirements of *SOS*.

- **kOS.** The *kOS* (*kind-of* or *kilobit*) operating system [BSSH05] is designed for iterative applications. In order to keep the duty cycle of applications below a certain threshold and thus to save power, the *kOS* scheduler adapts the periods of the applications. To keep the OS small and simple, the execution model used requires all applications to run to completion before other applications execute. *kOS* interacts with applications using a simple messaging interface. While these design decisions keep *kOS* small, they constraint its usage to iterative applications and do not enable e.g. more sporadic tasks such as dynamic loading of modules. The minimal memory footprint of *kOS* without any application is similar to the size of *Mantis*, requiring slightly above 12 KBytes ROM and about 500 Bytes RAM.
- **Timber.** *Timber* [KLN05] is a self-contained functional language based on an extension of Haskell. Since *Timber* is self-contained, it can be run without any other run-time or operating system and can thus be regarded as a stand-alone operating system also. In fact, *Timber* is used as the operating system for the Mulle sensor nodes [JVE$^+$04]. The language semantics are the most fundamental part of the OS and hence the run-time features such as scheduling, threading, memory management can be tailored to each individual application. In contrast to other operating systems for sensor nodes, *Timber* also supplies sufficient infrastructure for reactive concurrent programming and realizing real-time constraints.
- **DCOS.** The main objectives of the *DCOS* [5] (*Data Centric Operating System*) [DHH04], is to provide real-time guarantees, energy-efficient operation and online reconfigurability. *DCOS* uses an architecture which the authors call data centric. In this architecture, data is the main abstraction of events. The scheduled entities in *DCOS* are software components called `Data Centric Entities` (DCE). These entities produce data and they can be triggered by other data. While the system is running, *DCOS* adapts the system behavior by dynamically replacing DCEs and/or reconfiguring the data flow between DCEs.

4.1.3 Virtual Machines

Some systems use a virtual machine instead of an operating system running native machine code. Since the virtual machine code can be made smaller the energy consumption of transmitting the code over the network can be reduced. One of the drawbacks is the increased energy spent in interpreting the code for long running programs, the energy saved during the transport of the binary code is instead spent in the of execution of the code.

- **Maté**

  In order to provide run-time reprogramming for *TinyOS*, Levis and Culler have developed Maté [LC02] (also called *Maté Bombilla*), a virtual machine for *TinyOS* devices. Code for the virtual machine can be downloaded into the system at run-time. The virtual machine is specifically designed for the needs of typical sensor network applications.

  Maté is a byte-code interpreter running on *TinyOS*. Byte code is broken into capsules of 24 instructions, each one byte in length. This makes the capsules fit into a single *TinyOS* packet. Four types

[5]Also discussed in study 3.1.3 (Vertical Functions, Section 4.7)

of capsules exist: timer capsules, message receive capsules, message send capsules, and subroutine capsules. With the latter, programs can be longer than 24 instructions only. Capsules contain their type and a version number. *Maté* installs a received capsule if it contains a more recent version of the specified type than the currently installed one. Capsules are broadcasted with a single `forw` instruction.

Three execution contexts are known in *Maté*: clock timers, message receptions, and message send requests. Each context has its own two stacks: an operand stack and a return address stack. The first is used for all instructions handling data, the latter is used for subroutine calls. The maximum depth of the operand stack is 16, the maximum depth of the call stack is 8. The three types of events correspond to the execution contexts.

There are three operand types: values, sensor readings, and messages. Many instructions behave differently for different operands or combinations thereof.

Maté begins execution in response to an event and starts executing the first instruction of the corresponding context until it reaches the `halt` instruction. Contexts can run concurrently; their execution is interleaved at instruction granularity. *Maté* does not allow asynchronous operations. For example, it waits until a message is sent successfully or until an analog-digital conversion is done. Sending a message or sampling a sensor can be done as a single bytecode instruction. Messages are automatically routed to the destination; a task is automatically enqueued on arrival. Eight instructions can be defined by users, for example to do some complex data processing. They are implemented in *TinyOS*. Thus, a specially tailored version of *Maté* is to be built.

Maté on the Atmel AVR requires almost 40 KBytes ROM and more than three KBytes RAM and is thus larger than *SOS* and *TinyOS* using *Deluge* [HRS$^+$05]. The energy cost of the CPU overhead of a bytecode interpreter is outweighed by the energy savings of transmitting such concise program representations for a small number of executions. Native code is preferable for a large number of executions of the code since for simple instructions (e.g., logical operations) *Maté* takes 33 times more clock cycles than the native *TinyOS* implementation. Thus, any non-trivial mathematical operation is infeasible. Even though such operations can be implemented in native code and called as user instructions, they cannot be changed during runtime. An advantage is that all code runs in a sandboxed virtual environment and benefits from all of its safety guarantees.

Application Specific Virtual Machines (*ASVM*) [LGC05] is a enhancement of *Maté* and addresses its main limitations with respect to flexibility, concurrency, and propagation. *ASVM* supports a wide range of application domains, whereas Maté is designed for a single domain only. In *Maté*, only a single shared variable can be used. In *ASVM*, an operation component can register several shared variables, and the system ensures race-free and deadlock-free execution. The code propagation is not only done via broadcasts, but with a control algorithm based on Trickle to detect when code updates are really needed on other nodes.

Each *ASVM* consists of a template, which includes a scheduler, a concurrency manager, and a capsule store, and of extensions, which are the application-specific components that define a particular *ASVM*. Handlers as extension of *Maté*'s event contexts are code routines that run in response to system events. Operations are the units of execution functionality, divided into primitives and functions. A particular language is compiled to primitives which are, therefore, language specific. Functions are language independent and provided by the user to tailor an *ASVM* to a particular application domain. Capsules are again the units of code propagation, but can be longer than in Maté and are,

therefore, split into fragments during propagation. Building an *ASVM* involves connecting handlers and operations to the template.
Again, an *ASVM* instruction has an overhead of 400 cycles, thus making complex mathematical code in *ASVM* infeasible. But comparing to Maté, it is easier in *ASVM* to push expensive bytecode operations to native code using functions to minimize the amount of interpretation.

- **MagnetOS**
  MagnetOS [BBD+02, LRW+05] is a distributed operating system that simplifies the programming of ad hoc networking applications by making the entire network appear as a single virtual machine. It provides adaptation to the resource constraints, and changes in the network (e.g., topology, application behavior, available resources), increases the system longevity through good power utilization, supports nodes with heterogeneous resources and capabilities, and is highly scalable. Unlike *Maté*, *MagnetOS* targets larger platforms such as x86 laptops, Transmeta tablets, and StrongArm PocketPC devices.
  Applications consist of a set of event handlers that are executed in response to a system, sensor, or application-initiated occurrence. An event handler stores the instance variables and is free to move across nodes in the network. Execution consists of a set of event invocations that may be performed concurrently.
  The *MagnetOS* system provides the image of a virtual Java machine. Regular Java applications are partitioned into distributable components that communicate via events by a static partitioning service. Thereto, applications are rewritten at byte-code level. For example, object creations are replaced by calls to the *MagnetOS* runtime which selects an appropriate target node and constructs a new event handler at that location. Remote data accesses, lock aquisitions and releases, type-checking and synchronization instructions are converted as well.
  For object creation, *MagnetOS* provides at-most-once semantics. The system performs health checks using keep-alive messages only for long-running synchronous event invocations.
  Several algorithms in the core of the operating system decide when and where to move application components. All of them try to shorten the mean path length of data packets sent between components of an application by moving communicating objects to topologically closer nodes. *LinkPull* (formerly: *NetPull*) operates at physical link level and migrates components one hop at a time in the direction of greatest communication. *PeerPull* (formerly: *NetCenter*) operates at network level and is, therefore, able to migrate a component multiple hops at a time directly to the host with which a given object communicates most. *NetCluster* migrates a component to a randomly chosen node within the cluster it communicates with most. Finally, *TopoCenter* migrates components to a node such that the sum of migration cost and estimated future communication costs is minimized. Thereto, a partial view of the network is needed which is gathered along a packet's path by each node attaching its single-hop neighborhood.
  When *MagnetOS* decides to move an event handler, it sets a flag. The rewritten code detects the flag and checkpoints its current state. *MagnetOS* transports this state and resumes the computation at the destination. Application writers can also manually control the placement of components. A component can be strictly bound to a node or a starting node can be defined where the component is migrated to first and from where it can migrate further using the mechanisms described above.

- **SensorWare**

In *SensorWare* [BHS03], lightweight and mobile control scripts based on Tcl can be defined. Their replication and migration allows the dynamic deployment of distributed algorithms on sensor nodes, thus making them easily (re)programmable. *SensorWare* targets richer platforms than *Maté* like iPAQs since the framework is almost 180 kBytes in size. Although energy-efficiency is also a main consideration of the project, it is not clear how such a script-based solution can be implemented efficiently in sensor nodes with high resource limitations. On the other hand, code is more compact in *SensorWare*, it provides built-in multi-user support, and it is portable to other platforms.
SensorWare is built upon the operating system of the sensor nodes and uses its standard functions and services. The OS provides hardware abstraction. Control scripts rely completely on the *SensorWare* layer, while other static applications and services can use the standard functions and services of both *SensorWare* and the operating system.
Tcl is used as the basic scripting language in *SensorWare*. All *SensorWare* functions are defined as new Tcl commands, thus integrating fully into Tcl. Such commands abstract specific tasks, like communication with other nodes, or data sensing and filtering. Special commands allow the forwarding of the current program to other nodes while trying to avoid unnecessary code transfers by transmitting the code only if the script is not already running on the neighboring nodes.
SensorWare is an event-based language. But as *SensorWare* supports multi-threading, control scripts can use blocking waits until an event occurs. Examples for such events are the reception of a message, the expiration of a timer, or the availability of one or more sensor data.
There are two different types of task classes in the run-time environment of *SensorWare*: fixed tasks and platform-specific ones. The former are always included in every *SensorWare* implementation and handle system functions such as spawning of new scripts, surveillance of resource contracts, radio transmission and reception, etc. The latter depend on the hardware configuration and have to do with the specific types of sensors available at a given sensor node.
Threads in *SensorWare* are coupled with queues. Queues of scripts are receiving events. Queues for radio, sensors, times, etc. receive events of the device they are connected to as well as messages that declare interest in this event type. System Messages can be exchanged between system threads.
To address the problem of heterogeneity, any module or service (e.g., radio, sensing device, timer service) in *SensorWare* is represented as a virtual device. Every device implements a common interface with four commands to communicate with the device. Using these commands, it is possible to ask for information, to trigger an action, and to create and dispose event IDs. To facilitate porting the framework to other platforms, wrapper functions for several OS functions and hardware accesses are used.

4.1.4 Data Management Middleware

Operating systems usually provide only means to access the hardware – especially sensors – in a uniform way, but no means to manage the data flow. Thus, the actual data management functionality is located in the application layer. The *Data Management Middleware* is an abstraction layer between the sensor network and the application layer that provides access to information on a higher semantic level, including the storage, distribution, and querying of this information. So, it also allows the transparent addition of new sensor and new sensor types.

- **EnviroTrack**
  The main purpose of the *EnviroTrack* project [ABC+04, BNW+03] is to provide an efficient implementation for the class of applications that need to track the locations of entities in the environment. *EnviroTrack* distinguishes itself from traditional localization systems in the assumption of cooperative users that can wear beaconing devices that interact with location services in the infrastructure for the purposes of localization and tracking.
  Therefore, *EnviroTrack* provides a new abstraction based on context labels and tracking objects so that as the tracked entity moves, the identity and location of the sensor nodes in its neighborhood change. The programmer interacts with a changing group of sensors through a simple object interface.
  EnviroTrack works by providing the user with a programming abstractions that allow him to specify context types. This is done by activating a condition called `sense` which is used by sensors to join and leave the group of sensors in charge of tracking a given object. The second part of the context type definition is composed of a series of context variables that define an aggregate state. An aggregation is composed of readings taken by the various sensors of the group about a specific object. So, for example, this variable could be the average location for cars or a maximum temperature in a fire-detection scenario. For each context variable, applications define an aggregation function and two constants: a critical mass and freshness, which modify the aggregation process.
  The architecture of *EnviroTrack* is composed of the following elements:

  – **Pre-processor.** This component emits code that initializes the structures to track context labels and periodically calls the sense functions to allow entity discovery. It also translates the definition of context types.

  – **Group Management.** This protocol manages join and leaves of sensors for a specific entity. It also ensures that there is always at least one leader for the group.

  – **Aggregate State Computation.** Group members send their data to the leader periodically and the leader collects them, aggregates them and forwards them again to the appropriate location.

  – **Directory Services.** Context types are hashed to a location which serves as a directory for that type. Whenever new context labels are created, they register on the directory service their position which, if they move, is updated accordingly.

  EnviroTrack is able to track entities using an energy-efficient protocol and is able to provide fault-tolerance to message loss, leader failures and aggregate lost.

- **DSWare**
  Data Service Middleware (*DSWare*) [LSA03, LLS+04] is a specialized layer that performs the integration of various real-time data services for sensor networks. It provides a database-like abstraction to sensor networks in a similar way to *TinyDB* or *Cougar*.
  The distinguishing characteristic of *DSWare* is its support for group-based decision making and reliable storage to improve real-time system performance, reliability of aggregated results and reduction in communication overhead. In order to provide its functionality, *DSWare* is composed of the following components:

- **Data Storage.** This component takes care of storing data in the network according to the semantics associated with the data. It also provide some primitives to work with spatially correlated data, which has two advantages: it enables easy data aggregation and also makes it possible for the system to perform in-network processing.
- **Data Caching.** The purpose of this component is to provide multiple copies of popular data in the regions that need it, so that its availability is high and query execution can be performed in a faster way. There is a feedback mechanism that allows *DSWare* to decide on-the-fly whether or not copies of data should reside in frequent queries nodes. The control scheme uses the proportion of periodic queries, average response time, etc. to make its decisions.
- **Group Management.** Allows for the cooperation between various nodes in order to perform distributed computations, value comparison, etc. This component also supports the implementation of energy-saving actions like putting some nodes to sleep. For this, the formation and management of groups of sensor nodes is a crucial function of this component.
- **Data Subscription.** This component allows for the definition of continuous queries in the network. It defines the characteristics of the data feeding paths and uses stable traffic nodes to select the optimal routes. When many base stations make subscriptions for data from the same sensor node, the Data Subscription service puts copies of the data at intermediate nodes in order to save on communication costs. It is also able to merge several feeding paths into one if this saves communication costs.
- **Scheduling.** This component allows for the scheduling of services to all *DSWare* components. It provides two options: energy-aware and real-time scheduling.

• **TinyDB**
TinyDB [WMG04, MFHH02] is a project developed at the University of Berkeley in cooperation with Intel Research that aims at providing efficient data acquisition primitives for Sensor Networks. In the eyes of the *TinyDB* project, the most important type of query to be supported in sensor networks are continuous queries, since all other types can be mapped to a continuous one.
TinyDB, which has been implemented on top of *TinyOS* and runs on the MICA family of sensor nodes (also developed at UC Berkeley), defines an *Acquisitional Query Language* (*ACQL*), very similar in its structure to traditional SQL, that allows for the efficient retrieval of data within the network. Where possible, *TinyDB* performs in-network processing of data in order to reduce the size of transmissions. For example, the developers of *TinyDB* have developed *TAG*, a *Tiny AGgregation* engine that supports arbitrary decomposable aggregation functions using a generic and extensible framework.
For *ACQL*, all queries create a continuous data stream that can be mapped to each sensor node. *TinyDB* assumes the presence of a sink that is able to process the *ACQL* statement and translate it into an efficient binary representation that can be processed by each sensor. In *TinyDB*, the entire sensor network is a single table where the columns contain all the attributes in the network and the rows specify the individual sensor data. Using this data model, and special language capabilities explicitly designed for sensor networks, *TinyDB* is able to process the following types of queries:

- **Event-based queries.** Which have a precondition usually given by an event that triggers the execution of the query.

  - **Storage-based queries.** Which are able to perform caching and intermediate storage of data at specific locations in the network.
  - **Lifetime-based queries.** Which run for a specified amount of time in the network, collecting the necessary data.

  Although *TinyDB* provides a nice abstraction to retrieve data from a sensor network, it is only able to support applications that obtain data from the sensors and processes it outside the sensor network. For any other type of processing, it is necessary to create *TinyOS* components that provide the required functionality.

- **Cougar**
  Cougar [YG02] is a project developed at Cornell University whose philosophy dictates that monitoring is best described in a declarative manner. Since for the Cougar project, the sensor network is the database itself, they provide abstractions to represent the different devices in the sensor network as a database.
  The distinguishing feature of *Cougar* is the use of *Abstract Data Types* (ADTs) and virtual relations, which are a tabular representation of the functions that define the type of data available at different sensors. Using this information, the Sensor Network is seen as one large table that contains the data to be queried by the user. *Cougar* assumes the presence of a front-node that implements a full-fledged database server and translates the queries issued by the user into a format that can be understood by each sensor node to answer the query. Each sensor contains an instance of a mini-server that is able to understand these messages and return the appropriate answer.
  The mini-server supports synchronous queries whose results need to be returned immediately and on-demand, and asynchronous queries, used to monitor threshold events. Using this distinction, *Cougar* is able to answer the following types of queries:

  - **Historical queries.** Which usually refer to aggregate queries over historical data, such as: *"For each rainfall sensor, display the average level of rainfall for 1999"*.
  - **Snapshot queries.** Which refer to the values of data at a given point in time, such as: *"Retrieve the current rainfall level for all sensors in Tompkins County"*.
  - **Long-running queries.** Which refer to the values of data over a certain time interval, such as: ‘*‘For the next 5 hours, retrieve every 30 seconds the rainfall level for all sensors in Tompkins County"*.

  Although the idea of providing a well-known abstraction to represent the nodes in a sensor network is interesting, there are certain abstractions like events or publish/subscribe mechanisms that cannot be mapped to the classic view. However, *Cougar* is able to provide distribution transparency for queries issued to the sensor network.

4.1.5 Adaptive System Software

Since the requirements to Cooperating Objects or the system environment can change significantly during the lifetime and a constant manual adjustment is too costly, several systems have been developed that perform automatic adaptation.

- **MiLAN**
  MiLAN [HMCP04] is a “proactive” middleware that aims at providing a bridge between the capabilities of current middleware platforms and the need for proactive rules that have an effect on the network and the sensors themselves. It achieves its goal by allowing sensor network applications to specify their quality needs and subsequently making adjustments on specific properties of the sensor network to meet these needs.
  MiLAN supports data-driven applications that collect and perform an analysis of the data from the environment. The quality of data is affected by noise, redundancy and the capabilities of the sensors to detect this information. Furthermore, *MiLAN* also supports state-based applications which, being of dynamic nature, change over time the specific needs on the quality of acquired data.
  The three types of information used by *MiLAN* to perform adaptation are:

  – Data about the QoS level defined by the application.
  – Data about the overall performance of the system and about the user.
  – Data about the sensor network, such as available resources, state of sensors, energy level and channel bandwidth.

  Using this information, *MiLAN* is able to adapt the configuration of the sensor network to optimize the functionality of the system by proactively specifying which sensors need to send data and the role each sensor should play in the overall scheme.

- **Impala**
  Impala is a middleware system that was designed as part of the *ZebraNet* mobile sensor network and its architecture allows for application modularity, adaptivity, and reparability in wireless sensor networks [LM03].
  Impala supports multiple applications by adopting an event-based modular programming model and providing a friendly programming interface. It also features a lightweight system layer that performs on-the-fly application adaptation based on parameters and device failures which allows to improve the performance, reliability and energy-efficiency of the system. The modular application structure is used to perform application updates in small, modular pieces over the radio, similar as in *Contiki* and *SOS*. However, the system is implemented on devices similar to PDAs rather than simple sensor nodes.
  Impala adopts a layered approach in which the upper layer comprises the application protocols and programs. The lower layers are composed of the following middleware agents:

  – The *application adapter*, which performs adaptation on the application protocols at runtime to adapt to the changing environmental conditions.
  – The *application updates*, which receive and transmit updates using wireless technology in order to install new code versions on a node.
  – The *event filter*, which is responsible for capturing and dispatching the appropriate messages to the other two layers.

  Although *Impala* supports a certain degree of adaptation, it does not address the issue of heterogeneity which, in other middleware platforms, play a very important role.

- **TinyCubus**
  The overall architecture of *TinyCubus* [MLM+05, MMLR05] mirrors the requirements imposed by the heterogenity of applications and the hardware they operate on. *TinyCubus* has been developed with the goal of creating a generic reconfigurable framework for sensor networks.
  TinyCubus is implemented on top of *TinyOS* [HSW+00] using the nesC programming language [GLv+03], which allows for the definition of components that contain functionality and algorithms. *TinyOS* is primarily used 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.
  TinyCubus itself consists of three parts: the *Tiny Cross-Layer Framework*, the *Tiny Configuration Engine*, and the *Tiny Data Management Framework*.

  – **Tiny Data Management Framework.** The *Tiny Data Management Framework* 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 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 *Tiny Data Management Framework* contains a `Cubus` which combines optimization parameters, such as energy, communication latency or bandwidth; application requirements, such as reliability or consistency level; and system parameters, such as mobility or network 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 *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.

  – **Tiny Cross-Layer Framework.** The *Tiny Cross-Layer Framework* provides a generic interface to support the parametrization of components that use cross-layer interactions. Strict layering is not practical for wireless sensor networks because it might not be possible then 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. On the other hand, cross-layer interactions can influence the desirable properties of the software architecture, such as modularity, negatively. For example, if cross-layer interactions are not applied in a controlled way, it might not be possible to exchange a module without major changes to others. Therefore, in the *Tiny Cross-Layer Framework* a state repository is used to store the cross-layer data of all components, i.e., the components do not interact directly with each other. Thus, architectural properties are better preserved than with the unbridled use of cross-layer interactions.
  Other examples for cross-layer interactions are callbacks to higher-level functions, such as the ones provided by the application developer. The *Tiny Cross-Layer Framework* also supports this form of interaction. To deal with callbacks and dynamically loaded code, *TinyCubus* extends

the functionality provided by *TinyOS* to allow for the dereferencing and resolution of interfaces and components.

– **Tiny Configuration Engine.** In some cases parametrization, 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 efficient 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. The topology manager uses a generic specification language and a distributed role assignment algorithm 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.

4.1.6 Summary and Evaluation

This section has presented the state of the art of system architectures for single nodes comprising operating systems, virtual machines, data management middleware and adaptive system software. From the operating system point of view, the dominant operating systems *TinyOS*, *Contiki* and *Mantis* include all functionality required except for real-time support which has been addressed in recent operating systems such as *Timber* and *DCOS*. While we believe that most functionality is available in today's operating systems, the main issue seems to be the programmability of these systems, which relates to the issues discussed in section 3.
Virtual machine code is more compact than native code which reduces the energy consumption when sending code updates through the network compared to native code even when there is support for loadable modules as in *Contiki* or *SOS*. However, code interpretation is more expensive using virtual machines. This trade-off (which is apparently application-dependent) has not been evaluated thoroughly.
The view of a sensor network as a database might be good for an external user querying the network, for an internal application this approach seems to be of high costs. Other *Data Management Middleware* schemes are too application specific even if it includes several generic parts that are useful for many other scenarios as well.
The need for adaptive system software is obvious and several approaches exist. While *MiLan* focuses on the quality of the sensor data, *Impala* and *TinyCubus* deal with the optimization of the application itself. Unfortunately, it is hard to model *MiLan*'s functionality in the other two systems.

4.2 System Architecture: Interaction of Nodes

4.2.1 Introduction

In this section we focus on system architecture and topics related to cooperation among nodes. We can roughly abstract two main set of functionalities defining how nodes interact:

1. *Low-level functionalities*, including tasks corresponding to physical, link, routing, and transport layers;
2. *High-level functionalities*, including coordination and support, clustering, timing and localization, addressing, lookup, collaboration, failure detection, and security.

An approximate mapping of the above functionalities onto the traditional ISO/OSI protocol layers is shown in Figure 4.

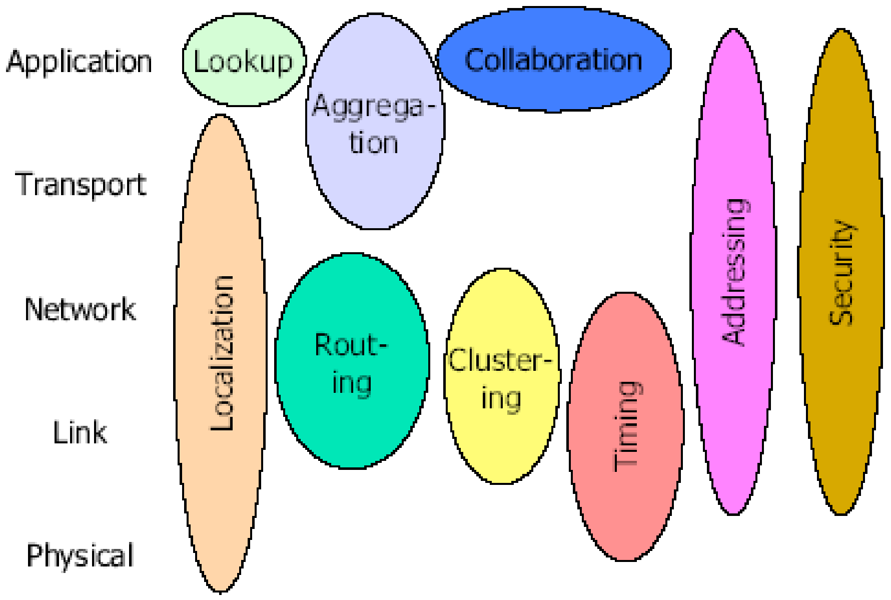

Figure 4: Approximate mapping of the interaction functionalities to ISO/OSI protocol layers.

This mapping shows a peculiarity of the sensor network, namely most of the functionalities extend over and depend on several traditional protocol layers [ASSC02]. This is because sensor networks have to provide functionalities that are not present in traditional networks. Furthermore, the efficiency constraints imposed by sensor networks' limited resources imply that any strictly layered approach, while possible, is likely to produce suboptimal solutions. On the other hand, a complete monolithic approach is unlikely to be manageable in complexity. Hence, a suitable level of integration has to be found.

4.2.2 Communication Models

Conceptually, communication within a sensor network can be classified into two categories [TAGH02]: application and infrastructure. The network protocol must support both these types of communication.

Application communication relates to the transfer of sensed data with the goal of informing the observer about the phenomena. Within application communication, there are two models: cooperative and non-cooperative. Under the cooperative sensor model, sensors communicate with other sensors to realize the observer interest. This communication is beyond the relay function needed for routing. In-network data processing is an example of co-operative sensors. Non-cooperative sensors do not cooperate for information dissemination; the co-operation is strictly limited to packets relay in multi-hop networks. Infrastructure communication refers to the communication needed to configure, maintain and optimize operation. More specifically, because of the ad hoc nature of sensor networks, sensors must be able to discover paths to other sensors of interest to them and have to be able to discover paths to the observer regardless of sensor mobility or failure. Thus, infrastructure communication is needed to keep the network functional, ensure robust operation in dynamic environments, as well as to optimize its overall performance. We note that such infrastructure communication is highly influenced by the application interests since the network must reconfigure itself to best satisfy these interests. As infrastructure communication represents the overhead of the network operation, it is important to minimize this communication while ensuring that the network can support efficient application communication. In sensor networks, an initial phase of infrastructure communication is needed to set up the network. Furthermore, if the sensors are energy-constrained, there will be additional communication for reconfiguration. Indeed the role of each node will be re-negotiated on the basis of its remaining energy. For example the role of cluster head in clustered networks is typically rotated in order to balance the energy consumption.
Similarly, if the sensors are mobile or the observer interests are dynamic, additional communication is needed for path discovery/reconfiguration. For example, in a clustering protocol, infrastructure communication is required for the formation of clusters and cluster-head selection; under mobility or sensor failure, this communication must be repeated (periodically or upon detecting failure). Finally, infrastructure communication is used for network optimization. Consider the Frisbee model, where the set of active sensors follows a moving phenomenon to optimize energy efficiency. In this case, the sensors wake up other sensors in the network using infrastructure communication. Sensor networks require both application and infrastructure communication. The amount of required communication is highly influenced by the networking protocol used. Application communication is optimized by reporting measurements at the minimal rate that will satisfy the accuracy and delay requirements given known sensor capabilities and the quality of the paths between the sensors and the observer. The infrastructure communication is generated by the networking protocol in response to application requests or events in the network. Investing in infrastructure communication can reduce application traffic and optimize overall network operation.

4.2.3 Network Dynamics

A sensor network forms a path between the phenomenon and the observer. The goal of ta sensor network protocol is to create and maintain this path (or multiple paths) under dynamic conditions, by means of interaction of nodes, while meeting the application requirements of low energy, low latency, high accuracy, and fault tolerance.
The way in which these communication paths are established and maintained, and thus the way in which nodes interact, strongly depends on the network dynamics. Network dynamics can be roughly classified as: static sensor networks and mobile sensor networks.

- **Static Networks.**
  In static sensor networks, the communicating sensors, the observer and the phenomenon are all static. An example is a group of sensors spread for temperature sensing. In this type of network, sensor nodes require an initial set-up infrastructure communication to create the path between the observer and the sensors with the remaining traffic mainly being for sake of application communication. Re-configuration can still occur for task re-assignment or failures due to energy consumption.

- **Dynamic Networks.**
  In dynamic sensor networks, either the sensors themselves, the observer, or the phenomenon are mobile. Whenever any of the sensors associated with the current path from the observer to the phenomenon moves, the path may fail. In this case, either the observer or the concerned sensor must take the initiative to rebuild a new path. During initial set-up, the observer can build multiple paths between itself and the phenomenon and cache them, choosing the one that is the most beneficial at that time as the current path. If the path fails, another of the cached paths can be used. If all the cached paths are invalid, then the observer must rebuild new paths. This observer-initiated approach is a reactive approach, where path recovery action is only taken after observing a broken path. Another model for rebuilding new paths from the observer to the phenomenon is a sensor-initiated approach. In a sensor-initiated path recovery procedure, path recovery is initiated by a sensor that is currently part of the logical path between the observer and the phenomenon and is planning to move out of range. Dynamic sensor networks can be further classified by considering the motion of the components. This motion is important from the communications perspective since the degree and type of communication is dependent on network dynamics.

  – **Mobile observer.** In this case the observer is mobile with respect to the sensors and phenomena. For example, a plane might fly over a field periodically to collect information from a sensor network. A model that is well-suited to this case is the Data Mules model [RCSB03]. The MULE architecture provides wide-area connectivity for a sparse sensor network by exploiting mobile agents such as people, animals, or vehicles moving in the environment. The system architecture comprises of a three-tier layered abstraction.The top tier is composed of access points/central repositories, which can be set up at convenient locations where network connectivity and power are present. These devices communicate with a central data warehouse that enables them to synchronize the data that they collect, detect duplicates, as well as return acknowledgments to the MULEs. The intermediate layer of mobile MULE nodes provides the system with scalability and flexibility for a relatively low cost. The key traits of a MULE are large storage capacities (relative to sensors), renewable power, and the ability to communicate with the sensors and networked access points. MULEs are assumed to be serendipitous agents whose movements cannot be predicted in advance. However as a result of their motion, they collect and store data from the sensors, as well as deliver acks back to the sensor nodes. In addition, MULEs can communicate with each other to improve system performance. The bottom tier of the network consists of randomly distributed wireless sensors. Work performed by these sensor nodes should be minimized as they have the most constrained resources of any of the tiers. Depending on the application and situation, a number of tiers in our three-tier abstraction could be collapsed onto one device. As data MULEs perform the collection of information to and from the sensor nodes when they are in the sensors radio range, sensor nodes can be very simple (they are only required to sense data and communicate them to the data MULE when in range). This in turn may reduce complexity, thus cost, of sensor nodes, enabling their adoption in very large scale systems. The price to pay in case a data MULEs-like solution is adopted, is an energy-latency trade-off. Complexity is shifted from the sensor nodes to the MULEs, the energy consumption is reduced as nodes only have to communicate the data they generate, but communication may experience high latencies as sensors have to wait for a MULE to pass by before the sensed data is communicated.

– **Mobile sensors.** In this case, the sensors are moving with respect to each other and the observer. For example, consider traffic monitoring implemented by attaching sensors to veichles [SSW+05]. If the sensors are co-operative, the communication paradigm imposes additional constraints such as detecting the link layer addresses of the neighbors and constructing localization and information dissemination structures. As sensor nodes are energy-constrained devices, their mobility can be foreseen as the (often uncontrollable) mobility of the mobile devices they are attached to (users, cars etc.)
– **Mobile phenomena.** In this case, the phenomenon itself is moving. A typical example of this paradigm is sensors deployed for animal detection. In this case the infrastructure level communication should be event-driven. Depending on the density of the phenomena, it will be inefficient if all the sensor nodes are active all the time. Only the sensors in the vicinity of the mobile phenomenon need to be active. The number of active sensors in the vicinity of the phenomenon can be determined by application specific goals such as accuracy, latency, and energy efficiency. A model that is well-suited to this case is the Frisbee model [CEE+01].

4.3 Architectures and Functionalities summary

In this section we sketch a general architecture design based on the considerations made in the above sections. Furthermore we briefly discuss a set of functionalities to allow a flexible, but efficient interaction among nodes to reach a common goal.
We can identify two main layers of abstraction: the sensor and networking layer, and the distributed services layer.

- The sensor and networking layer is made of sensor nodes and network protocols. Ad-hoc routing protocols allow messages to be forwarded through multiple sensor nodes taking into account the mobility of nodes and the dynamic change of topology. Communication protocols must be energy-efficient because of the limited energy and computational resources.

- The distributed services layer is made of distributed services for the mobile sensor applications. Distributed services co-ordinate with each other to perform decentralized services. Resources might be replicated for higher availability, efficiency and robustness. A Lookup Service supports mobility, instantiation, and reconfiguration. Finally the Information Service deals with aspects of collecting data. This service allows vast quantities of data to be easily and reliably accessed, manipulated, disseminated, and used by applications.

Applications run on the top of this architecture and exploit the functionalities provided be the sensor and networking layer and distributed services layer, see Figure 5.

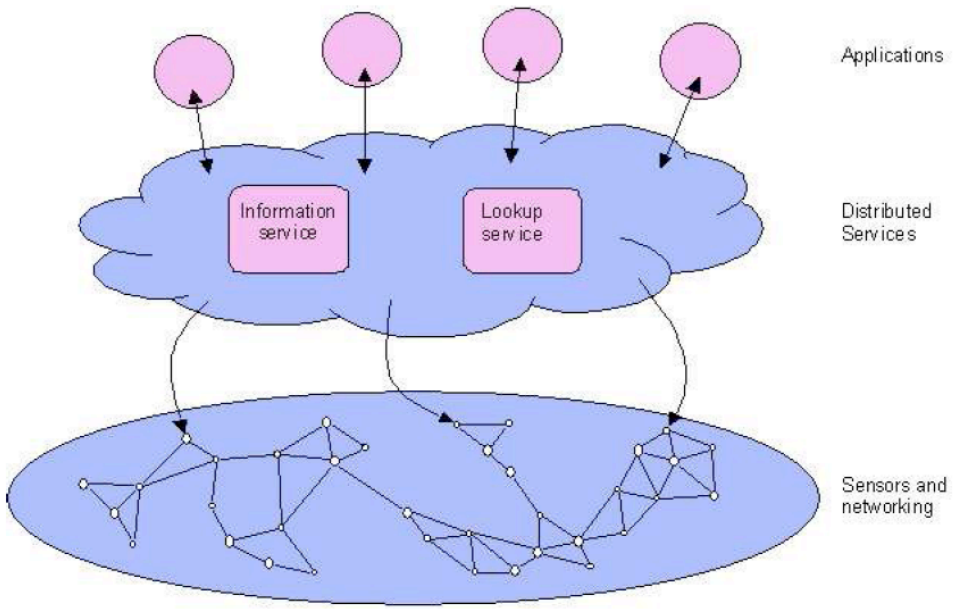

Figure 5: Architecture overview.

We can identify two distinct phases which bring the sensor network in a fully operational mode: The initialization phase and the operation phase.

- **Initialization phase.**
  When new sensor nodes are added to the sensor network, they should learn about the capabilities and functions of other sensors and work together to perform co-operative tasks and networking functionalities.
  The initialization phase is further divided in the following phases:
  - **Discovery phase.** In this phase nodes explore their environment and establish contact with neighbors using ad-hoc networking techniques. The nodes perform a distributed coordination protocol in order to find a suitable networking topology that allows all nodes to communicate.
  - **Synchronisation phase.** In this phase nodes try to synchronise with neighboring nodes in order to have some common notion of time. This is required for example to enable dynamic power management strategies in networking and data processing protocols, to use TDMA-like MAC protocols, to be able to identify the time at which events occur according to a common reference time.
  - **Positioning phase.** In this phase nodes try to find out their relative and possibly geographic positions.
  - **Registration phase.** In this phase, nodes can communicate and there is a common agreement on time and position/topology. A node connects to the lookup service to register its presence, its position, and its capabilities.

  - **Configuration phase.** In this phase the lookup service configures a sensor node or the node configures itself to perform a specific task.

  Once these phases are completed, the sensor network becomes operational. Sensors may move and may be required to perform different tasks. This implies that some of the above-mentioned phases would be performed again in re-configuration tasks.

- **Operation phase.**
  Once the sensor network has been initialized, it will perform the tasks as specified in the initialization phase. Sensor nodes are organized in ad-hoc and highly dynamic networks. They must react to mobility, changes in task assignment, and network and device failures. Therefore, each sensor node must be autonomous and capable of organising itself in the community of sensors to perform coordinated activities with global objectives. This is performed via self-organizing in possibly hierarchical structures, topology control protocols, protocols addressing and exploiting mobility in sensor networks, dynamic routing protocols etc [6].

5 Conclusions and Future Work

This section presents summaries of the different parts of this study, namely for programming models and system architectures in form of node internals and interaction of nodes. An overview of the corresponding part is given followed by a discussion on identified trends and open issues.

5.1 Programming Models

In order to support the development, maintenance, deployment and execution of applications for Cooperating Objects, an appropriate abstraction sitting between the operating system and the application itself need to be provided to the programmer [RKM02]. In the last couple of years different approaches for the design and development of such abstractions were presented in the research community and some of them evolved to running systems, most of which where surveyed in Section 3.2.
Most of the discussed approaches were designed for a specific application scenario or were tailored to some specific design goals and appear thus able to comply with only a subset of the requirements listed in Section 3.1. We can also conclude that for complex systems like networks of Cooperating Objects, the design a "unique" suitable programming abstraction, able to comply with the requirements and constraints of the many envisioned application scenarios appears as an extremely challenging task.

Trends. Most of the surveyed programming paradigms (e.g., database approach, agent-based approach, event-based approach) are not new, but required significant adaptation for being used for Cooperating Objects. These approaches differ with respect to ease of use, expressiveness, scalability, overhead, etc., as outlined in section 3.3. Some paradigms, like the database or the group-approaches, allow an easy way to programming the network as a single (virtual) entity, but the definition and completion of more complex tasks than just simple sensing is not yet appropriately supported by such approaches. Virtual machines, like

[6] An extensive survey of the different protocols proposed in the literature for performing such tasks has been reported in WP3, task 3.1.2.

Maté or *MagnetOS* give priority to energy-saving issues and are well suited for hiding device heterogeneity to the programmer, but implicate cumbersome local code interpretation. The *Generic Role Assignment* and the *Virtual Market* approaches are tailored to provide a general and easy-to-use framework for supporting task assignment and device coordination in networks of Cooperating Objects and are still object of active research.

Some open issues. Since most of the investigated programming abstraction are only partially able to comply with the several requirements posed by complex systems like network of Cooperating Objects, we believe that the research community should try to better understand which models and abstractions are the most appropriate for the different classes of devices covered by the notion of Cooperating Objects.
More effort should also be put on the provision of easy-to-use programming primitives, since most systems still require experienced programmers. Since future applications envision cooperation among devices which may strongly differ in terms of dimension, power and computing resources, mobility, etc., we also recognize the need of further research on paradigms that allow an easy and efficient programming of networks that incorporate very heterogeneous devices. Scalability issues should also gain more attention, since it is still unclear if the existing approach will be perform satisfactory for very large networks of Cooperating Objects. Finally, we believe that the design and development of adequate tools for debugging at application level should also be considered as a fundamental issue for the success of a programming abstraction for Cooperating Objects.

5.2 Node Internals

The main tasks of operating systems for Cooperating Objects is to provide an abstract interface to the underlying hardware and to schedule system resources. The main challenges are the small memory footprint and limited energy budget of the nodes. Therefore, scaled-down versions of traditional operating systems have been successful for larger devices such as IPAQs but not for extremely resource-constrained sensor network nodes. For these nodes, operating systems from scratch have been developed and are in use.
Other abstractions for Cooperating Objects have been developed: *Virtual Machines* provide a high-level programming language that allows to write complex programs with only a few commands. *Data Management Middleware* offers a uniform view to the data gathered in a Cooperating Object and its processing, storage, and distribution. *Adaptive System Software* frees the user from adjusting the application to every possible system condition.

Trends. *TinyOS* is the dominant operating system for sensor networks. *TinyOS*' most outstanding feature is its small memory footprint but it does not have support for loadable modules, its concurrency model is based on events only and *TinyOS* is hard to program. Therefore, newer operating system such as *Contiki*, *Mantis* and *SOS* have support for loadable modules and offer more sophisticated concurrency models which slightly increases the memory footprint of the systems but simplifies OS and application development. Fairly recent operating systems such as *DCOS* and *Timber* also address the aspect of real-time support not found in the earlier, dominating operating systems.
Virtual Machines also allow for efficient code updates since virtual machine code is more compact than native code which reduces the energy consumption when sending code updates through the network

compared to native code even when there is support for loadable modules as in *Contiki* or *SOS*. However, code interpretation is more expensive using virtual machines.
Virtual Machines, *Data Management Middleware*, and *Adaptive System Software* provide several different abstractions and functionality that help to simplify the programming and operation of cooperating objects. This is still a hot topic. *Data Management Middleware* first concentrated on database abstractions for the whole network, but they could only be used from the outside. Currently, middleware for applications with common characteristics (e.g., location tracking, real-time data services) are developped. Adaptive systems also concentrated on one aspect at first, e.g. to ensure the quality of the sensor readings. This is currently extended to other algorithms, and finally the whole system is to be adapted.

Some open issues. Real-time aspects will become more important in sensor networks in the future. While there have been some efforts, real-time operating systems for sensor networks can still be seen as an open issue.
While we believe that most functionality is available in today's operating system, the main issue seems to be the programmability of these systems, in particular for highly optimized systems such as *TinyOS* and *Contiki*. Abstractions such as *Protothreads* found in the *Contiki* OS are a step in the right direction, but more work needs to be done as also pointed out above in the discussion on programming models.
The support by *Data Management Middleware* are another step, but either the approaches are not usable inside the network of objects, or they are too application specific and, thus, not interoperable. Therefore, a general concept for the basic functionality of such a middleware is clearly needed.
Since code updates are essential for Cooperating Objects with long-term installations, every system will support updates in the future. For *Virtual Machines*, updates are most simple since the script or byte-code to interpret can be loaded from every place. For this reason, all of the VMs mentioned have this capability. Updates are more difficult for native code, since function calls and variable access have to be performed at a low-level. Either the user is willing to accept overhead for indirections, or the new code has to be fully integrated into the existing code. No general solutions exist here.
With *Virtual Machines*, *Data Management Middleware*, and *Adaptive System Software*, several approaches exist that abstract from different parts of a Cooperating Object. It has not been studied yet if and how these approaches can be combined to a single overall framework for Cooperating Objects.

References

[ABC$^+$03] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: system support for multimodAl NeTworks of in-situ sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 50–59, 2003.

[ABC$^+$04] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 582–589, Washington, DC, USA, 2004. IEEE Computer Society.

[ASSC02] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, March 2002.

[BBD$^+$02] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the Need for System-Level Support for Ad Hoc and Sensor Networks. *Operating System Review*, 36(2):1–5, April 2002.

[BCPB04] Michel Banâtre, Paul Couderc, Julien Pauty, and Mathieu Becus. Ubibus: Ubiquitous Computing to Help Blind People in Public Transport. In *6th International Symposium Mobile Human-Computer Interaction (Mobile HCI 2004)*, pages 310–314, Glasgow, UK, September 2004.

[BDR97] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[BGR05] Z. Benenson, N. Gedicke, and O. Raivio. Realizing Robust User Authentication in Sensor Networks. In *First Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[BGS00] P. Bonnet, J. E. Gehrke, and P. Seshadri. Querying the Physical World. *IEEE Journal of Selected Areas in Communications*, 7(5):10–15, October 2000.

[BHS03] A. Boulis, C.C. Han, and M. B. Srivastava. Design and Implementation of a Framework for Programmable and Efficient Sensor Network. In *MobiSys 2003*, San Franscisco, USA, May 2003.

[BIK$^+$02] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode. Cooperative Computing for Distributed Embedded Systems. In *22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 227–236, July 2002.

[BIK$^+$04] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial Programming Using Smart Messages: Design and Implementation. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 690–699, Hachioji, Tokyo, Japan, March 2004.

[Bir93] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):3753, December 1993.

[BNW$^+$03] Brian M. Blum, Prashant Nagaraddi, Anthony D. Wood, Tarek F. Abdelzaher, Sang Hyuk Son, and Jack Stankovic. An Entity Maintenance and Connection Service for Sensor Networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, pages 201–214, San Francisco, CA, USA, May 2003. USENIX.

[BSSH05] M. Britton, L. L. Shum, L. Sacks, and H. Haddadi. A Biologically-Inspired Approach to Designing Wireless Sensor Networks. In *2nd European Workshop on Wireless Sensor Networks*, Istanbul, Turkey, January 2005.

[bt:01] Bluetooth SIG, Specifications of the Bluetooth System - core,version 1.1, February 2001.

[BW99] M. Banâtre and F. Weis. SIS: a new paradigm for mobile computer systems. In *Information Society Technologies Conference*, Helsinki, Finland, November 1999.

[CB03] P. Couderc and M. Banâtre. Ambient computing applications: an experience with the SPREAD approach. In *36th Annual Hawaii International Conference on System Sciences (HICSS'03)*, Big Island, Hawaii, January 2003.

[CEE$^+$01] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat Monitoring: Application Driver for Wireless Communications Technology. In *Proc. ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, April 2001.

[CSS$^+$91] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[DGV04] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *First IEEE Workshop on Embedded Networked Sensors*, 2004.

[DHH04] Stefan Dulman, Tjerk Hofmeijer, and Paul Havinga. Wireless sensor networks dynamic runtime configuration. In *Proceedings of ProRISC 2004*, 2004.

[Dic73] L. I. Dickman. Small Virtual Machines: A Survey. In *Workshop on Virtual Computer Systems*, pages 191–202, Cambridge, Massachusetts (USA), June 1973.

[DSV05] A. Dunkels, O. Schmid, and T. Voigt. Using Protothreads for Sensor Network Programming. In *First Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[eCo] eCos free open source runtime system. Web page: http://www.ecoscentric.com/. Visited 2005-06-27.

[ECPS02] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[EGHK99] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. *International Conference on Mobile Computing and Networks (MobiCOM 99)*, August 1999.

[FR05] Christian Frank and Kay Römer. Algorithms for Generic Role Assignment in Wireless Sensor Networks. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, San Diego, CA, USA, November 2005. To appear.

[GLv+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, San Diego, California, USA, 2003.

[HMCP04] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to Support Sensor Network Applications. *IEEE Network*, pages 6–14, January 2004.

[HRS+05] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. SOS: A dynamic operating system for sensor networks. In *MobiSys 2005*, Seattle, USA, June 2005.

[HSW+00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS 2000*, Cambridge, USA, November 2000.

[iri] IrisNet: Internet-scale Resource-Intensive Sensor Network Service.

[JVE+04] J. Johansson, M. Völker, J. Eliasson, Å. Östmark, P. Lindgren, and J. Delsing. Mulle: A minimal sensor networking device - implementation and manufacturing challenges. In *IMAPS Nordic 2004*, 2004.

[KBM+00] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V.Krishnan, H. Morris, J. Schettino, and B. Serra. People, Places, Things: Web Presence for the Real World. Technical Report TR HPL-2000-16, Internet and Mobile Systems Laboratory, HP Laboratories Palo Alto, February 2000.

[KLBF04] J. Koberstein, N. Luttenberger, C. Buschmann, and S. Fischer. Shared Information Spaces for Small Devices: The SWARMS Software Concept. In *Workshop on Sensor Networks at Informatik 2004*, Ulm, Germany, September 2004.

[KLN05] M. Kero, P. Lindgren, and J. Nordlander. Timber as an RTOS for Small Embedded Devices. In *First Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[KST99] G. Kortuem, Z. Segall, and T. G. Cowan Thompson. Close Encounters: Supporting Mobile Collaboration through Interchange of User Profiles. In *1st International Symposium on Handheld and Ubiquitous Computing (HUC99)*, Karlsruhe, Germany, November 1999.

[LC02] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS X*, San Jose, USA, October 2002.

[LGC05] Philip Levis, David Gay, and David Culler. Active Sensor Networks. In *Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, Boston, USA, May 2005.

[LLS$^+$04] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. *Telecommunication Systems*, 26(2-4):351–368, June 2004.

[LM03] Ting Liu and Margaret Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, June 2003.

[LMG$^+$04] Philip Levis, Sam Madden, David Gay, Joe Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyos. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 29–42, San Jos, USA, March 2004.

[LMRV00] M. Langheinrich, F. Mattern, K. Rómer, and H. Vogt. First Steps Towards an Event-Based Infrastructure for Smart Things. In *Ubiquitous Computing Workshop (PACT 2000)*, Philadelphia, USA, October 2000.

[LR05] Ciaran Lynch and Fergus O Reilly. Pic-based tinyos implementation. In *Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 93–107, January 2005.

[LRW$^+$05] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin G&#252;n Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162, New York, NY, USA, 2005. ACM Press.

[LSA03] S. Li, S. H. Son, and J. A.Stankovic. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In *IPSN 2003*, Palo Alto, USA, April 2003.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[Mad02] S.R. Madden. Query Processing for Streaming Sensor Data (Ph.D. Qualifying Exam Proposal), May 2002.

[MFHH02] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TAG: A Tiny Aggregation Service for Ad-Hoc Sensor Networks. In *OSDI*, Boston, USA, December 2002.

[MFHH03] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor For Sensor Networks. In *SIGMOD*, San Diego (CA), USA, June 2003.

[MKL$^+$04] G. Mainland, L. Kang, S. Lahaie, D. Parkes, and M. Welsh. Using Virtual Markets to Program Global Behavior in Sensor Networks. In *11th ACM SIGOPS European Workshop*, Leuven, Belgiun, September 2004.

[MLM+05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A Flexible and Adaptive Framework for Sensor Netwo rks. In Erdal Çayırcı Şebnem Baydere, and Paul Havinga, editors, *Proceedings of the 2nd European Workshop on Wireless Sensor Networks*, pages 278–289, Istanbul, Turkey, January 2005.

[MM05] S. Mahlknecht and M.Roetzer. Energy supply considerations for self-sustaining wireless sensor networks. In *Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 93–107, January 2005.

[MMLR05] Pedro José Marrón, Daniel Minder, Andreas Lachenmann, and Kurt Rothermel. TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks. *Information Technology*, 47(2):87–97, April 2005.

[MW96] T. Mullen and M.P. Wellman. Some Issues In The Design of Market-Oriented Agents. In M. Wooldridge, J. Mueller, and M. Tambe (eds.), editors, *Intelligent Agents II: Agent Theories, Architectures, and Languages*. Springer-Verlag, 1996.

[NKG+02] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An architecture for enabling sensor-enriched Internet service. Technical report, Intel Research Pittsburgh, December 2002.

[NR96] K. Nagao and J. Rekimoto. Agent Augmented Reality: A Software Agent Meets the Real World. In *2nd International Conference on Multi-Agent Systems (ICMAS-96)*, pages 228–235, San Jos, USA, 1996.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[PMR99] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *21st International Conference on Software Engineering*, pages 368–377, Los Angeles, USA, May 1999.

[qnx] QNX RTOS. Web page: http://www.qnx.org. Visited 2005-06-27.

[R04̈] K. Römer. Programming Paradigms and Middleware for Sensor Networks. *GI/ITG Workshop on Sensor Networks*, pages 49–54, February 2004.

[RCSB03] Sushant Jain Rahul C Shah, Sumit Roy and Waylon Brunette. Data mules: Modeling a three-tier architecture for sparse sensor network. In *IEEE Workshop on Sensor Network Protocols and Applications (SNPA)*, May 2003.

[RFMB04] K. Römer, C. Frank, P.J. Marron, and C. Becker. Generic Role Assignment for Wireless Sensor Networks. In *11th ACM SIGOPS European Workshop*, pages 7–12, Leuven, Belgium, September 2004.

[RHH01] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent Group Membership in Ad Hoc Networks. In *23rd International Conference in Software Engineering (ICSE)*, Toronto, Canada, May 2001.

[RKM02] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware Challenges for Wireless Sensor Networks. *ACM Mobile Computing and Communication Review*, 6(4):59–61, October 2002.

[SSW$^+$05] Cory Sharp, Shawn Schaffert, Alec Woo, Naveen Sastry, Chris Karlof, Shankar Sastry, and David Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 93–107, January 2005.

[TAGH02] S. Tilak, N. Abu-Ghazaleh, and W. Heinzelman. A taxonomy of wireless microsensor network models, 2002.

[tin] TinyDB: A Declarative Database for Sensor Networks.

[UWMG05] A. Ulbrich, T. Weis, G. Mühl, and K. Geihs. Application Development for Actuator and Sensor Networks. In *GI Workshop on Sensor Networks*, ETH Zurich, Switzerland, March 2005.

[Wal99] J. Waldo. The Jini Architecture for Network-Centric Computing. *Communication ACM*, 42(7):76–82, 1999.

[Wea02] J. Weatherall. A Ubiquitous Control Architecture for Low Power Systems. In *ARCS 2002 International Conference on Architecture of Computer Systems*. Springer-Verlag, April 2002.

[Wel96] M.P. Wellman. Market-oriented programming: Some early lessons. 1996.

[Wel03] M. Welsh. Exposing Resource Tradeoffs in Region-Based Communication Abstractions for Sensor Networks. In *2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, San Jos, USA, November 2003.

[WM04] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 29–42, San Jos, USA, March 2004.

[WMG04] Alec Woo, Sam Madden, and Ramesh Govindan. Networking support for query processing in sensor networks. *Communications of the ACM*, 47(6):47–52, 2004.

[WSCB04] K. Whitehouse, C. Sharp, D. Culler, and E. Brewer. Hood: A Neighborhood Abstraction for Sensor Networks. In *MobiSys 2004*, Boston, USA, June 2004.

[xmk] eXtreme Minimal Kernel. Web page: http://www.shift-right.com/xmk.htm. Visited 2005-06-27.

[YG02] Y. Yao and J.E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3):9–18, September 2002.