VANDERBILT UNIVERSITY

INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS

Institute for Software Integrated Systems
Vanderbilt University
Nashville Tennessee 37235

TECHNICAL REPORT

TR #: ISIS-06-706

Title: OASiS: A Service-Oriented Middleware for Pervasive Ambient-Aware Sensor Networks

Authors: Isaac Amundson, Manish Kushwaha, Xenofon Koutsoukos, Sandeep Neema, Janos Sztipanovits

OASiS: A Service-Oriented Middleware for Pervasive Ambient-Aware Sensor Networks

Isaac Amundson, Manish Kushwaha, Xenofon Koutsoukos, *
Sandeep Neema, Janos Sztipanovits

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, Tennessee 37235, USA

Abstract

Heterogeneous sensor networks consisting of networked devices embedded into the physical world have a significant role in pervasive computing systems. Such sensor networks may contain wireless sensor networks that are ensembles of small, smart, and cheap sensing and computing devices that permeate the environment, as well as high-bandwidth rich sensors such as satellite imaging systems, meteorological stations, air quality stations, and security cameras. Emergency response, homeland security, and many other applications have a very real need to interconnect such diverse networks and access information in real-time. While Web service standards provide well-developed mechanisms for resource-intensive computing nodes, linking such mechanisms with wireless sensor networks is very challenging because of limited resources, volatile communication links, and often node mobility.

This paper presents a service-oriented programming model and middleware for ad-hoc wireless sensor networks which permits discovery and access of Web services. Sensor network applications are realized as graphs of modular and autonomous services with well-defined interfaces that allow them to be published, discovered, and invoked over the network, providing a convenient mechanism for integrating services from heterogeneous sensor systems. Our approach provides dynamic discovery, composition, and binding of services based on an efficient localized constraint satisfaction algorithm that can be used for developing ambient-aware applications that adapt to changes in the environment. A tracking application that employs many inexpensive sensor nodes, as well as a Web service, is used to illustrate the approach. Our results demonstrate the feasibility of ambient-aware applications that interconnect wireless sensor networks and Web services.

* Corresponding author: Tel.: +1 615 322 8283; Fax: +1 615 343 5459.
Email address: `xenofon.koutsoukos@vanderbilt.edu` (Xenofon Koutsoukos,).

1 Introduction

Wireless sensor networks (WSNs) consist of small, inexpensive computing devices which interact with the environment and communicate with each other to identify spatial and temporal patterns of physical phenomena. A sensor web is a heterogeneous collection of such networks, and can also include high-bandwidth sensing platforms such as satellite imaging systems, meteorological stations, air quality stations, and security cameras. Such heterogeneous sensor networks have a significant role in pervasive computing systems which greatly benefits applications ranging from emergency response to homeland security.

Sensor network applications often run on networks of hundreds or thousands of nodes distributed over a wide area. Limited computing resources, volatile communication links, and node dropout are common occurrences and must therefore be considered when developing the application. Pervasive computing environments, in which small-scale networked devices are embedded into the physical landscape, demand applications to be scalable and support device heterogeneity. Sensor nodes may have different and varying capabilities, be manufactured and operated by different vendors, and be accessed by multiple clients exercising different functionalities [1].

A *service-oriented architecture* (SOA) offers flexibility in the design of WSN applications since it provides accepted standards for representing and packaging data, describing the functionality of services, and facilitating the search for available services which can be invoked to meet application requirements. SOA deployment has already proved successful on the World Wide Web, however Web service technologies have been developed assuming standard Internet protocols and are not realizable in resource-constrained sensor networks.

In this paper, we present OASiS, an Object-centric, Ambient-aware, Service-oriented Sensornet programming framework for WSN applications. In the *object-centric* paradigm, the application programmer is presented with a layer of abstraction in which an event detected by the sensor network is represented as a logical object which then drives the application. Furthermore, our model is *ambient-aware*, enabling the application to adapt to network failures and environmental changes by employing a dynamic service discovery protocol.

Applications are realized as graphs of services, executed in response to the detection of a physical phenomenon. Services are modular and autonomous, properties which permit them to be dynamically composed into complete applications. Because applications can consist of multiple services on multiple nodes, we have encapsulated service discovery, scheduling, and access mechanisms into a sensor node middleware. These mechanisms are designed to function efficiently on resource-constrained sensor nodes, and in the presence of volatile communication links.

In order to effectively execute a service graph, services should be invoked only if they satisfy a set of constraints specified by the user or required by the application domain. For example, it may not be desirable for two CPU-intensive services to be running simultaneously on the same node. Therefore, this constraint needs to be declared in the service graph and evaluated when the application must locate and run a select set of services. The process of ad-hoc service discovery and constraint satisfaction is called *dynamic service configuration*.

Our programming model can be used to build a wide variety of dataflow applications such as mobile vehicle tracking, fire detection and monitoring, and distributed gesture recognition. As proof of concept, we have developed a simplified indoor tracking experiment, which monitors a heat source as it travels through the sensor network region. The application employs many resource-constrained sensor nodes, but it can also access Web services provided by high-bandwidth nodes. The case study demonstrates the feasibility and utility of a service-oriented WSN programming model. Furthermore, by providing access to Web services, we can incorporate functionality into our WSN applications that would otherwise be unavailable.

This paper is organized as follows. Section 2 provides an overview and highlights the contributions of OASiS. Section 3 describes our programming model. Section 4 describes dynamic service configuration. Our middleware implementation is presented in Section 5. Section 6 presents the details of our case study. In Section 7, we compare our research to similar work that has recently appeared in the literature. Section 8 concludes.

2 The OASiS Programming Framework

At present, users wishing to deploy WSN applications must be adept at developing the sensor network middleware, the domain-specific functionality, and perhaps even an interactive front-end. Application development will benefit from a programming paradigm that provides *separation of concerns* (SoC). OASiS is a programming framework that facilitates SoCs for application development through a multilayer development process. Figure 1 illustrates the relationship between each development stage in OASiS.

In OASiS, core sensor network functionalities are bundled as *middleware services* including service discovery, service graph composition, failure detection, node management, and others. The *domain services* development layer provides domain-specific service libraries written by the domain experts, which

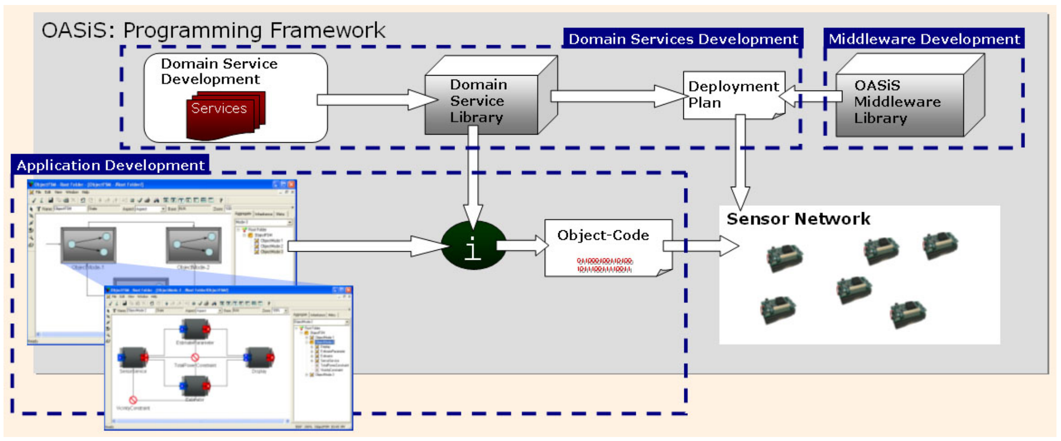

Fig. 1. OASiS: Programming Framework

then can be used by the application developer. OASiS then wires the domain services onto the middleware to produce node-level executable code for deployment on the network. *Application development* in OASiS does not require any expertise in sensor network programming. Instead, complete applications are developed using model-integrated computing techniques [2].

In addition to providing multilayer development, the proposed programming framework addresses many design challenges for networked embedded sensor systems:

- Our ambient-aware middleware supports dynamic service discovery and configuration to address changes in network topology due to failures and unreliable communication links.
- Heterogeneity is addressed by employing well-defined services on *heterogeneous* platforms that can be composed together in a seamless manner. For example, resource-intensive Web services can easily be plugged into OASiS applications and are treated as ordinary WSN domain services.
- OASiS supports *real-world integration* in application design by providing the means to specify spatio-temporal service constraints. The ability to attach such constraints to services before they are invoked is an important aspect of WSN application programming.
- The service-oriented approach enables in-network *data aggregation* using services with inputs from multiple sensor services.
- OASiS supports specification of both application-specific and network *QoS requirements* and can handle QoS violations using application reconfiguration to satisfy QoS requirements.

3 The OASiS Programming Model

This section presents our programming model for object-centric, service-oriented, ambient-aware sensor network applications. The model defines the logical elements necessary to support a wide variety of WSN dataflow application.

3.1 The Object-centric Paradigm

In an *object-centric* application, an *object* is a unique logical entity corresponding to a physical phenomenon under observation. The object is unique because it resides on only one node at a time. This does not imply the object is confined to a single node over its entire lifetime; it has the ability to migrate between nodes as it follows its real-world counterpart. The network application is then driven by the object; that is, its behavior reflects the object's current state.

In OASiS, an environmental phenomenon of interest is represented by a finite state machine (FSM), referred to as the *logical object*. We elected to use the FSM representation because this model of computation is an intuitive method for describing the distinct states a physical object might exhibit. Each FSM mode corresponds to a different physical state, and contains a service graph specifying the appropriate actions to take for the specific situation. For example, in a gesture recognition application, a target may enter the field of view of one or more cameras. Depending on the degree of coverage, different algorithms described by different dataflow graphs and constraints will need to be executed. In this manner, application execution is driven by the state of the object. For this work, the service graph in each FSM mode is known a priori.

Before a logical object is instantiated, a physical phenomenon must be detected. This is achieved by comparing sensor data with an *object context*. The object context defines the physical phenomenon in logical terms. For example, we might declare an object context for *fire* as TEMPERATURE $\geq 100^{\circ}C$. The object context also specifies how frequently the environment should be sampled.

Because multiple nodes may detect the same physical phenomenon at roughly the same time, a mechanism is required to ensure that only one logical object is instantiated. To provide this guarantee, OASiS employs an object-owner election algorithm, similar to that of [3]. The object creation protocol, executed by each candidate node, is outlined in Algorithm 1.

After the object creation protocol completes, exactly one node, referred to as

Algorithm 1 Object Creation Protocol

```
1: if object creation condition == TRUE then
2:    declare yourself a candidate
3:    if owner election not already in progress for recently detected object then
4:       initiate owner election
5:    end if
6:    if you win the owner election then
7:       declare yourself the owner
8:       populate the object state variables
9:       identify the object default mode and initiate dynamic service configuration
10:   end if
11: end if
```

the *object node*, is elected owner of the logical object corresponding to the physical phenomenon. The object initiates in the default mode of the FSM and starts the process of dynamic service configuration (described below), after which the application begins execution. The object maintenance protocol evaluates the mode transition conditions every time the object state is updated. If a mode transition condition evaluates *true*, the protocol makes the transition to the new mode. The mode transition involves resetting any object variables, if applicable, and configuring the new service graph corresponding to the new object mode.

The object also has a migration condition, which if evaluates *true*, invokes the object migration protocol. The selection policy for migration destination is tied to the migration condition that triggers the protocol. In tracking applications, for example, an increase in the variance of location estimate can serve as a migration condition, and the owner selection policy will choose the node closest to the physical phenomenon. Other migration conditions include the object-node running low on power, in which case the selection policy chooses a node with a sufficient power reserve. The migration process consists of running the owner election algorithm to select the migration destination based on the selection policy and transferring the object state to the new object node. The migration protocol is outlined in Algorithm 2.

Algorithm 2 Object Migration Protocol

```
1: if object migration condition == TRUE then
2:    initiate owner election
3:    remove yourself as candidate for owner
4:    transfer the object to winner
5: end if
```

When the sensor network is no longer able to detect the physical phenomenon, the logical object must be destroyed. This is a simple matter of resetting the logical object state to *null*. After an object has been destroyed, the sensor network begins searching for a new object context.

The object-centric paradigm provides abstractions which place the focus on the environmental phenomena being monitored, thus bypassing the complex issues of network topology and distributed computation inherent to sensor network application programming. This effectively transfers ownership of common tasks such as sensing, computation, and communication from the individual nodes to the object itself, providing a greater amount of flexibility and efficiency both at design-time and at run-time.

In addition, object-centric programming tackles scalability issues by focusing on only a small portion of the network close to the physical phenomenon of interest. In OASiS, this facilitates ambient-awareness by considering dynamic service configuration only for a neighborhood of the network and solving a localized constraint satisfaction problem. Ambient-awareness is discussed in detail in Section 4.

3.2 Services in Sensor Networks

In our service-oriented architecture, the object contains one service graph for each FSM mode whose constituent services provide the application with the required functionality. Specifically, a service graph contains a set of services, a set of bindings, and a set of constraints, where a service is represented by a service ID and a port ID, a binding is a connection between two services, and a constraint is a restrictive attribute relating one or more services.

Figure 2 depicts the service graph used in our tracking application example (presented in Section 6). Our localization algorithm requires sensor data from three nodes surrounding the source event, in addition to the current wind velocity in the region. Therefore, the service graph consists of the three Sensing services and one Wind Velocity service whose outputs are wired to the inputs of a Localization service. The Localization service is wired to a Notification service, which informs us of the source's current position.

Services are resources capable of performing tasks that form a coherent functionality from the point of view of provider entities and requester entities [4]. They are the basic unit of functionality in OASiS, and have well-defined interfaces which allow them to be described, published, discovered, and invoked over the network. Each service can have zero or more *input ports* and zero or more *output ports*. For example, the Localization service in our tracking example has four input ports, three for sensor readings and one for wind velocity, and one output port, on which the position estimate is placed.

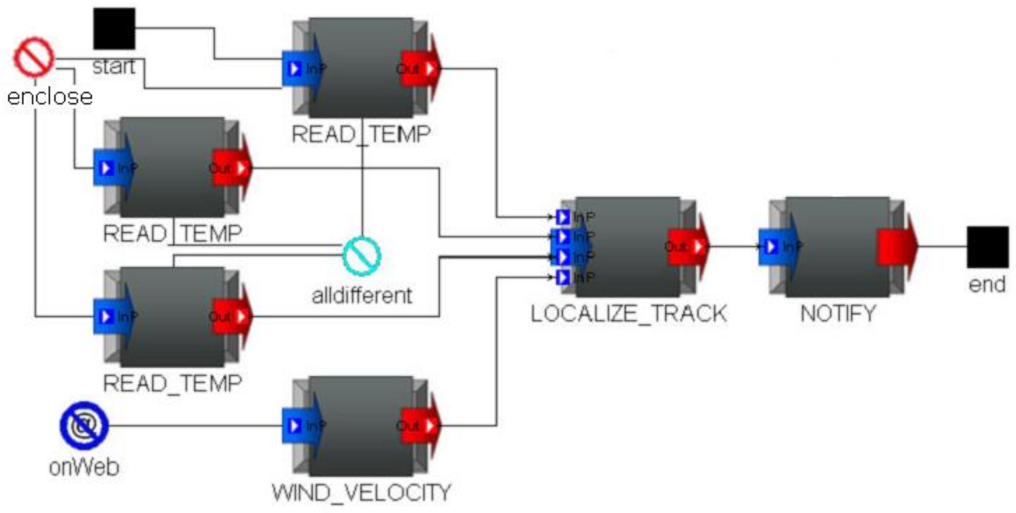

Fig. 2. Service Graph for Tracking Application

One necessary property of OASiS services is that they must be able to communicate asynchronously with each other because the network lacks a global clock for synchronous communication between nodes. Our programming model accounts for this by means of the globally asynchronous, locally synchronous (GALS) model of computation [5]. GALS guarantees that communication between services will occur asynchronously, while intra-service communication such as method calls will exhibit synchronous behavior. As such, GALS is an important and desirable feature of sensornet-based service-oriented applications.

Application services can run on the resource-constrained nodes of the sensor network or they may be executed on more powerful sensor nodes in a high-bandwidth network. In our work, these richer services are implemented as *Web services*. We elect to use Web services due to their current popularity, well-defined and documented standards, and the existing functionality they provide. By taking advantage of these high-bandwidth Web services, applications have access to a wide range of functionality which would otherwise be too resource-intensive for the sensor node platform.

Services are modular and function autonomously, properties which both facilitate application programming and provide an efficient mechanism for application reconfiguration during runtime. Because services provide an interface describing their functionality in terms of inputs and outputs, the programmer does not have to be concerned with their physical placement, hardware platform, or implementation language. Furthermore, services allow new functionality to be easily inserted into the network without having to redeploy the underlying WSN application.

3.3 Service Constraints

It is often undesirable for multiple services in an application to be running concurrently on the same node. Conversely, there arise situations in which two services *must* be running on the same node. Many localization algorithms require sensing services to be situated in a precise spatial configuration. Other sensor node properties such as power level and physical position may also be important when deciding where to run a service. The ability to specify these types of constraints is a necessary aspect of service graph creation.

Typical constraints associated with a dataflow graph can be categorized as either *property* or *resource-allocation* constraints. Property constraints specify a relation between the properties of services (or the nodes providing the services) and some constant value. The ENCLOSE property constraint, for example, specifies that nodes providing services a, b, and c must surround the physical phenomenon of interest. The ENCLOSE constraint is very important for tracking spatial phenomena and is discussed in more detail in Section 4. Resource-allocation constraints define a relationship between the nodes that provide the services. For example, a resource-allocation constraint can specify that services a, b, and c must run on different nodes (or must all run on the same node).

Constraints can further be categorized as being either *atomic* or *compositional* based on their cardinality, or *arity*. Hence, a constraint involving a single service is an atomic (*unary*) constraint, while constraints involving two (*binary*) or more (*n-ary*) services are compositional constraints.

In the following, we formally define the constraints considered in our framework. A method for determining a service configuration which satisfies such constraints in presented in Section 4.

(1) *Atomic property constraint:*

$$\mathbf{service}.p \ \mathbf{op} \ K$$

where, p is a node property, **op** is a relational operator ($\mathbf{op} \in \{>, \geq, <, \leq, ==, \neq\}$), and K is some constant value. For example, the constraint that service a must be provided by a node at least one meter above the ground is written as,

$$a.provider.\mathrm{z} \geq 1$$

(2) *Compositional property constraint:*

$F(p)$ **op** K **over** $\mathcal{S}$

where p and **op** are defined above, and F is a composition function on property p for all services in the set $\mathcal{S}$. For example, to specify that the average power level of nodes providing services a, b, and c must be greater than or equal to 85% is written as,

$\textbf{average}(provider.\textsc{power}) \geq 85$ **over** $\{a, b, c\}$

(3) *Atomic resource-allocation constraint:*

service.*provider.type* **op** TYPE_SET

where, **op** $\in \{==, \neq, \in, \notin\}$. For example, the following constraint ensures that service a does not run on a set of nodes with particular IDs.

$a.provider.\textsc{id} \notin \{NODE_1, NODE_2, NODE_3\}$

(4) *Compositional resource-allocation constraint:*

$F(provider.type)$ **over** $\mathcal{S}$

where, $F \in \{\textbf{allSame}, \textbf{allDifferent}\}$. For example, the constraint that services a and b must run on the same node, and c must run on a different node can be written as

allSame(*provider*.ID) **over** {a,b} && allDifferent(*provider*.ID) **over** {a,c}

The types of constraints that can be specified in a service graph are consistent with those required for modeling typical WSN applications, such as vehicle tracking and gesture recognition. By placing more constraints on services in the service graph, the programmer can specify precise application behavior. However, constraining an application too much may result in an infeasible configuration during runtime.

3.4 Service Discovery and Composition

Before an object can start executing a service graph, a *Service Discovery Protocol* (SDP) is invoked to determine which nodes in the network provide which services. Our model employs passive service discovery, in which a provider advertises a service only when a request for that service has been received [6].

The SDP maintains a *service repository* (SR) which catalogs application services running both locally and remotely. Should an entry become stale due to communication failure or node dropout, for example, or a new service request arrives, the SDP locates a provider for that service by performing the steps outlined in Algorithm 3.

Algorithm 3 Service Discovery Protocol

```
 1: Input: Service ID
 2: search the Service Repository
 3: if Service ID is found in SR then
 4:     send local Service Info to Composer
 5: else
 6:     compose Service Discovery Message
 7:     broadcast Service Discovery Message
 8:     receive Service Discovery Reply message
 9:     record service provider node ID in SR
10: end if
11: send Service Info to Composer
```

The service discovery algorithm receives as input a service ID, which if not present in the service repository, will prompt the SDP to broadcast a service request to other nodes in the network, up to a specified number of hops away. The outgoing *service discovery message* contains the ID of the requested service and the node ID of the sender. Nodes providing the requested service will send a *service discovery reply message*, which includes information containing node vitals such as physical location and remaining power level. The SDP caches the provider node information in the SR, and forwards the message to the *Composer*.

It is the Composer's job to produce a set of services and service providers that satisfy the constraints specified in the service graph. These services are then *bound* and eventually invoked. The Composer's behavior is outlined in Algorithm 4. The ID of each service in the service graph is passed to the SDP (lines 3-5). Because several instances of the same service could be residing on multiple nodes across the network, the Composer can expect multiple replies. As replies arrive, the Composer checks to see that any atomic service graph constraints are satisfied, and if so, the node information is stored (lines 6-9). Compositional constraint satisfaction commences after all replies have been received. Finally, the connections between the services in the service graph are examined, and a *service binding message* is created for each (line 12). The binding message contains the service and node IDs of the connection source, as well as the service and node IDs of the connection destination. In addition, the binding message contains the IDs of the intermediate nodes along the multi-hop path between source and destination. The message is sent to the connection source node (line 13) so that it may properly direct the output of its service to the input of the service specified by the connection destination.

Algorithm 4 Composer

```
 1: **Input:** Service Graph $\mathcal{G}$
 2: parse $\mathcal{G}$ into sets of Services, Connections, and Constraints
 3: for all $S \in Services$ do
 4:    send S to Service Discovery Protocol
 5: end for
 6: receive Service Discovery Reply from SDP
 7: if node satisfies Atomic Constraints then
 8:    cache node info
 9: end if
10: do Compositional Constraint Satisfaction
11: for all $C \in Connections$ do
12:    create a Service Binding message
13:    send Service Binding message to service provider node
14: end for
```

Once the object has finished initialization, the service graph can be executed. This involves the invocation of the source services in the service graph. Depending on the nature of the object, the service graph may be executed periodically, in which case the source services are invoked at a predetermined rate. Because each application service invokes the next, the service graph will execute to completion without the need for any type of centralized control.

Dynamic network behavior in WSNs can cause problems during application execution such as service unavailability and violation of constraints. Querying a centralized service repository each time a new service instance is needed can be expensive, especially when the repository is located multiple transmission hops away. The passive service discovery approach was found to be the most energy efficient for mobile ad hoc networks with limited power resources [6]. Requests are flooded a limited number of hops throughout the network, and all providers of the requested service respond with a message that follows a direct path back to the object node. The Composer is then provided with a list containing only those services requested.

4 Ambient-aware Programming

Before an object is instantiated, each node in the sensor network periodically takes samples of the environment, which are then compared against an object context. A positive comparison indicates the network has detected a target, and an object is then created. During execution of the application, access to a new instance of a service may become necessary if the node providing the current service drops off the network. This necessitates the ability to locate a service provider both efficiently and quickly. An application capable of adapting to the environment in such a manner is said to be *ambient-aware.*

Ambient-awareness is key to pervasive computing environments. A dynamic network topology requires each device in the system to have up-to-date knowledge of its neighborhood. Without this information, message transmissions may never reach their intended destination, network services may not be discovered, and physical phenomena may not be detected. As pervasive computing systems become more commonplace, ambient-aware mechanisms will play a vital role in maintaining the graceful execution of the application.

Our SOA is made ambient-aware by means of *dynamic service configuration.* Before a service graph is executed, knowing the locations of the services it contains is irrelevant as this information can become outdated before it is ever required. Dynamic service configuration composes and binds the service graph on demand, which results in fewer message transmissions and the most up-to-date service configuration.

At all times after initialization, each node has a notion of the location of the services it requires. If a communication failure occurs during the process of invoking one of these services, the application is able to recover by locating a new acceptable instance of the service.

4.1 Constraint Satisfaction

Service graph instantiation can be modeled as a constraint satisfaction problem, where services in the *abstract* service graph are the constraint variables, and the nodes that provide a particular service constitute the domain. The constraint satisfaction problem (CSP) is formally defined in [7].

A finite CSP $\mathcal{P} = (X, \mathcal{D}, \mathcal{C})$ is defined as a set of n *variables* $X = \{x_1, ..., x_n\}$, a set of finite *domains* $\mathcal{D} = \{D_1, ..., D_n\}$ where D_i is the set of possible *values* for variable i, and a set of *constraints* between variables $\mathcal{C} = \{C_1, ..., C_m\}$. A constraint C_i is defined on a set of variables $(x_{i_1}, ..., x_{i_j})$ by a subset of the Cartesian product $D_{i_1} \times ... \times D_{i_j}$. A solution is an assignment of values to all variables which satisfy all the constraints.

The design space for a constraint satisfaction problem is the set of all possible tuples of constraint variables. Formally,

$$\mathcal{D} = \{(v_1, v_2, ..., v_n) | v_1 \in D_1, v_2 \in D_2, ..., v_n \in D_n\}$$

Constraint satisfaction prunes the design space as much as possible for all different types of constraints, followed by backtracking until a feasible solution is found. The specific pruning method depends on the constraint under consideration, specifically the constraint property, constraint operator, and composition function.

1) Atomic Constraint Satisfaction: Atomic constraints are straightforward to satisfy. Because each atomic constraint is defined on a single variable, pruning the domain of that variable will leave the domain *consistent*, and hence satisfy the constraint. In Algorithm 5, the resulting pruned domain $\tilde{D}_i$ for constraint variable x_i is consistent.

Algorithm 5 Atomic Constraint Satisfaction

```
1: $\tilde{D}_i = D_i$
2: for all $v_i \in \tilde{D}_i$ do
3:   if !satisfy$(C_i, v_i)$ then
4:     $\tilde{D}_i = \tilde{D}_i - v_i$
5:   end if
6: end for
```

2) Compositional Constraint Satisfaction: Algorithm 6 outlines the process of compositional constraint satisfaction.

Algorithm 6 Compositional Constraint Satisfaction

```
1: for all $C_i \in \mathcal{C}$ do
2:   $\tilde{\mathcal{D}} = \text{prune\_design\_space}(C_i, \mathcal{D})$
3: end for
4: $okay = \text{FALSE}$
5: while !$okay$ do
6:   $sol = \{(v_{index_1}, v_{index_1}, ..., v_{index_1}) | \forall i\ v_{index_i} \in \tilde{D}_i\}$
7:   $okay = \text{TRUE}$
8:   for all $C_j \in \mathcal{C}$ do
9:     if !satisfy$(C_j, sol)$ then
10:      $okay = \text{FALSE}$
11:      $backtrack()$
12:    end if
13:  end for
14: end while
```

a) Compositional Property Constraints: The compositional property constraints are described in Section 3, where F is the composition function. Our programming model includes definitions for several common aggregate functions such as SUM, AVERAGE, and MEDIAN.

Many tracking applications employ localization algorithms which require measurement data to come from multiple sensors surrounding the physical phenomenon of interest. The quality of the localization estimate often depends on how well the spatial configuration of these sensors is described. We have therefore defined an additional composition function called ENCLOSE which is useful for specifying the spatial configuration of sensor nodes. For example, in our tracking application we use ENCLOSE to specify that we would like to have at least three different sensor nodes enclosing the tracked phenomenon at all times. The constraint ENCLOSE(L) *over* $\mathcal{S} = \{s_1, s_2, s_3\}$, specifies that the location L must be enclosed by the sensor nodes which provide services s_1, s_2, and s_3. The enclosure location, L can be specified as a fixed location or as a node ID. For example, ENCLOSE$(s_4.\text{location})$ *over* $\mathcal{S} = \{s_1, s_2, s_3\}$ specifies that the location of the node providing service s_4 must be enclosed by sensor nodes that provide services s_1, s_2, and s_3.

In general, higher-level, complex constraints are more difficult and demanding to satisfy. However, such constraints can be transformed into lower-level, simple constraints that provide the desired result, while minimizing the power and resources expended in satisfying it [8]. We model the ENCLOSE constraint based on the AM_I_SURROUNDED query described in [8]. The two-dimensional definition of ENCLOSE is as follows: L is surrounded by $\{s_1, s_2, s_3\}$ if there is no line in the plane that can separate L from all of $\{s_1, s_2, s_3\}$. For this definition, the constraint can be reduced to ENCLOSE(L) *over* $\{s_1, s_2, s_3\} \Rightarrow$ CCW(L, s_1, s_2) & CCW(L, s_2, s_3) & CCW(L, s_3, s_1), where CCW(a, b, c) specifies that locations a, b, and c form a counter-clockwise-oriented triangle in 2-D. The geometric constraint CCW(L, s_3, s_1) is easy to satisfy by simple computation.

The definition of ENCLOSE varies for different sensor domains. For example, one domain can define an *enclosed* region to be the overlap of member sensing ranges. Consider another example of camera sensors with orientation and limited field-of-view. The enclosed region in this case is the intersection of fields of view recorded by all member cameras. Figure 3 illustrates various enclosed region definitions.

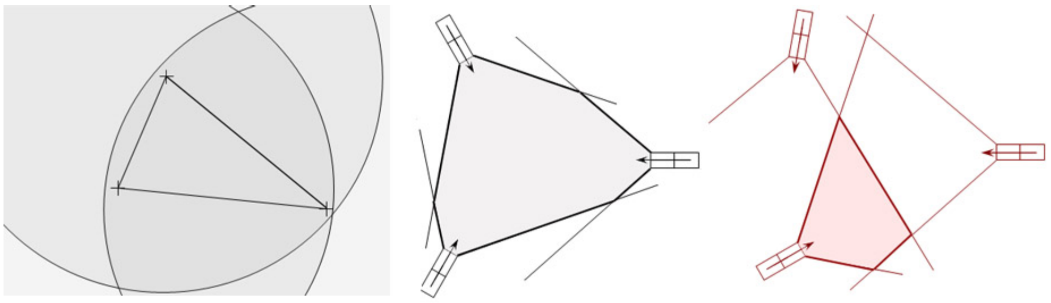

Fig. 3. Enclose Constraint

b) Compositional Resource-Allocation Constraints: There are two types of composition functions for compositional resource-allocation constraints, *all-*

Same and *allDifferent*. Satisfying the *allSame* constraint is relatively straightforward; the design space is the intersection of domains of all the participating constraint variables. To satisfy the *allDifferent* compositional constraint, a solution is picked from the domain for each constraint variable. If the current set of solutions satisfies the constraint, a valid solution has been found. Otherwise, a backtracking algorithm is required to replace the solution for one constraint variable and re-evaluate the constraint. At the end of the backtracking step, either a solution has been found or the entire design space has been searched without finding any valid solution.

Other than property and resource-allocation constraints, quality of service (QoS) constraints can also be specified for an application. For example, in our tracking application, we have a QoS constraint that requires the variance of the location estimate from the Localization service to be below a certain threshold. These constraints are evaluated by the object upon completion of each periodic service graph execution.

Although solving CSPs can be computationally expensive [9], by limiting the scope of the service discovery protocol in a neighborhood of the object node and by keeping the constraint specification syntax simple, the problem can be solved on resource-constrained sensor nodes. For example, our results in Section 7 show that the problem can be solved online for a 3-hop neighborhood. The constraint specification syntax still permits the user to accurately specify desired application behavior. OASiS implicitly assumes constraint satisfaction will terminate with a valid configuration. This assumption is reasonable for WSNs, since redundancy is one of their main characteristics. Note that OASiS does not attempt to find an optimal configuration, because this can be too computationally expensive. Instead, the first feasible configuration that satisfies all the constraints is selected.

5 The OASiS Middleware

We have developed a suite of middleware services which support the features of our programming model. The middleware provides a layer of network abstraction, shielding the application programmer from the low-level complexities of sensor node operation such as resource management and communication. It gracefully handles the decomposition of desired application behavior to produce node-level executable code for an object-centric, service-oriented WSN application.

The middleware services, which include a *Node Manager*, *Object Manager*, and *Dynamic Service Configurator*, provide support to the application services. Figure 4 illustrates the relationship between our middleware and the sensor network, while Figure 5 illustrates the relationship between the different types of middleware and application services at the sensor node level.

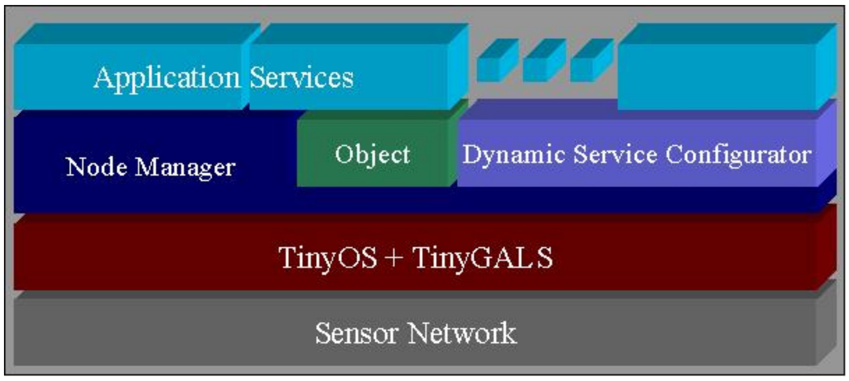

Fig. 4. Middleware

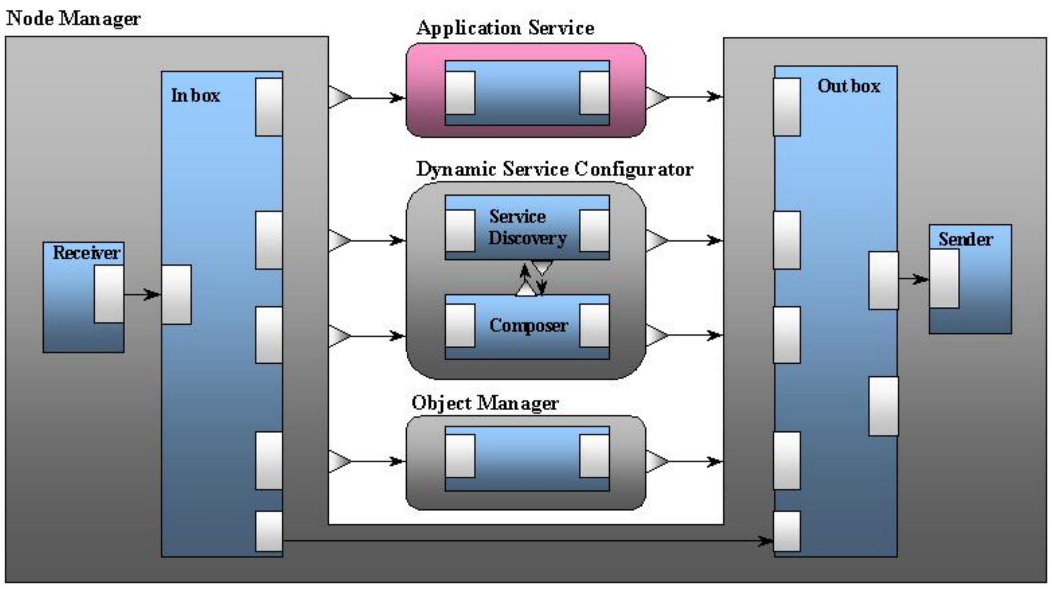

Fig. 5. Middleware node architecture

The Node Manager is responsible for message routing between services, both local and remote. The first nine bytes of any message handled by the Node Manager consist of a control structure which contains source and destination node IDs (2 bytes each), source and destination service IDs (1 byte each), message type (1 byte), and hop number (1 byte). The Node Manager examines the control structure and determines the appropriate destination for the message. For efficiency, it has *short circuit* functionality that allows it to catch outgoing messages bound for local services and reroute them directly.

Three key types of messages are handled by the Node Manager. *Service discovery messages* come from neighboring nodes inquiring if a specific service is available. The Node Manager passes these messages to the local Service Discovery Protocol. An incoming *service binding message* indicates that a local service has been registered for use by an object, and includes information on where to send its output data when complete. A *service access message* is a request to run a local service, and may also contain input data. The Node Manager invokes the specified service and passes in the data.

The middleware supports multi-hop routing, and employs a protocol similar to DSR [10], optimized for the OASiS architecture. There are three types of messages that require routing information: (i) service discovery reply messages, (ii) service binding messages, and (iii) service access messages. Note that service discovery request messages are flooded throughout the network, and therefore do not require any routing information. The Node Manager provides routing functionality by maintaining a *next-hop table* which stores the node ID of a known service provider, along with the ID of the next node along the multihop path to that provider. As a service discovery message travels from the object node to service provider nodes, the Node Manager at each intermediate forward point records the node ID of the previous forward point. This gives the service provider a direct path back to the object node for service discovery reply messages.

A service discovery request message will flood the network up to a maximum number of hops, MAX_HOPS, which is a value specified a priori by the domain-service or application developer. At each forward point, the *hopNum* counter in the message header is incremented, and the message will not be forwarded once the counter reaches MAX_HOPS. Note that MAX_HOPS is the maximum number of hops from the object node to a service provider. This implies that service-to-service communication could possibly travel twice as many hops, if each service provider were MAX_HOPS from the object node on opposite sides. Service-to-service communication presents a challenge for multi-hop routing because, although the object node knows the shortest path from itself to each service provider, it has no way of knowing the shortest path between the providers themselves. Rather than expending energy by sending out numerous path-probing message transmissions, the shortest path between two service providers is estimated by using the knowledge of the physical location of the service provider and the maximum physical distance a message can be transmitted. This method does not guarantee that the shortest path selected will be a feasible one, in which case the next shortest path should be selected.

The Dynamic Service Configurator contains the SDP and Composer, and functions as described in Section 3. Dynamic service configuration is a relatively energy-intensive operation, due to the number of message transmissions involved in service discovery and composition. A node performing these operations will transmit $2S$ messages, where S is the number of services in the service graph. Nodes responding to service discovery requests transmit at most S replies, one for each service they provide. However, these transmissions only occur during configuration, and not during service graph execution, thus power consumption is kept to a minimum.

The Object Manager is responsible for 1) parsing the object-code byte string, 2) detecting the object context and evaluating the object creation condition at each sample period, 3) invoking the object creation protocol and owner election algorithm, and 4) maintaining the object state variables and evaluating the migration and FSM mode transition conditions. Essentially, the object manager implements the object-centric programming operations described in Section 3.

5.2 WWW Gateway

In order to take advantage of high-bandwidth Web services, the sensor network must have access to at least one World Wide Web *Gateway*. The Gateway resides on a base station and provides access to Web services by translating node-based byte sequence messages to the comparatively bulky XML-based messages used in most Web service standards.

As such, it is also the job of the Gateway to speak the language of Web services. When a service discovery message arrives, the Gateway must locate this service on the Internet. This is accomplished by using the *Universal Description, Discovery and Integration* (UDDI) protocol [11], a Web service standard used for locating and accessing services. Given the proper keys, a UDDI inquiry returns the access point for a specific service as an URL string. Service access is achieved by means of XML-based *SOAP* [12] messages. If the service returns a value, it is also enclosed in a SOAP message. The Gateway to composes and parses these various XML messages and marshals the data appropriately when translating between the sensor network and the World Wide Web.

The role of the Gateway is transparent to the rest of the network. It appears simply as another node, running identical middleware services and providing a set of application services. That the available application services happen to be remotely located is of no interest to the object node making the request. Similarly, other application services inputting data from, or outputting data to a Web service believes the Web service is being provided by the Gateway node.

Note that communication between the sensornet and Internet is bi-directional. Not only can OASiS WSN applications access Web services, but OASiS services can be accessed from the World Wide Web. This permits users, who have no understanding of wireless sensor networks, to access sensor data or run sensor network applications, from a website with access to the OASiS Gateway.

To return to our tracking example, suppose we could improve our localization estimate if we knew the present wind velocity. However, our sensor nodes are not equipped to take wind measurements, so instead we rely on an Internet-based *WindVelocity* service. The service interface definition is provided in a *Web Service Definition Language* (WSDL) [13] file available on the host. This provides us with the information necessary to access the Web service, including input and output parameters and their data types.

While the tracking application is running on the sensor network, the Gateway receives a service discovery message for the WindVelocity service. It receives this message because one of the nodes in the sensor network is attempting to bind a service graph requiring this service. If the Gateway does not already have the WindVelocity service in its cache of recently accessed services, it makes a UDDI inquiry to a registry at a known location which returns the WindVelocity accesspoint URL, if available. The Gateway stores this information, then responds to the Service Discovery Protocol of the requesting node that the WindVelocity service is available.

The Gateway may then receive a service binding message, indicating that the WindVelocity service may be accessed in the near future. The message contains the IDs of the node and service to send the wind velocity data. This information is cached for rapid future access.

When the Gateway receives a service access message from the sensor network, it packages the input data into a SOAP message and invokes the WindVelocity service. The reply is parsed using an XML parser and forwarded to the next service specified in the *service binding repository.*

5.3 Implementation

Our middleware was implemented on the Mica2 mote hardware platform [14] running TinyOS [15]. Our main objective in developing the middleware was to minimize resource requirements while maintaining a robust component-based architecture. The code was developed using galsC [16], a GALS-enabled extension of nesC [17], the de facto programming language for the motes. The Gateway application was developed in Java. Our Web service implementation was realized using a suite of Apache services [18], including the Tomcat 5.5 web server, Axis 1.4 SOAP implementation, and jUDDI 0.9rc4, a Java-based UDDI implementation. MySQL 5.0 was used for the UDDI repository.

Table 1 lists each middleware service, with its code size and memory requirements. These memory sizes are suitable for executing applications on the motes, which have approximately 64 KB of programming memory and 4 KB of RAM. It should be noted that these components can be optimized to further reduce memory size, however there is a tradeoff between an application's compactness and its robustness.

Service	Program memory (bytes)	Required RAM (bytes)
Node Manager	8500	367
Dynamic Service Configurator	11894	822
Object Manager	3560	151
TinyGALS queues & ports	702	1013
Total	24656	2353

Table 1
Implementation Memory Requirements

6 Case Study

Our case study is motivated by an application for tracking chemical clouds. By distributing chemical sensors on the ground, we can detect the passage of a chemical cloud, and alert emergency management teams for containment, rescue, and evacuation. Because the speed and direction of the cloud is heavily dependent on current wind conditions in the region, leveraging this information could greatly improve our trajectory estimate. However, obtaining wind velocity information for an entire region is not a trivial task. One solution would be to deploy anemometers over the region and periodically aggregate wind velocity data, at great expense in terms of time and money. A more practical solution would be to have the tracking application access a public wind velocity Web service on the Internet. We demonstrate the features of the OASiS programming model and middleware by developing such an application. In place of a chemical cloud, our experimental setup consists of a simplified indoor WSN application for tracking a heat source. The case study demonstrates the feasibility and benefits of using OASiS.

6.1 Experimental Setup

Our experimental setup consists of a simplified indoor sensor network application for tracking a heat source, as shown in Figure 6. There are 5 sensor nodes in the field each having a unique ID. Each node also contains a number of pre-loaded services. Table 2 summarizes the sensor node attributes. The heat source follows the trajectory along the path shown in the figure. The path is a straight line from $[180, 180]$ to $[670, 670]$ with Gaussian process noise ($N[0, 10]$).

Node ID	Position	Preloaded Services
1	[400 800]	READ_TEMP, NOTIFY
2	[700 400]	READ_TEMP, NOTIFY
3	[0 500]	READ_TEMP, NOTIFY, LOCALIZE_TRACK
4	[200 0]	READ_TEMP, NOTIFY
5	[800 1000]	READ_TEMP, NOTIFY
6	N/A	WIND_VELOCITY

Table 2
Experimental Setup

The application takes periodic temperature readings from thermistor-equipped sensor nodes. Simultaneously, a Web service is accessed and the current wind velocity obtained. For purposes of this experiment, the wind velocity service returns predetermined values based on the location of the object. At each iteration, these data are processed by a Localization service which estimates the position of the heat source. This estimate is then sent to a Notification service, which reports the location estimate to the user. The object FSM consists of a single mode with a service graph containing six services, shown in Figure 2. We have three instances of the Temperature-sensor service, and specify that each must reside on a different node and in a specific spatial configuration. Our Wind-velocity service is a Web service, which we specify via an atomic *onWeb* resource allocation constraint. The object context is set to "TEMPERATURE ≥ 30".

The Localization service in this application was implemented using an extended kalman filter (EKF) [19]. The system state is a vector of heat source coordinates, $\mathbf{x} = [x\ y]^T$. The measurement vector is the collection of measurements from three temperature-sensor services. The system model that we use is represented by the equation,

$$\begin{bmatrix} x_k \\ y_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \end{bmatrix} + \begin{bmatrix} u_x \\ u_y \end{bmatrix} + \begin{bmatrix} w_x \\ w_y \end{bmatrix} \tag{1}$$

where $[x_{k-1}\ y_{k-1}]^T$ is the previous system state, $[u_x\ u_y]^T$ is the wind velocity, and $[w_x\ w_y]^T$ is the process noise with zero mean and covariance Q. The observation model is given by,

$$z_k^i = \frac{T}{||\mathbf{x}_k - \xi_i||} + v^i \tag{2}$$

where, z_k^i is the k^{th} measurement at the i^{th} sensor node, $\xi_\mathbf{i}$ and v^i are the location and measurement noise at i^{th} sensor, and T is a constant. The sensor node measurement noise is normally distributed with covariance R.

The EKF is initialized with process and measurement noise covariances Q and R, observation model constant T, and initial system state estimate $\mathbf{x}_0$ and its covariance $\mathbf{P}_0$. At each time-step, the service accepts temperature measurements, sensor locations, and wind velocity data as input and produces the estimated source location as output.

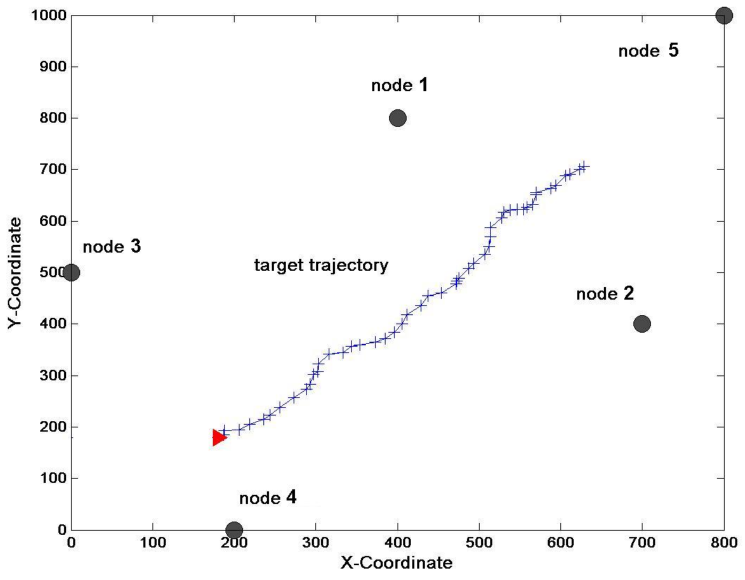

Fig. 6. Experimental setup

6.2 Performance Evaluation

The feasibility and effectiveness of OASiS was evaluated by performing a set of experiments around object maintenance and Web service integration in a multihop network. We present the number of messages transmissions and response time for each stage of execution, which are indicators of bandwidth utilization, energy consumption, and responsiveness of the sensor nodes. We also present the localization estimation error when invoking the WindVelocity Web service versus localization without the Web service. Finally, we compare the overall performance of OASiS against a similar macroprogramming framework, EnviroTrack [20].

Experiment 1: Object creation and application execution

The heat source originates at $[180, 180]$. Nodes 3 and 4 register temperatures higher than the detection threshold and start the object creation protocol. Since node 4 registered the highest temperature, it is elected as the object-node and instantiates an object. The object then parses the pre-loaded object-code, retrieves the service graph for the current object mode, and initiates dynamic service configuration, including service discovery, constraint satisfaction and service binding. The application is configured by invoking instances of the READ_TEMP service on nodes 2, 3 and 4, the LOCALIZE_TRACK service on node 3, and the NOTIFY service on node 4. Once the service graph is configured, application execution commences. The number of message transmissions for object creation and application configuration is summarized in table 3. The delay for object creation and application configuration is 2000 and 3000 time units respectively (1024 time-units = 1 second). The time delay for each action depends on pre-defined timeout values; an *owner-election timeout* for object creation and a *service-configuration timeout* for service graph configuration.

	number of messages
object creation	5 (owner-election messages)
service graph configuration	15 (3 service request messages) (9 service information messages) (3 service binding messages)

Table 3
Experiment 1 results

Experiment 2: Object migration

Once the physical object goes beyond the *enclosure* of nodes 2, 3, and 4, the variance in the location estimate starts to grow. This increase in location estimate variance causes a QoS violation and triggers object migration. As part of the migration protocol, node 4 starts a new owner election procedure by broadcasting a migration message. Nodes reply to the migration message with their most recently sampled temperature values. The current owner elects the node with the highest temperature value as the migration destination, sends the object to it, and *unbinds* all previously bound services. For this experiment, node 4 sends the object to node 2. Table 4 presents the number of messages communicated for object migration and service graph unbinding. The delay for object migration is approximately 2000 time units.

	number of messages
object migration	8 (5 migration messages) (1 object-migration) (1 object-migration ack) (1 object-migration notification)
service graph unbinding	3 (*un*-binding messages)

Table 4
Experiment 2 results

These experiments indicate that OASiS incurs an overhead on the number of messages required and the time delay for object creation, maintenance, migration and service graph maintenance. The tables above demonstrate that the number of messages communicated is reasonably small. The time delays, which were dependent on various timeout constants and were found to be insignificant in this experiment, exhibit the responsiveness of an OASiS application.

Experiment 3: Tracking

Tracking performance was evaluated by comparing the actual heat source trajectory with the estimated trajectory. The tracking accuracies for cases with and without wind velocity data ($u_x = u_y = 0$) was also measured. Figure 7 (a) and (b) shows the tracking results for tests with and without wind velocity data. Figure 7 (c) and (d) shows the tracking results with varied system and measurement noise parameters.

Message transmissions were kept to a minimum due to the passive service discovery protocol as well as the service-oriented architecture itself. Because service messages for this application are small, only one transmission per message was required. Service discovery and binding required a total of 14 transmissions, while a complete execution of the service graph required only six transmissions.

Response times for various operations were also obtained, and are displayed in Table 5. The service discovery response time is provided with and without the Web service. Additionally, Web service access is not included in the service graph execution time, but is provided separately. This is to illustrate the overhead imposed on the system by adding Web service capability. It should be

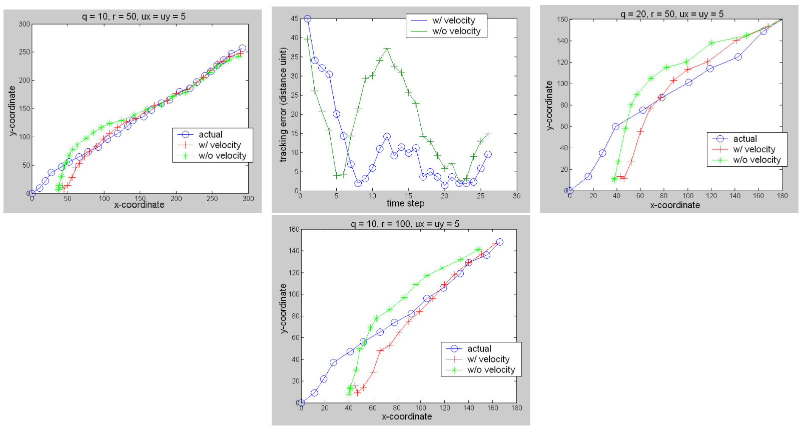

Fig. 7. Tracking results

noted that our Web service implementation is not optimized for speed, however the current service discovery and constraint satisfaction latency is quite acceptable for performing dynamic service configuration. Similarly, the current access latency is acceptable for tracking slower-moving, wide-area phenomena such as chemical clouds and fire. Applications that require service graph execution at higher frequencies should not include Web service access in each iteration.

Operation	Response Time (ms)	Standard Deviation (ms)
Service discovery	4092	113
Service discovery w/o Web service	1400	0.01
Constraint satisfaction	15	0
Service graph execution w/o Web service	81	13
Web service access	502	65
Localization service access	11	0

Table 5
Operation Response Times

6.3 Comparison with EnviroTrack

EnviroTrack [20] is an object-centric macroprogramming framework which enables the run-time system to dynamically instantiate objects corresponding to external events. Like OASiS, objects are unique logical entities that undergo owner election and migrate between nodes following the geographical movement of their equivalent real-world phenomena. Because EnviroTrack provides similar abstractions to OASiS, it serves as a good side-by-side comparison. EnviroTrack has been extended into the EnviroSuite [3] framework (Section 8). However, at the time of publication, only the EnviroTrack source code was available for testing.

We developed an EnviroTrack application based on our case study. A typical EnviroTrack application executes as follows:

- Individual nodes sense the environment for the phenomenon of interest
- When the phenomenon is detected, leader election starts
- When a leader is elected, it aggregates sensor data from neighboring nodes
- Sensor data is analyzed, then output to base station
- When leader is no longer close to object, a new leader is elected and the object migrates

In OASiS, data is aggregated across the network according to the configuration of the service graph. EnviroTrack includes an aggregation library, which contain typical aggregation functions such as AVERAGE, MAX, and MIN. Although it is possible for the programmer to add custom aggregation functions to the library, it is a complicated task. We therefore chose not to implement an aggregation function for the Kalman filter but we used EnviroTrack's default functionality. This choice does not allow us to compare the tracking results but we can compare the message transmission overhead as well as the memory requirements.

Both EnviroTrack and OASiS transmit heartbeat messages during leader election, in which messages are broadcast at a given *heartbeat rate* over a given *candidacy period*. However, unlike OASiS, EnviroTrack leaders continue transmitting periodic *heartbeat* messages for *group maintenance*, which inform neighboring nodes that the leader is alive, and that a phenomenon of interest is in the region. Although EnviroTrack does employ mechanisms for energy-efficiency, we feel that heartbeat messages result in unnecessary power consumption, and therefore use them in OASiS only during object-owner election. Table 6 compares messages transmission overhead for each operation in both frameworks, where C is the number of candidate leader nodes, H is the heartbeat rate, N is the number of nodes currently participating in tracking the target, and S is the number of services in the service graph.

Operation	OASiS	EnviroTrack
Leader Election	O(C * H)	O(C * H)
Dynamic Service Configuration	O(N + S)	0
Group Maintenance	0	O(H)
Application Execution	O(S)	O(N)

Table 6
Message Transmission Overhead in OASiS and EnviroTrack

Overall, OASiS transmits fewer messages than EnviroTrack. Although OA-

SiS must perform dynamic service configuration, this occurs at infrequent intervals, such as when service graph constraints are no longer satisfied and immediately after migration.

Table 7 compares the memory required by both applications. The EnviroTrack application requires less memory, however, the OASiS application is still below the limit, and we believe the additional capabilities of OASiS justify the memory requirements.

Framework	Program Memory (bytes)	Required RAM (bytes)
OASiS	40248	2820
EnviroTrack	23646	1184
Available Mica2 Memory	64000	4000

Table 7
Memory requirements of OASiS and EnviroTrack

7 Scalability

Scalability is a major challenge in the design of sensor networks. Our approach tackles scalability by focusing on small network neighborhood around the object node. Nevertheless, there is a need to perform a scalability analysis of the service discovery protocol in order to measure its effectiveness.

Our analysis is based on Prowler [21], a probabilistic wireless network simulator capable of simulating wireless distributed systems, from the application to the physical communication layer. Although Prowler provides a generic simulation environment, its current target platform is the Berkeley MICA mote running TinyOS.

We consider two different types of network topology: (i) a grid topology where nodes are placed in a 2-dimensional grid and (ii) a random uniform topology where node locations are randomly distributed. We assume that it is required to gather service information from all the nodes in the n-hop neighborhood of the object node. For each type of network topology, we compute the overhead of the OASiS service discovery protocol as a function of the number of hops of the network neighborhood of the object node. The overhead is measured by (i) the number of message transmissions, (ii) the number of nodes discovered, and (iii) the time required for completing the service discovery.

Each sensor node host a number of services and replies with the list of services when queried. For the analysis in Prowler we measure the number of nodes discovered, which presumably provides us with the complete list of services running on those discovered nodes. In other words, node discovery is an implicit measure of service discovery.

We perform the analysis in Prowler by keeping a number of local variables at each node. Specifically, we keep a list of discovered nodes, the number of messages sent, and a hop number on each node. At the end of service discovery we compute the total number messages sent by all the nodes, the total number of *nodes* discovered at the object node, and the total time required by the service discovery.

Figure 8 shows the number of message transmissions for n-hop service discovery, figure 9 shows the number of sensor nodes discovered, and figure 10 shows the time taken for n-hop service discovery for each of the network topologies.

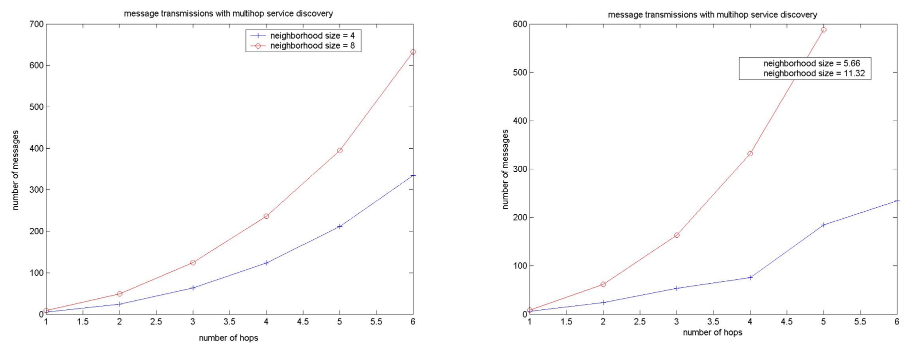

Fig. 8. Number of message transmissions for (a) grid and (b) random topology

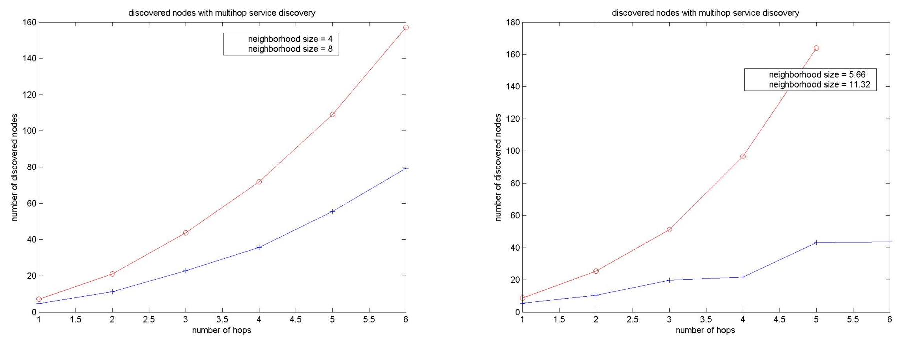

Fig. 9. Number of discovered nodes for (a) grid and (b) random topology

In addition, we define a parameter called *discovery ratio* as the ratio of service discovery messages to the number of nodes discovered at the object node. In other words, discovery ratio is the number of service discovery messages per discovered node. Figure 11 shows the discovery ratio for different network topologies.

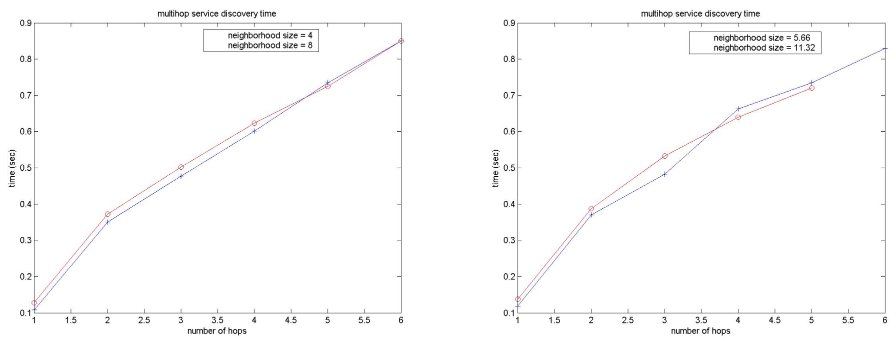

Fig. 10. Time taken for n-hop service discovery for (a) grid and (b) random topology

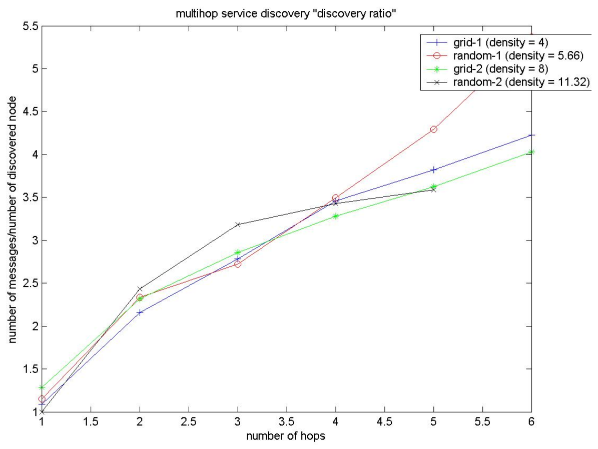

Fig. 11. Discovery ratio with number of hops for all four network topologies

From the results above, we can make some general useful observations. The number of discovery messages seems to increase quadratically with the number of hops, and increases approximately linearly with the node density. The total time taken for service discovery increases linearly with number of hops, while it remains approximately constant with sensor node density. The most interesting obsrvations are concerned with the discovery ratio which increases linearly with number of hops, i.e. the protocol requires approximately n discovery messages per discovered node for n-hop service discovery. Interestingly, discovery ratio remains approximately constant with node density.

Our results indicate that the service discovery protocol in OASiS performs linearly for the number of discovery messages per discovered node with respect of number of hops. Hence, the optimal number of hops for service discovery can be specified in an OASiS application based on the number of services required for the application and the distribution of services in the network.

8 Related Work

Design principles for traditional distributed computing middleware are not directly applicable to WSNs because sensor nodes are small-scale devices with a limited power supply, directly affecting computation, sensing, and (especially) communication, and often operate unattended for prolonged periods of time. The design of a successful WSN middleware must address the various challenges imposed by these characteristics. Many middleware challenges have already been identified [22]. Recently, the WSN community has seen the emergence of a diverse body of macroprogramming languages, frameworks, and middleware which provide solutions to some of these challenges (see [22] and the references therein). In the following, we focus on the frameworks that are more closely related to OASiS.

SONGS [1] is a service-oriented programming model, similar to ours in many respects. However, unlike our object-centric approach to driving application behavior, SONGS dynamically composes a service graph in response to user-generated queries. While this technique works well as an information retrieval system, SONGS lacks the ability to alter its behavior based on a change in environmental conditions.

The object-centric paradigm has been successfully used in the EnviroSuite programming framework [3]. EnviroSuite and OASiS provide a similar level of network abstraction to the application developer, however by employing a service-oriented architecture, OASiS is able to incorporate aspects of modular functionality, resource utilization, and ambient-awareness more efficiently.

The Abstract Task Graph (ATaG) [23] is a macroprogramming model which allows the user to specify global application behavior as a series of abstract tasks connected by data channels for passing information between them. Currently, the ATaG is only a means for describing application behavior. A model interpreter must be employed to decompose this behavior to node-level executable code. In addition, the ATaG provides no means for delegating tasks to sensor nodes which satisfy specific property or resource constraints.

The Agilla framework [24] adopts a mobile agent-based paradigm. However, unlike most other frameworks, Agilla does not require the sensor network application to be deployed statically. Instead, autonomous agents, each with a specific function, are injected into the network at run-time, a technique referred to as *in-network programming*. This approach allows the underlying network application to only be uploaded once onto the node hardware, after which applications can be swapped out or reconfigured at any time. The primary disadvantage of using an Agilla network, compared with our middleware, is that all nodes must be executing the Agilla run-time application. This rules out access to a variety of devices operating on different architectures.

Ambient-aware computing [25] is an emergent technology in which applications are given the ability to interact with their environment such that all devices and services within a fixed geographical range are known at all times. However, for sensor networks consisting of resource-constrained nodes, communication with neighboring devices is often costly. Hence a tradeoff exists between the rate at which a node can update its understanding of the surrounding environment and the amount of time the node can run before depleting its power supply.

Bridging a sensornet-based SOA with the Internet has been realized in the CodeBlue project [26] in which sensors used for healthcare monitoring are able to relay data to a Web service. This provides a convenient mechanism for transferring a patient's vital signs, obtained through an embedded sensor device, to a medical records system or monitoring alert center. CodeBlue's gateway application is similar to our own, with the exception that it translates sensor data into the HL7v3 format, a standard used for communicating medical information.

Dynamic software reconfiguration in sensor networks has been achieved in [9] by expressing system requirements as constraints on design space quality-of-service parameters. A run-time search of the design space is made possible by situating the reconfiguration controller on a powerful base station, a strategy which cannot be realized in resource-constrained sensor nodes such as the motes.

MiLAN [27] is a middleware which aids in WSN application development by optimizing the tradeoff between application quality of service and network resource utilization. Quality of service constraints are specified in graphs, which MiLAN interprets and uses to maintain a minimum set of active devices which provide the functionality required by the application. Although MiLAN employs a dynamic service configuration mechanism similar to that of OASiS, it only assists the application developer in managing quality of service, and is not a complete macroprogramming framework.

9 Conclusion

We have developed OASiS[1] an object-centric, service-oriented programming model and middleware for ambient-aware sensor network applications. Our service-oriented model permits the composition of any type of dataflow application. Upon detection of an external event, the sensor network instantiates a unique logical object which then drives the application. Application functionality is bundled in modular, autonomous services distributed across the network, and overall behavior is specified by a service graph. Dynamic service configuration is employed at run-time to locate and bind these services. This process involves an efficient search of the design space to ensure all constraints have been satisfied. In addition, a Gateway application, deployed on a base station, permits the sensor network to discover and access Web services. This capability provides a substantial benefit to WSN applications, as they are able to perform computations and access information using methods unavailable to resource-constrained sensor nodes.

The utility of our programming model was demonstrated with a simple indoor heat-source tracking application. A service graph was composed consisting of sensing, Web, and computational services, and the application deployed. Our results indicate not only the feasibility of our approach, but the benefits of using a sensornet-based SOA and dynamic service configuration as well.

The ambient-aware behavior of our programming model can be further developed to react gracefully to communication failures and node dropout during application execution. This will involve failure detection, isolation, and recovery mechanisms that restore the network application to a stable configuration both quickly and efficiently. This can be achieved by taking advantage of the OASiS service-oriented architecture. By placing an upper bound on the execution time of the service graph and appending a token to service messages that traverse the graph, a failure will be assumed if the token does not come full circle. The object can then initiate dynamic service configuration, reconstruct the service graph, and resume application execution. This is the subject of our current work.

References

[1] J. Liu, F. Zhao, Towards semantic services for sensor-rich information systems, in: Basenets, 2005.

[2] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-integrated development of embedded software, in: Proc. IEEE, Vol. 91, 2003.

[3] L. Luo, T. Abdelzaher, T. He, J. Stankovic, Envirosuite: An environmentally immersive programming system for sensor networks, in: TECS, 2006.

[4] Web services architecture, '`http://www.w3.org/TR/ws-arch/`'.

[1] The source code for the OASiS programming framework, including the application used in our case study, can be found on our project website at `http://www.isis.vanderbilt.edu/Projects/OASiS/`.

[5] E. Cheong, J. Liebman, J. Liu, F. Zhao, Tinygals: a programming model for event-driven embedded systems, in: SAC, 2003.

[6] P. Engelstad, Y. Zheng, Evaluation of service discovery architectures for mobile ad hoc networks, in: WONS, 2005.

[7] J.-C. Regin, A filtering algorithm for constraints of difference in CSPs, in: AAAI, 1994.

[8] L. J. Guibas, Sensing, tracking, and reasoning with relations, in: IEEE Signal Processing Magazine, 2002.

[9] S. Kogekar, S. Neema, B. Eames, X. Koutsoukos, A. Ledeczi, M. Maroti, Constraint-guided dynamic reconfiguration in sensor networks, in: IPSN, 2004.

[10] D. B. Johnson, D. A. Maltz, Dynamic source routing in ad hoc wireless networks, in: Mobile Computing, Kluwer Academic Publishers, 1996.

[11] Universal description, discovery, and integration, '`http://www.uddi.org`'.

[12] Soap, '`http://www.w3.org/TR/soap/`'.

[13] Web service description language, '`http://www.w3.org/TR/wsdl/`'.

[14] U.c. berkeley, '`http://www.tinyos.net/scoop/special/hardware\#mica2`'.

[15] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, D. Culler, The emergence of networking abstractions and techniques in tinyos, in: NSDI, 2004.

[16] E. Cheong, J. Liu, galsc: A language for event-driven embedded systems, in: DATE, 2005.

[17] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler, The nesc language: A holistic approach to networked embedded systems, in: PLDI, 2003.

[18] Apache web services, '`http://ws.apache.org/`'.

[19] G. Welch, G. Bishop, An introduction to the kalman filter, Tech. Rep. TR 95-041, Department of Computer Science, University of North Carolina at Chapel Hill (2004).

[20] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, T. He, L. Luo, S. Son, R. Stoleru, J. Stankovic, A. Wood, Envirotrack: Towards an environmental computing paradigm for distributed sensor networks, in: ICDCS, 2004.

[21] G. Simon, P. Volgyesi, M. Maroti, A. Ledeczi, Simulation-based optimization of communication protocols for large-scale wireless sensor networks, in: IEEE Aerospace Conference, 2003.

[22] S. Hadim, N. Mohamed, Middleware: Middleware challenges and approaches for wireless sensor networks, in: IEEE Distributed Systems Online, Vol. 7, 2006.

[23] A. Bakshi, V. Prasanna, J. Reich, D. Larner, The abstract task graph: A methodology for architecture-independent programming of networked sensor systems, in: EESR, 2005.

[24] C.-L. Fok, G.-C. Roman, C. Lu, Rapid development and flexible deployment of adaptivewireless sensor network applications, in: ICDCS, 2005.

[25] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, W. D. Meuter, Ambient-oriented programming, in: OOPSLA, 2005.

[26] S. Baird, S. Dawson-Haggerty, D. Myung, M. Gaynor, M. Welsh, S. Moulton, Communicating data from wireless sensor networks using the hl7v3 standard, in: BSN, 2006.

[27] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, M. A. Perillo, Middleware to support sensor network applications, in: IEEE Network, Vol. 18, 2004.