OASiS: A Service-Oriented Architecture for Ambient-Aware Sensor Networks*

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

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, Tennessee 37235, USA
{xenofon.koutsoukos,manish.kushwaha,isaac.amundson
sandeep.neema,janos.sztipanovits}@vanderbilt.edu

Abstract. Heterogeneous sensor networks are comprised of ensembles of small, smart, and cheap sensing and computing devices that permeate the environment, as well as resource intensive sensors such as satellite imaging systems, meteorological stations, and security cameras. Emergency response, homeland security, and many other applications have a very real need to interconnect these diverse networks and access information in real-time. Web service technologies provide well-developed mechanisms for exchanging data between heterogeneous computing devices, but they cannot be used in resource-constrained wireless sensor networks. This paper presents OASiS, a lightweight service-oriented architecture for sensor networks, which provides dynamic service discovery and can be used to develop ambient-aware applications that adapt to changes in the network and the environment. An important advantage of OASiS is that it allows seamless integration with Web services. We have developed a middleware implementation that supports OASiS, and a simple tracking application to illustrate the approach. Our results demonstrate the feasibility of a service-oriented architecture for wireless sensor networks.

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 [1]. A sensor web is a heterogeneous collection of such networks, and can also include resource-intensive sensing platforms such as satellite imaging systems, meteorological stations, and security cameras. Such heterogeneous sensor networks can greatly benefit applications ranging from emergency response to homeland security [2], [3], [4].

* This work is partially supported by ARO MURI W911NF-06-1-0076, Microsoft External Research, and by NSF Grant CCR-0225610.

F. Kordon and O. Sokolsky (Eds.): Monterey Workshop 2006, LNCS 4888, pp. 125–149, 2007.
© Springer-Verlag Berlin Heidelberg 2007

At present, users wishing to deploy WSN applications must be adept at low-level sensor network programming, as well as implementing the necessary domain-specific functionality. These applications must be able to run on large networks with nodes that have varying capabilities, are manufactured and operated by different vendors, and are accessed by multiple clients exercising different functionalities. 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 [5]. 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.

This paper presents OASiS, an Object-centric, Ambient-aware, Service-oriented Sensornet programming model and middleware implementation for WSN applications. In the *object-centric* paradigm, the application programmer is presented with a layer of abstraction in which the phenomenon monitored by the sensor network is represented by a unique logical object which drives the application [6]. The model is *ambient-aware*, which enables the application to adapt to network failures and environmental changes by employing a dynamic service discovery protocol. OASiS is a *lightweight* framework which avoids the use of bulky XML-based messages found in Web service standards, however, it still provides a simple mechanism for Web service integration.

We have implemented a suite of middleware services for the Mica2 mote hardware platform [17] running TinyOS [18] to support OASiS. Key characteristics of our approach that can benefit the design of sensor network applications are:

- Dynamic service discovery and configuration for reacting to changes in the network due to failures and unreliable communication links.
- Application reconfiguration for reacting to changes in the behavior of the monitored phenomenon.
- Service deployment onto heterogeneous platforms using well-defined interfaces enabling a seamless integration.
- Real-world integration by incorporating spatial service constraints that are necessary to monitor physical phenomena.
- Data aggregation by using services which accept input from multiple sensor nodes.

The OASiS programming model can be used to build a wide variety of dataflow applications such as target tracking, fire detection and monitoring, and distributed gesture recognition. To demonstrate the feasibility and utility of OASiS, we have developed a simplified indoor tracking experiment, which monitors a heat source as it travels through the sensor network region. The application is comprised of services provided by several resource-constrained sensor nodes, but it also invokes a Web service provided by a remote server. By providing access to the Web service, we incorporate functionality into our WSN application that would otherwise be unattainable.

This paper is organized as follows. Section 2 overviews our programming model and Section 3 describes dynamic service configuration. Our middleware implementation is presented in Section 4. Section 5 presents a case study followed by a scalability analysis in Section 6. In Section 7, we compare our research to similar work that has recently appeared in the literature. Section 8 concludes.

2 The OASiS Programming Model

This section presents the OASiS programming model. To illustrate the model, we use an environmental monitoring example in which a network of chemical sensor nodes is deployed for detecting and tracking chemical clouds. Upon detection, the sensor network begins estimating the speed and heading of the cloud, and continues to do so until the cloud leaves the sensing region. This tracking data is forwarded to a base station, which alerts local emergency management officials.

2.1 The Object-Centric Paradigm

The entity that drives an *object-centric* application is the *physical phenomenon* under observation. In OASiS, the physical phenomenon is represented by a *logical object*, which is comprised of a finite state machine (FSM), a service graph for each FSM mode, and a set of state variables. Figure 1 illustrates this representation in the context of the chemical cloud example. The estimated position of the chemical cloud is maintained by the logical object using the state variables (e.g. the mean and variance of the center of the chemical cloud). Each FSM mode represents a specific behavior of the chemical cloud, (e.g. stationary or moving), and contains a service graph that represents a dataflow algorithm. The algorithm is executed periodically, and its output is used to update the state. When the behavior of the physical phenomenon changes, the logical object transitions to a new mode containing a different service graph.

The logical object is instantiated upon detection of a physical phenomenon of interest. This is achieved by comparing sensor data with an *object context*, which defines the detection conditions for the physical phenomenon. This comparison is made periodically at a frequency specified by a refresh rate. 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 [6], which is executed by each candidate node.

After object instantiation completes, exactly one node, referred to as the *object node*, is elected owner of the logical object. The logical 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 logical object state variables, if applicable, and configuring the new

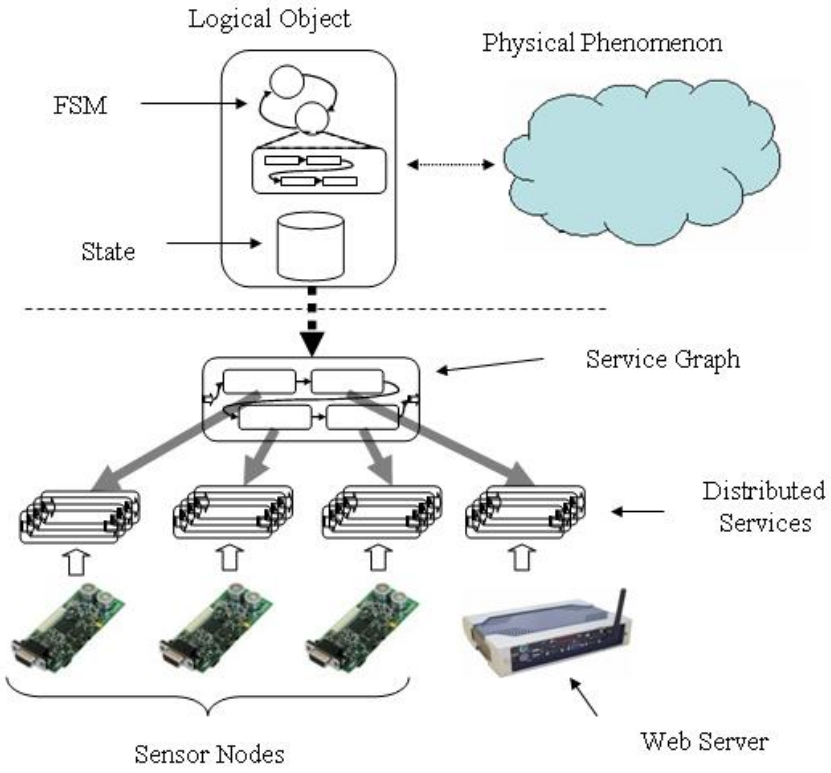

Fig. 1. OASiS: Programming Model

service graph corresponding to the new logical object mode. Because OASiS is a programming model for resource constrained WSNs, the FSM is intended to contain only a small number of modes representing a few broad behaviors of the physical phenomenon. We also assume the frequency of mode transitions will be much slower than the sampling rate required for tracking the phenomenon.

The logical object has a migration condition, which if evaluates *true*, invokes the object migration protocol. The selection policy for the migration destination is tied to the condition that triggers the migration protocol. In the above example, an increase in the variance of the location estimate can serve as a migration condition, and the owner selection policy will choose the node that is currently closest to the chemical cloud. Another migration condition could be a low power reading on the object-node, in which case the selection policy chooses a nearby 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. In this way the logical object follows the physical phenomenon through the sensing region. 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 physical phenomenon.

The goal of an object-centric programming model is to provide abstractions focusing on the physical 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. In addition, object-centric programming in OASiS facilitates dynamic service discovery and configuration by considering only a single neighborhood in the network and solving a localized constraint satisfaction problem. Details are discussed in Section 3.

2.2 Services in Sensor Networks

In OASiS, each mode in the logical object FSM contains a *service graph* whose constituent services provide the functionality necessary to update the state. Specifically, a service graph contains a set of services, a set of bindings, and a set of constraints, where a binding is a connection between two services, and a constraint is a restrictive attribute relating one or more services. We assume that the service graph is known a priori for each mode. Note that the service graph is simply a specification of an application and not the actual implementation. The implementation is provided by the services themselves, which may or may not be provided by the object node. *Services* are resources capable of performing tasks that form a coherent functionality from the point of view of provider entities and requester entities [7]. 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 one *output port*. Services are modular and autonomous, and are accessible by any authorized client application. For these reasons, services are typically stateless.

Figure 2 depicts the service graph for tracking a moving chemical cloud. Our localization algorithm requires chemical concentration measurements from sensor nodes surrounding the center of the cloud, and the current wind velocity in the region. Therefore, the service graph consists of Chemical Sensor services and one Wind Velocity service whose outputs are connected to the inputs of a Localization service. The Localization service uses a Kalman filter [8], and therefore requires current state variables obtained via an input port connected directly to the object. Similarly, the Localization service passes the updated coordinates of the chemical cloud back to the logical object. The Localization service is also connected to a Notification service, which informs the emergency management agency of the cloud's current position.

Because services can be publicly accessible, an attempted invocation might be blocked due to mutual exclusion if the service is currently executing some shared resource. Our programming model accounts for this with a globally asynchronous, locally synchronous (GALS) model of computation [9]. GALS

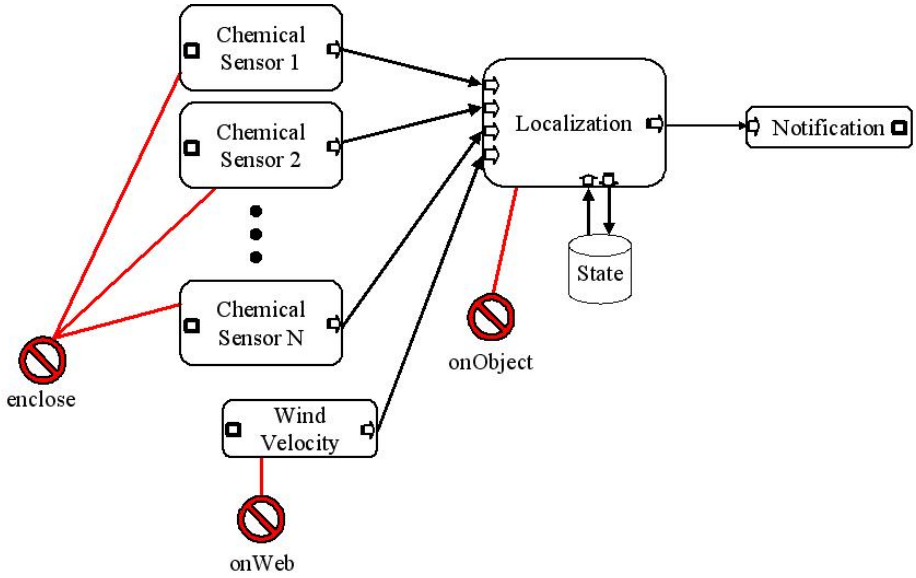

Fig. 2. Service Graph for Chemical Cloud Tracking Application

guarantees that communication between services will occur asynchronously (i.e. non-blocking), while intra-service communication such as method calls will exhibit synchronous (blocking) behavior. In this manner, a service never has to wait for its output to be consumed before processing data arriving on an input port. As such, GALS is an important and desirable paradigm for service-oriented applications for sensor networks.

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 resource-intensive services are implemented as *Web services*. We elect to use Web services due to their well-defined and documented standards. By taking advantage of Web services, applications have access to a wide range of functionality which would otherwise be unattainable. For example, our Localization service requires the wind velocity in the region. One option for obtaining the wind velocity is to equip a subset of sensor nodes with anemometers, however this approach can be cost-prohibitive and difficult to implement. Instead, we rely on an Internet-based *Wind Velocity* service, for example, one provided by the U.S. National Weather Service. The service interface definition is provided in a *Web Service Definition Language* (WSDL) file available on the host server. This provides us with the information necessary to access the Web service, including input and output parameters and their data types.

The modular and autonomous properties of services 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.

2.3 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 composing service graphs at run-time.

Typical constraints associated with a service 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 3. 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 is presented in Section 3.

(1) *Atomic property constraint:*

$$\textbf{service}.provider.p\ \textbf{op}\ K$$

where p is a property of the node providing **service**, **op** is a relational operator ($\textbf{op} \in \{>, \geq, <, \leq, ==, \neq\}$), 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 $a.provider.\text{Z} \geq 1$.

(2) *Compositional property constraint:*

$$F(provider.p)\ \textbf{op}\ K\ \textbf{over}\ \mathcal{S}$$

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 $\textbf{average}(provider.\text{POWER}) \geq 85\ \textbf{over}\ \{a, b, c\}$.

(3) *Atomic resource-allocation constraint:*

$$\textbf{service}.provider.type\ \textbf{op}\ \text{TYPE\_SET}$$

where $\textbf{op} \in \{==, \neq, \in, \notin\}$. For example, $a.provider.\text{ID} \notin \{NODE_1, NODE_2, NODE_3\}$ is used to ensure that service a does not run on a set of nodes with particular IDs.
(4) *Compositional resource-allocation constraint:*

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

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 $allSame(provider.\text{ID})$ **over** $\{a, b\}$ && $allDifferent(provider.\text{ID})$ **over** $\{a, c\}$. Similarly, more complex compositional resource-allocation constraints can be specified by using combinations of *allSame* and *allDifferent*.

2.4 Service Discovery and Composition

There are three types of events that will trigger service discovery and composition: object instantiation, mode transition, and migration. Object instantiation and mode transition are similar in that the logical object enters a new (possibly default) mode containing a service graph. For migration, the mode may not change, but the logical object is transferred to a new provider, which must parse the service graph in the current mode in order to execute it.

Before an object can start executing the 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 [10]. The SDP is provided as a service by each node and maintains a local *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 for a service that is not present in the SR, the SDP locates a new provider for that service. 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. 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 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*.

The objective of the Composer is to instantiate the configuration that satisfies the constraints specified in the service graph. These services are then *bound* together and eventually invoked. The ID of each service in the service graph is passed to the SDP. 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. 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. 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. The message is sent to the connection source node so that it may properly direct the output of its service to the input of the service specified by the connection destination. The Composer will not reuse bindings in the event a mode had been entered previously, because service graph constraints may no longer be valid.

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 [10]. 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.

Service discovery over *multiple hops* is achieved using a protocol similar to DSR [11]. There are three types of messages that require routing information: (i) service discovery reply messages, (ii) service binding messages, and (iii) service access messages. Service discovery request messages are flooded throughout the network, and therefore do not require any routing information. Routing information is maintained in a *next-hop table*, which stores the node ID of a known service provider, along with the ID of the next node along the multi-hop path to that provider. As a service discovery message travels from the object node to the service provider nodes, each intermediate node along the path records the ID of the preceding node. 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, specified a priori by the domain-service or application developer. At each intermediate node, a *hop-number* counter in the message header is incremented, and the message will not be forwarded once the counter reaches the maximum number allowed. Note that this maximum is the largest 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 the maximum number of hops from the object node on opposite sides. 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 another path should be selected.

3 Dynamic Service Configuration

This section describes dynamic service configuration that is required for reacting to changes in the network or in the behavior of the physical phenomenon.

3.1 Constraint Satisfaction

Service graph instantiation can be modeled as a constraint satisfaction problem [12], where services in the *abstract* service graph are the constraint variables, and the nodes that provide a particular service constitute the domain.

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 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.

2) Compositional Constraint Satisfaction

a) Compositional Property Constraints: The compositional property constraints are defined in Section 2, and involve the use of a composition function. OASiS includes several composition functions for aggregation, such as SUM, AVERAGE, and MEDIAN. In addition, we have defined a composition function called ENCLOSE for specifying the spatial configuration of sensor nodes. Many tracking applications employ localization algorithms which require measurement data to come from multiple sensors surrounding the physical phenomenon. The quality of the localization estimate often depends on how well the spatial configuration of these sensors is described. In the chemical cloud tracking example, three chemical concentration sensors are required, and they must be positioned such that they enclose the cloud. 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. For example, ENCLOSE(s_4.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. We model the ENCLOSE constraint based on the AM_I_SURROUNDED query described in [13]. 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 check by simple computation [13].

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 different enclosed regions.

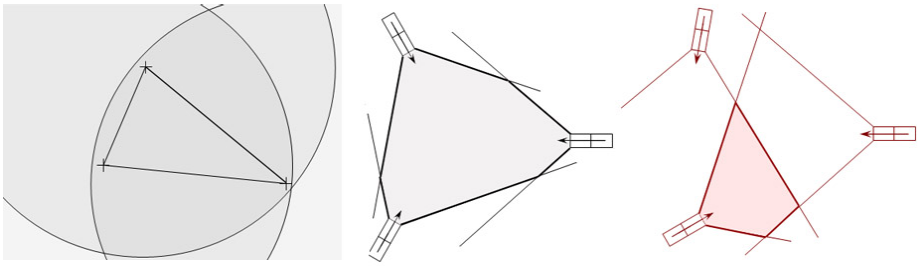

Fig. 3. Various definitions of ENCLOSE

b) Compositional Resource-Allocation Constraints: There are two types of composition functions for compositional resource-allocation constraints, *allSame* and *allDifferent*. Satisfying the *allSame* constraint is straightforward; the design space is the intersection of domains of all the participating constraint variables. To satisfy the allDifferent compositional constraint, a value is picked from the domain for each constraint variable. If the current set of values satisfy the constraint, a valid solution has been found. Otherwise, a backtracking algorithm [14] is used. The backtracking algorithm performs a depth-first search on the design space. Each leaf vertex represents a possible solution, assigning all constraint variables to a value. Non-leaf vertices are decision-points for constraint variables, where each path from the vertex assigns a value to the constraint variable. 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.

Algorithm 1 outlines the process of compositional constraint satisfaction. Lines 1-3 solve for constraints such as allSame as described above. The resulting pruned set is an exact set of solutions with respect to that constraint. In general, the pruned design space is an over-approximation that needs to be searched for a

valid solution. Lines 4-14 solve for other compositional constraints by exploring the design space and backtracking. Although solving CSPs can be computationally expensive, 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. 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 holds when services are redundantly distributed throughout the sensor network, and is reasonable for WSNs because 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. If a better solution is desired, it must be specified in the form of additional constraints on the service graph.

Algorithm 1. Compositional Constraint Satisfaction

```
1: for all C_i ∈ C do
2:    D̃ = prune_design_space(C_i, D)
3: end for
4: okay = FALSE
5: while !okay do
6:    sol = {(v_index_1, v_index_1, ..., v_index_1) | ∀i v_index_i ∈ D̃_i}
7:    okay = TRUE
8:    for all C_j ∈ C do
9:       if !satisfy(C_j, sol) then
10:         okay = FALSE
11:         backtrack()
12:      end if
13:   end for
14: end while
```

4 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 developer from the low-level complexities of sensor network 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.

4.1 Middleware Services

The middleware services include a *Node Manager*, *Object Manager*, and *Dynamic Service Configurator*. Figure 4 illustrates the relationship between the middle-

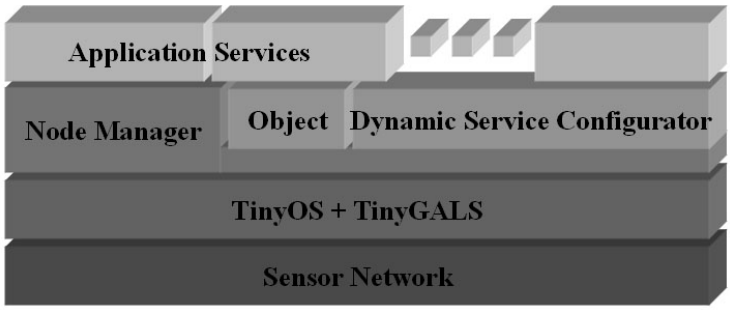

Fig. 4. Middleware

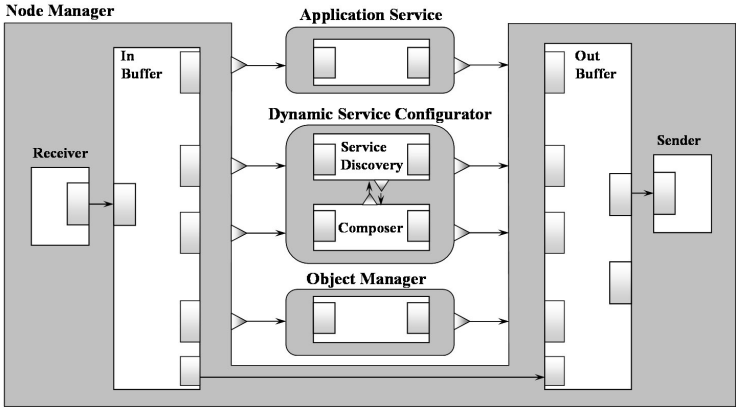

Fig. 5. Middleware architecture

ware and the sensor network, while Figure 5 illustrates the relationship between the application and middleware services at the sensor node level.

The Node Manager is responsible for message routing between services, both local and remote. This includes maintaining the multi-hop routing table and forwarding messages appropriately. The first eight 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 Dynamic Service Configurator contains the SDP and Composer, and functions as described in Section 2. 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.

4.2 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 sensor network base station and provides access to Web services by translating node-based byte sequence messages to the comparatively bulky XML-based messages used in Web service standards. As such, it is 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, a Web service standard used for locating and accessing services [15]. 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* messages [16]. If the service returns a value, it is also enclosed in a SOAP message. The Gateway composes and parses these 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 believe the Web service is being provided by the Gateway node. Note also that communication between the sensornet and Internet is bidirectional. 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 experience with wireless sensor networks to retrieve sensor data or run sensor network applications from a website with access to the OASiS Gateway.

To return to our tracking example, while the 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 SDP 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 to. 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*.

4.3 Implementation

Our middleware[1] is implemented on the Mica2 mote hardware platform [17] running TinyOS [18]. 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 [19], a GALS-enabled extension of nesC [20], 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 [21], 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 trade-off between an application's compactness and its robustness.

5 Case Study

We demonstrate the features of the OASiS programming model and middleware by developing a simplified indoor experiment which tracks a heat source.

[1] The source code for OASiS can be found on our project website at `http://www.isis.vanderbilt.edu/Projects/OASiS/`.

Table 1. Implementation Memory Requirements

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

5.1 Experimental Setup

Our experimental setup is shown in Figure 6. Five sensor nodes equipped with thermistors are placed in a region, each providing a set of pre-loaded services, and a heat source passes through the region. A sixth node is connected to a Web server that provides a Velocity service. For this simple indoor experiment, the velocity provided by the Web service is set to a constant 5 m/s which is approximately the velocity of heat source. Table 2 summarizes the sensor node attributes. The Localization service, implemented using an extended Kalman filter, estimates the position of the heat source from the sensor data. This estimate is then sent to the Notification service. The application is represented by a service graph as in Figure 2 with three temperature services that must reside on different nodes in a spatial configuration that encloses the heat source.

Table 2. Experimental Setup

Node ID	Position	Preloaded Services
101	[400 800]	Temperature, Notification
109	[700 400]	Temperature, notification
113	[0 500]	Temperature, notification, Localization
143	[200 0]	Temperature, notification
169	[800 1000]	Temperature, notification
base station	N/A	velocity estimation

5.2 Performance Evaluation

The feasibility and effectiveness of OASiS was evaluated by performing a set of experiments using the simple tracking application.

Experiment 1: Object creation and application execution. 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 ms, respectively, and depends on pre-defined timeout values; an *owner-election timeout* for object creation and a *service-configuration timeout* for service graph configuration.

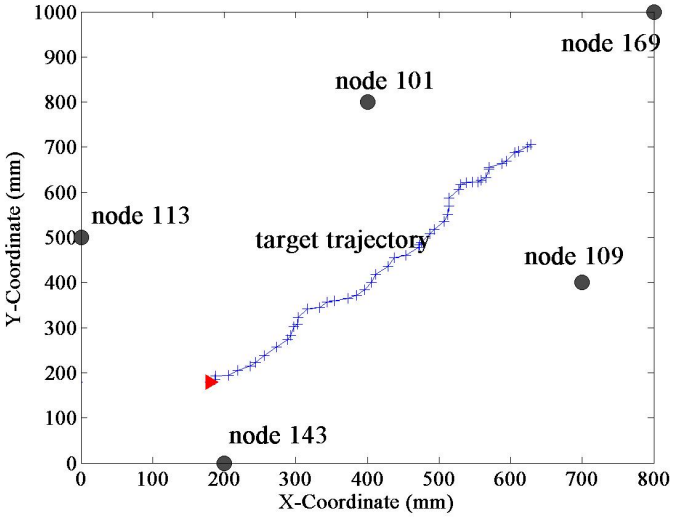

Fig. 6. Experimental setup

Table 3. Experiment 1 results

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

Experiment 2: Service disruption / Object migration. Once the physical object goes beyond the enclosure of nodes 109, 113, and 143, the variance in the location estimate starts to grow, which triggers object migration. As part of the migration protocol, node 143 begins a new owner election procedure by broadcasting a migration message. Nodes reply 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. In our experiment, node 143 sends the object to node 109. The number of messages communicated for object migration and service graph unbinding are summarized in Table 4. The delay for object migration is approximately 2000 ms. This experiment indicates that OASiS incurs an overhead on the number of messages required and the time delay for object creation, maintenance, migration and service graph maintenance. Table 4 indicates that the number of messages communicated is reasonably small.

Experiment 3: Tracking. Tracking performance was evaluated by comparing the actual heat source trajectory with the estimated trajectory. The tracking

Table 4. Experiment 2 results

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

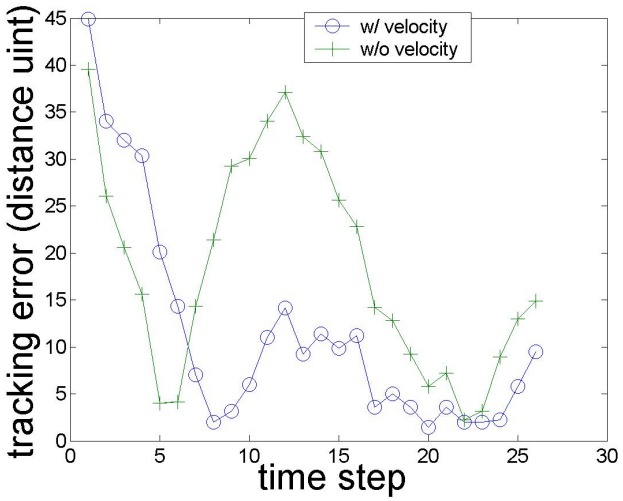

Fig. 7. Tracking results

accuracies for cases with and without estimated velocity data ($u_x = u_y = 0$) was also measured, and is summarized in Figure 7.

In all experiments, message transmissions were kept to a minimum due to the passive service discovery protocol. The service message size for this application requires only one transmission per message. 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 instead is provided separately to illustrate the overhead imposed on the system by adding Web service capability. It should be 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.

Table 5. Operation Response Times

Operation	Response Time (ms)	Standard Deviation
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

6 Scalability

To measure the effectiveness of our multi-hop service discovery protocol, we performed a scalability analysis using Prowler [22], a simulator for WSN applications. We simulated both grid and uniform random topologies. For each, we measured the message overhead of the service discovery protocol by considering (i) the number of message transmissions, (ii) the number of nodes discovered, and (iii) the time required for completing the service discovery.

When queried, each sensor node replies with a list of the services it provides. For our analysis, we measured the number of unique replies, which is an implicit measure of the number of services discovered. After service discovery completed, the total number of messages sent by all the nodes was tallied, along with the total number of discovered nodes and the total time required for 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. As expected, the number of message transmissions and discovered nodes increases quadratically with the number of hops, while the time taken for service discovery increases linearly. In addition, the number of message transmissions and discovered nodes increases linearly with node density in the network, while time taken for service discovery remains approximately constant.

We define the *discovery ratio* as the ratio of service discovery messages to the number of discovered nodes. Figure 11 shows the discovery ratio for different network topologies. From the results above, we can make some general useful observations. The discovery ratio increases linearly with the number of hops (i.e. the protocol requires approximately n discovery messages per discovered node for n-hop service discovery). Interestingly, the discovery ratio remains mostly constant with respect to node density. These results indicate that the service discovery protocol performs linearly for the number of discovery messages per discovered node with respect to the number of hops. Hence, the optimal number of hops for service discovery can be selected based on the distribution and number of services in the network.

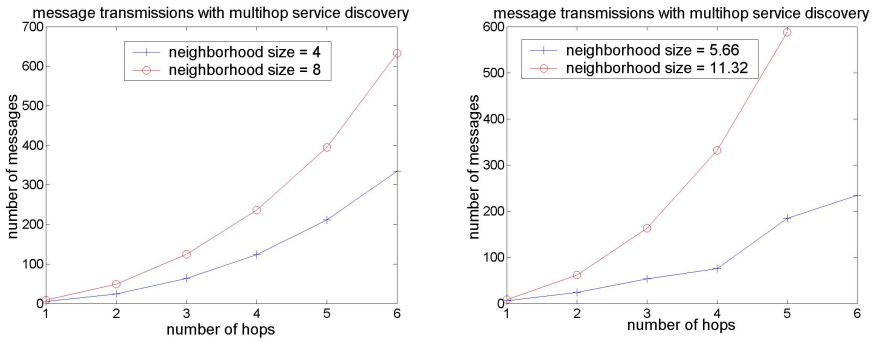

Fig. 8. Number of message transmissions for (a) grid and (b) random topology

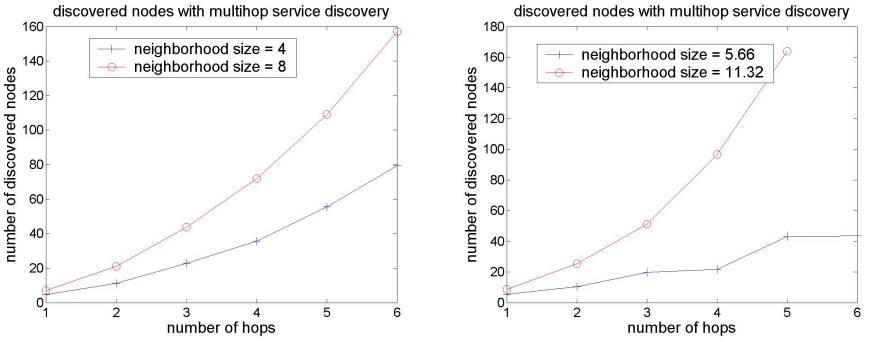

Fig. 9. Number of discovered nodes for (a) grid and (b) random topology

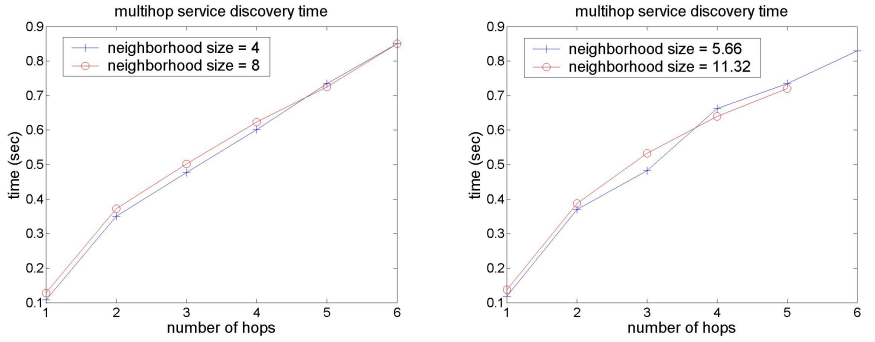

Fig. 10. Time taken for n-hop service discovery for (a) grid and (b) random topology

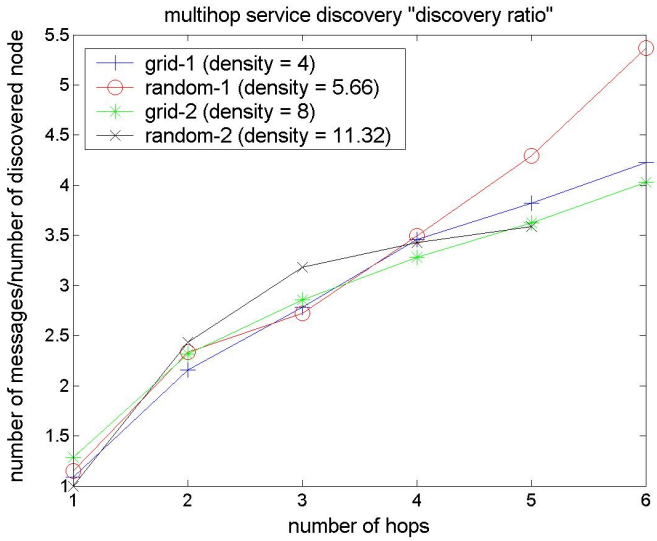

Fig. 11. Discovery ratio with number of hops for all four network topologies

7 Related Work

Design principles for traditional distributed computing middleware are not directly applicable to WSNs because sensor nodes are small-scale devices with limited resources, properties which directly affect computation, sensing, and communication. Recently, the WSN community has seen the emergence of a diverse body of macroprogramming languages, frameworks, and middleware that provide solutions to overcome these limitations (see [23] and the references therein). In the following, we focus on models and frameworks similar to OASiS.

SONGS [5] 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 [6]. 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) [24] 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 [25] 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 [26] 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 service-oriented architecture with the Internet has been realized with the CodeBlue project [27] in which sensors used for healthcare monitoring 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 station. 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 [28] 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.

MiLAN [29] is a middleware for WSN application development that optimizes the trade-off between application QoS 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 QoS, and is not a complete programming framework.

Spatial Programming [30] is a programming model for distributed embedded systems that abstracts the network into a single virtual address space. Nodes are referenced based on their location and provided functionality rather than ID, providing the application programmer with greater design flexibility in the presence of dynamic network topology. The authors implemented the spatial programming model using the Smart Messages [31] architecture in Java, and

deployed the application on Linux PDAs. It is unclear how such a model will perform on sensor nodes with tighter resource constraints.

8 Conclusion

We have developed OASiS, an object-centric, service-oriented programming model and middleware for ambient-aware sensor network applications. 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 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. Our results indicate service-oriented architectures are feasible and can benefit the design of sensor network applications.

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.

References

1. Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless Sensor Networks: A Survey. IEEE Computer 38(4), 393–422 (2002)
2. Yarvis, M., Kushalnagar, N., Singh, H., Rangarajan, A., Liu, Y., Singh, S.: Exploiting Heterogeneity in Sensor Networks. In: INFOCOM. Proceedings of the 24th Annual IEEE International Conference on Computer Communiation (March 2005)
3. Duarte-Melo, E., Liu, M.: Analysis of Energy Consumption and Lifetime of Heterogeneous Wireless Sensor Networks. In: Globecom. Proceedings of the 45th Annual IEEE Global Communications Conference (2002)
4. Lazos, L., Poovendran, R., Ritcey, J.A.: Probabilistic Detection of Mobile Targets in Heterogeneous Sensor Networks. In: IPSN. Proceedings of the 6th International Conference on Information Processing in Sensor Networks (2007)
5. Liu, J., Zhao, F.: Towards Semantic Services for Sensor-rich Information Systems. In: BaseNets. Proceedings of the 2nd IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks (2005)
6. Luo, L., Abdelzaher, T., He, T., Stankovic, J.: EnviroSuite: An Environmentally Immersive Programming System for Sensor Networks. ACM Transactions on Embedded Computing Systems 5(3), 543–576 (2006)
7. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web Services Architecture, `http://www.w3.org/TR/ws-arch/`

8. Welch, G., Bishop, G.: An Introduction to the Kalman Filter. Technical Report TR 95-041, Department of Computer Science, University of North Carolina at Chapel Hill (2004)
9. Cheong, E., Liebman, J., Liu, J., Zhao, F.: TinyGALS: A Programming Model for Event-driven Embedded Systems. In: SAC. Proceedings of the 18th Annual ACM Symposium on Applied Computing (2003)
10. Engelstad, P., Zheng, Y.: Evaluation of Service Discovery Architectures for Mobile Ad Hoc Networks. In: WONS. Proceedings of the 2nd Annual Conference on Wireless On Demand Network Systems and Services (2005)
11. Johnson, D.B., Maltz, D.A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: Imielinski, T., Korth, H. (eds.) Mobile Computing, Kluwer Academic Publishers, Dordrecht (1996)
12. Regin, J.C.: A Filtering Algorithm for Constraints of Difference in CSPs. In: Proceedings of the 12th National Conference on Artificial Intelligence, vol. 1 (1994)
13. Guibas, L.J.: Sensing, Tracking, and Reasoning with Relations. IEEE Signal Processing Magazine (March 2002)
14. Baase, S., Gelder, A.V.: Computer Algorithms: Introduction to Design and Analysis, 3rd edn. Addison-Wesley, Reading (1999)
15. Universal Description, Discovery, and Integration, `http://www.uddi.org`
16. SOAP, `http://www.w3.org/TR/soap/`
17. Mica2, `http://www.tinyos.net/scoop/special/hardware/#mica2`
18. Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Woo, A., Brewer, E., Culler, D.: The Emergence of Networking Abstractions and Techniques in TinyOS. In: NSDI. Proceedings of the 1st Symposium on Networked Systems Design and Implementation (2004)
19. Cheong, E., Liu, J.: galsC: A Language for Event-driven Embedded Systems. In: DATE. Proceedings of the Conference on Design, Automation and Test in Europe (2005)
20. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: PLDI. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (2003)
21. Apache Web Services, `http://ws.apache.org/`
22. Simon, G., Volgyesi, P., Maroti, M., Ledeczi, A.: Simulation-based Optimization of Communication Protocols for Large-scale Wireless Sensor Networks. In: IEEE Aerospace Conference (2003)
23. Hadim, S., Mohamed, N.: Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. IEEE Distributed Systems Online 7 (2006)
24. Bakshi, A., Prasanna, V., Reich, J., Larner, D.: The Abstract Task Graph: A Methodology for Architecture-independent Programming of Networked Sensor Systems. In: EESR. Workshop on End-to-end, Sense-and-respond Systems, Applications, and Services (2005)
25. Fok, C.L., Roman, G.C., Lu, C.: Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In: ICDCS. Proceedings of the 25th International Conference on Distributed Computing Systems (2005)
26. Dedecker, J., Cutsem, T.V., Mostinckx, S., D'Hondt, T., Meuter, W.D.: Ambient-oriented Programming. In: OOPSLA. Proceedings of the 20th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications (2005)

27. Baird, S., Dawson-Haggerty, S., Myung, D., Gaynor, M., Welsh, M., Moulton, S.: Communicating Data from Wireless Sensor Networks Using the hl7v3 Standard. In: BSN. International Workshop on Wearable and Implantable Body Sensor Networks (2006)
28. Kogekar, S., Neema, S., Eames, B., Koutsoukos, X., Ledeczi, A., Maroti, M.: Constraint-guided Dynamic Reconfiguration in Sensor Networks. In: IPSN. Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks (2004)
29. Heinzelman, W.B., Murphy, A.L., Carvalho, H.S., Perillo, M.A.: Middleware to Support Sensor Network Applications. IEEE Network 18(1), 6–14 (2004)
30. Borcea, C., Iyer, D., Kang, P., Saxena, A., Iftode, L.: Spatial Programming Using Smart Messages: Design and Implementation. In: ICDCS. Proceedings of the 24th International Conference on Distributed Computing Systems (2004)
31. Borcea, C., Iyer, D., Kang, P., Saxena, A., Iftode, L.: Cooperative Computing for Distributed Embedded Systems. In: ICDCS. Proceedings of the 22nd International Conference on Distributed Computing Systems (2002)