OASiS: A Programming Framework for Service-Oriented Sensor Networks

Manish Kushwaha, Isaac Amundson, Xenofon Koutsoukos, Sandeep Neema, Janos Sztipanovits
Institute for Software Integrated Systems (ISIS)
Vanderbilt University
Nashville, TN 37235, USA
{manish.kushwaha, isaac.amundson, xenofon.koutsoukos, sandeep.neema, janos.sztipanovits}@vanderbilt.edu

Abstract— **Wireless sensor networks consist of small, inexpensive devices which interact with the environment, communicate with each other, and perform distributed computations in order to monitor spatio-temporal phenomena. These devices are ideally suited for a variety of applications including object tracking, environmental monitoring, and homeland security. At present, sensor network technologies do not provide off-the-shelf solutions to users who lack low-level network programming experience. Because of limited resources, ad hoc deployments, and volatile wireless communication links, the development of distributed applications require the combination of both application and system-level logic. Programming frameworks and middleware for traditional distributed computing are not suitable for many of these problems due to the resource constraints and interactions with the physical world.**

To address these challenges we have developed OASiS, a programming framework which provides abstractions for object-centric, ambient-aware, service-oriented sensor network applications. OASiS uses a well-defined model of computation based on globally asynchronous locally synchronous dataflow, and is complemented by a user-friendly modeling environment. Applications are realized as graphs of modular services and executed in response to the detection of physical phenomena. We have also implemented a suite of middleware services that support OASiS to provide a layer of abstraction shielding the low-level system complexities. A tracking application is used to illustrate the features of OASiS. Our results demonstrate the feasibility and the benefits of a service-oriented programming framework for composing and deploying applications in resource constrained sensor networks.

I. Introduction

Wireless sensor networks (WSNs) are inherently dynamic in nature due to node mobility, limited resources, and unreliable communication links. It is therefore imperative that WSN applications consider these issues in order to ensure their correct execution. Unfortunately, tackling the problem of dynamic network behavior places a substantial burden on the programmer. In addition to an application's core functionality, the programmer must implement low-level operations such as robust communication protocols, resource management algorithms, and fault tolerance mechanisms. Unlike traditional network programming, WSN programming requires a coupling between application-level and system-level logic for resource management and sensing operations. Not only does the implementation of these components require substantial time and effort on behalf of the programmer, but it also increases the risk of deploying incorrect code, due to the often unanticipated behavior of distributed network applications.

In this paper, we present OASiS, an Object-centric, Ambient-aware, Service-oriented Sensornet programming framework and middleware which enables the development of WSN applications without having to deal with the complexity and unpredictability of low-level system and network issues. The framework provides a well-defined model of computation based on globally asynchronous locally synchronous dataflow [1], and is complemented by a user-friendly modeling environment. OASiS decomposes specified application behavior and generates the appropriate node-level code for deployment onto the sensor network.

In *object-centric* programming, an object is a logical element which represents some physical phenomenon being monitored by the network. The application is driven by the object, and its behavior is governed by the object's current state. The programmer is able to specify this behavior from the viewpoint of the object, and need not worry about how the behavior is implemented at the node level.

Object-centric programming by itself does not address the issues of network failures and dynamic network topology. *Ambient-aware* programming [2] has emerged as a paradigm for mobile computing in which each node in the network has up-to-date knowledge of its neighborhood. This includes awareness of its neighbors and the services they provide.

OASiS takes a *service-oriented* approach to behavioral decomposition and application execution. In a service-oriented WSN application, each activity (sensing, aggregation, service discovery, etc.) is implemented as a separate service. The advantages of using a service-oriented architecture for WSN applications are similar to those of Web services. Services are modular, autonomous, and have well-defined interfaces which allow them to be described, published, discovered, and invoked over a network. These properties permit services to be dynamically composed into complete applications.

We have developed a suite of middleware services which support such object-centric, ambient-aware, service-oriented applications. The services include components for managing internal sensor node operations, communication, service discovery, and object maintenance. The middleware serves as a layer of abstraction, shielding the programmer from the low-level complexities of sensor node operation.

The OASiS programming framework can be used to develop any type of dataflow application such as vehicle tracking, fire detection, 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. The case study highlights the various features of OASiS, and demonstrates the feasibility and utility of a service-oriented WSN programming model.

The paper is organized as follows. Section II describes the design principles and challenges involved in developing a programming framework for WSNs. In Section III, we describe the OASiS programming framework. Section IV describes the OASiS programming model, followed by a detailed description of the middleware in Section V. We demonstrate the capabilities of OASiS in Section VI. In Section VII, we compare our research to similar work that has recently appeared in the literature. Section VIII concludes.

II. Design Principles and Challenges

In this section, we discuss the challenges and design choices involved in WSN programming framework development. Until recently, WSN application programming has followed an approach that leaves the programmer with the responsibility of implementing many low-level details such as sensing, communication between nodes, and efficient use of energy. Not only does this extra work consume valuable time and resources, but it also increases the chances of delivering error-prone code.

A more robust approach makes use of a programming framework which provides higher levels of abstraction, enabling the developer to program from a global point of view. The developer can then focus on the desired overall behavior of the application, without having to be concerned with the complexities associated with distributed systems. Programming frameworks typically accept a desired application behavior as input and by means of compiler or interpreter, and often a framework API library, generate a functional network application as output. The output can either take the form of executable code or a byte-string instruction set to be interpreted by a network virtual machine. The application runs on top of a layer of middleware services which handle the underlying hardware and network operations.

Design principles for traditional distributed computing middleware are not directly applicable to WSNs for the following reasons:

- Sensor nodes are small-scale devices with a limited power supply, directly affecting computation, sensing, and (especially) communication.
- WSN middleware services often depend on the physical phenomenon being monitored.
- Node mobility, failure, and volatile communication links produce a dynamic underlying network.
- Variation in node hardware, computation, communication, and sensing abilities introduces heterogeneity.
- Sensor nodes often operate unattended for prolonged periods of time.

The design of a successful WSN middleware must address the various challenges imposed by the aforementioned characteristics. Many middleware challenges have already been identified [3] and are summarized below.

Limited resources: Every aspect of middleware design should attempt to include resource optimizations, and an analysis of resource utilization is important for preventing unintentional abuse. Power is of significant importance, especially in terms of communication where a single transmission consumes an amount of energy equivalent to more than a thousand computations.

Dynamic network topology: WSN topology is subject to frequent changes due to node mobility, node failure, and volatile communication links. A WSN middleware must support robust and reliable sensor network operations by providing mechanisms for fault tolerance, network awareness and application self-reconfiguration.

Heterogeneity: As WSNs become more prevalent and diverse, middleware will be required to operate gracefully in heterogeneous environments. This not only requires the middleware to maintain communication and distributed computation mechanisms across different architectures, but quality of service guarantees as well.

Real-world Integration: WSN applications interact with the physical world and react to an evolving environment over time and space. This requires middleware services which provide spatio-temporal abstractions, as well as real-time functionalities.

Application knowledge: Although middleware is intended to be application independent, knowledge of the application domain may improve overall performance. Therefore, providing middleware with parameterized services which accept configuration options as input will increase robustness and improve usability.

Data aggregation: The popularity of WSNs is due in no small part to their ability to accumulate environmental data over a wide physical area. However, the data sampled by each node is often meaningless until it is combined and analyzed. Middleware can play a key role by providing an efficient mechanism for collecting data located across the network.

Quality of service: Not only should middleware be capable of maintaining network-related quality of service guarantees, but also guarantees relating to the performance of the WSN application. Without such guarantees, application behavior becomes less stable and predictable.

III. The OASiS Programming Framework

This section describes the three stages of development within the OASiS programming framework, and the advantages of OASiS in view of the design challenges.

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 these *separation of concerns* (SoC). In software engineering, SoC is the process of breaking

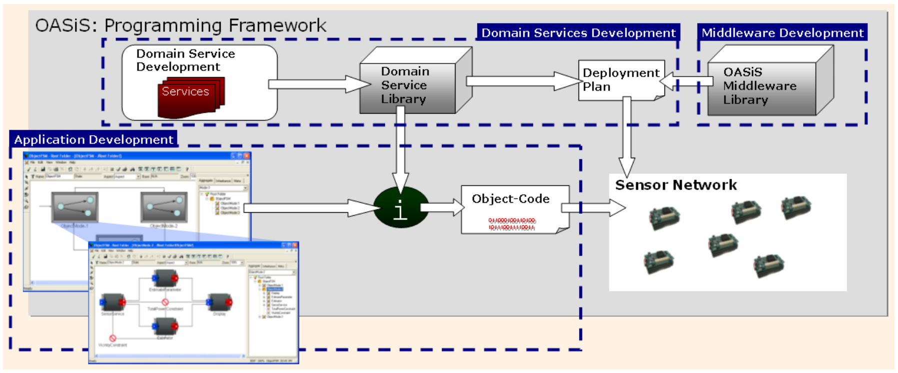

Fig. 1. OASiS: Programming Framework

a program into distinct features that overlap in functionality as little as possible [4]. In the application development context it can be redefined as the process of breaking the responsibilities of application development into distinct stages for programmers with different skills. OASiS is a programming framework that facilitates these 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. We present the OASiS middleware in detail in Section V. The *domain services* development layer provides domain-specific service libraries written by the domain experts, which 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 [5].

In addition to providing multilayer development, the programming framework addresses many of the aforementioned design challenges in Section II.

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 solved by our service-oriented approach, where well-defined services on *heterogeneous* platforms 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. Many localization algorithms, for example, require sensing services to be situated in a precise spatial configuration, such as surrounding the physical phenomenon. The ability to attach such constraints to services before they are invoked is an important aspect of WSN application programming. Again, due to the service-oriented approach of OASiS, we are able specify in-network *data aggregation* as services requiring inputs from multiple sensor services. OASiS supports specification of both application-specific and network *QoS requirements*. OASiS gracefully handles QoS violations by application re-configuration to satisfy QoS requirements.

IV. OASiS Programming Model

The OASiS programming model is organized into three key paradigms: service-oriented, object-centric, and ambient-aware programming.

Service-Oriented Architecture

A *service-oriented architecture* (SOA) simplifies WSN domain development by providing standards for data representation, service interface description, and facilitating service discovery. By wrapping application functionality into a set of modular services, a programmer can then specify execution flow by simply connecting the appropriate services together. The resulting service graph is a concise representation of the application, which can then be executed irrespective of the physical location of the distributed services.

In OASiS, a *service* is the basic unit of application functionality. Services have well-defined interfaces consisting of input and output ports, as well as the data types supported by each. A well-defined interface is important when binding two services together. A data type mismatch between the output port of one service and the input port of the next will cause an execution violation. OASiS catches these types of violations, preventing undesirable execution behavior.

Wiring a group of domain services together results in a *service graph* describing application flow. In OASiS, services can be wired together to compose a service graph representing any type of data flow application. In addition to services and their connections, a service graph can include constraints which restrict where and when a service can be invoked.

Because services must communicate asynchronously with each other, OASiS employs the globally asynchronous, locally synchronous (GALS) model of computation [6]. GALS maintains asynchronous communication between services, while intra-service communication, such as method calls, exhibit synchronous behavior.

Dynamic network topology 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. In OASiS, each node maintains two service repositories, a *local service repository* (LSR) that catalogs the application services available locally, and a *discovered services repository* (DSR) that catalogs the remote application services that have been discovered in the past. The combined use of these two repositories allows the system to be more responsive in the event of service unavailability.

Service discovery is passive in that service providers wait for a specific request before advertising a service. This passive approach was found to be the most energy efficient for mobile ad hoc networks with limited power resources [7]. Requests are flooded a limited number of hops throughout the network, and all providers of the requested service respond with a message specifying their node ID, physical location, and power level. This information can be used to discard any service provider which fails to satisfy the constraints specified in the service graph.

OASiS is intended for resource-constrained devices which are unable to use current Web service standards such as SOAP, WSDL, and UDDI due to their bulky XML-based messaging format. Instead, OASiS uses a compact byte-sequence message structure for accessing services and passing data between them. However, the OASiS SOA does provides a straightforward mechanism for accessing Web services, and similarly, for granting Web services access to the WSN application [8]. This capability allows WSN applications to perform computations and access information using methods unavailable to resource-constrained nodes.

Object-Centric Programming

Complexity in OASiS is minimized by providing the application developer with a means for specifying global network behavior; that is, behavior from the viewpoint of the physical phenomenon of interest. For example, the global network behavior for target tracking involves nodes taking sensor measurements of the environment until a target is detected, then organizing to localize the target and follow its movement. Global behavior does not involve specifying how nodes take their sensor measurements, which nodes communicate with each other once the target is detected, or how target localization is accomplished.

In OASiS, a physical phenomenon of interest is represented by a finite state machine (FSM), also referred to as the *logical object*. We selected 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. A service graph corresponding to an object FSM mode encapsulates the application behavior if the physical object is in a particular physical state. An example of object FSM is shown in the application development stage part in Figure 1.

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

$$\text{TEMPERATURE} \geq 100^{o}C.$$

The object context also contains information on 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 only one logical object is instantiated. To provide this guarantee, OASiS employs an object-owner election algorithm, similar to that of [9]. The object creation protocol, executed by each node, is outlined in Algorithm 1.

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
```

After the object creation protocol completes, exactly one node, referred to as 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 (see 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 migration 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 an object-node running low on power, in which case the selection policy is to select 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.

Ambient-Aware Programming

Nodes in an ambient-aware sensor network always have up-to-date knowledge of their neighborhood, including the services provided, and the properties of the nodes providing them. Ambient-awareness is built into OASiS to ensure efficient service discovery, minimal downtime in the event of communication failure or node dropout, and service graph constraint satisfaction.

OASiS allows constraints to be placed on services relating them to a specific physical location or to the relative locations of other services. For example, correct execution of an application may only be possible if sensing services are provided by nodes surrounding the physical phenomenon. Such real-world integration requires a dynamic service configuration mechanism. Dynamic service configuration instantiates the service graph by satisfying constraints, binding the services together, checking for any constraint violations, and reconfiguring the application when a violation is found.

We have identified several different types of constraints that can be placed in the service graph. Depending on the scope of the constraint, it can be atomic (applying to a single service) or compositional (applying to a group of services). Constraints can be further categorized as either property-based or resource-allocation-based. Property constraints specify a relation between the properties of the scoped services, while resource-allocation constraints define a relationship between nodes that provide the scoped services. For example, *the node providing* SERVICE-A *must have power level more than 85%* is an atomic property constraint, while SERVICE-A *and* SERVICE-B *must run on the same node* is a compositional resource-allocation constraint.

A significant constraint for real-world integration is *enclose*. 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 that host services s_1, s_2 and s_3. 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. The enclosed region in case of camera sensors with orientation and limited field-of-view (FoV) is the intersection of FoVs recorded by all member cameras.

We model service graph configuration as a constraint satisfaction problem (CSP). A feasible solution to the CSP is the host sensor node assignment for each service in the service graph. The main idea of the algorithm is to prune the service graph 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. Additional details on types, representation and satisfaction of service constraints can be found in [8].

V. The OASiS Middleware

We have developed a suite of middleware services that support the programming model features presented in Section IV. The OASiS middleware comprises a set of services which include a node manager, composer, service discovery protocol, and object manager. Figure 2 shows the relationship between the OASiS middleware and the sensor network. Figure 3 shows the connections between the different middleware services and domain services at the sensor node level. Note that

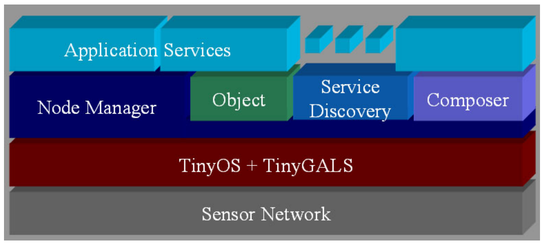

Fig. 2. Middleware

Figure 3 follows the TinyGALS [1] notation of actors, ports, components and interfaces. Each of the middleware service is implementation as TinyGALS actors. Actors asynchronously communicate with each other by putting a message in the event queue of input port connected to the output port. The scheduler removes the message from the event queue and calls the method linked to the input port with the message passed as its argument. Components residing inside actors communicate synchronously.

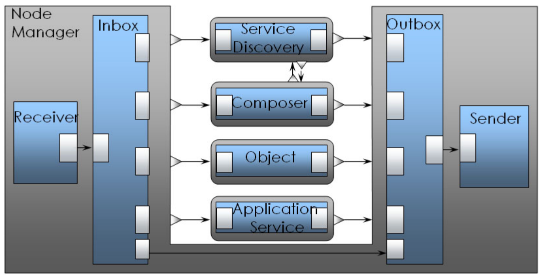

Fig. 3. Middleware node architecture

Node Manager: The Node Manager is responsible for message routing between services, both local and remote, which essentially makes it a dispatching service. Node manager sends and receives radio messages using components sender and receiver respectively. All messages handled by node manager consist of a control structure which contains source and destination node IDs (2 bytes each), source and

Service	Program memory (bytes)	Required RAM (bytes)
Node Manager	8500	367
Service Discovery Protocol	3858	313
Composer	8036	509
Object Manager	3560	151
GALSC queues & ports	702	1013
Total	40248	2820

TABLE I

IMPLEMENTATION CODE STATISTICS

destination service IDs (1 byte each), and message type (1 byte). The node manager resolves the appropriate destination for the message based on its control structure and forwards the message to the proper node.

Service Discovery: The Service Discovery component implements the service discovery protocol described in section IV. The component locates domain services by broadcasting a service request message. The request message contains the ID of the desired service, as well as the ID of the node requesting it. Any node providing the requested service will return a reply message containing its ID, and other node attributes. This information gets recorded in the discovered services repository for rapid future access.

Object Manager: 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 condition.Essentially, object manager implements the object-centric programming described in section IV.

Composer: The Composer is responsible for 1) parsing a service graph and sending service requests to the service discovery component, 2) instantiating a service graph by satisfying all specified constraints, and 3) creating a binding message for each instantiated service in the service graph. The binding message informs each service where to send its output data. The constraint satisfaction and dynamic reconfiguration functionalities of composer facilitates the ambient-aware programming paradigm of OASiS.

Prototype Implementation

We implemented our middleware on the Mica2 mote hardware platform [10] running TinyOS [11]. The code was developed using galsC [1], a GALS-enabled extension of nesC [12], the programming language for the motes. Table I lists each middleware service, with its code size and memory requirements. These memory requirements are sufficient for executing an application on the motes, which has approximately 128 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.

VI. CASE STUDY: HEAT-SOURCE TRACKING

As a concrete example of using OASiS, we consider a simplified indoor sensor network for tracking heat-source. The case study can be viewed as an example of a more general class of applications for tracking an object such as a vehicle or a chemical cloud. Tracking is a representative sensor network application that illustrates a number of interesting challenges, such as application-specific QoS metrics including detection accuracy and latency, in-network data aggregation, and ambient-aware adaptation.

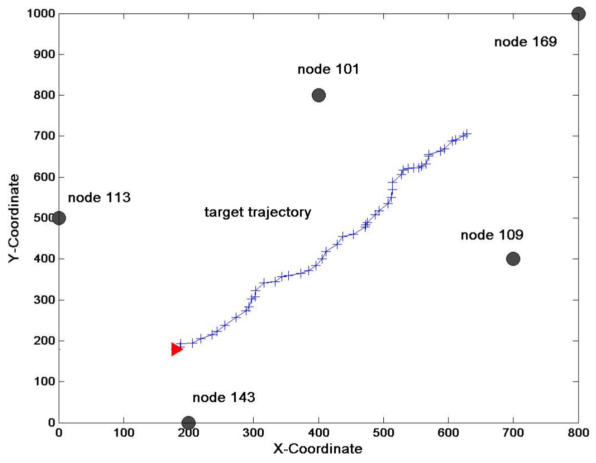

Fig. 4. Experimental Setup

Our experimental setup consists of the sensor network shown in Figure 4. There are 5 sensor nodes in the field each having a unique ID. Each node also contains a number of pre-loaded services. Table II 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
143	[200 0]	READ_TEMP, NOTIFY
113	[0 500]	READ_TEMP, NOTIFY, LOCALIZE_TRACK
109	[700 400]	READ_TEMP, NOTIFY
101	[400 800]	READ_TEMP, NOTIFY
109	[800 1000]	READ_TEMP, NOTIFY

TABLE II

EXPERIMENTAL SETUP

The object context is set to "TEMPERATURE $\geq$ 30". The object FSM consists of a single mode with a service graph shown in Figure 5. Sensor nodes equipped with a temperature sensing service, called READ_TEMP, are capable of sampling the local temperature, which indicates the proximity of the heat source. READ_TEMP is a wrapper service for the temperature sensing component on the sensor nodes. Once a heat source is detected, three temperature sensing services report their location and temperature measurement to a node running the localization service, called LOCALIZE_TRACK, which calculates the estimated heat source location based on the sensor input data and the previous estimate. For this experiment, we implemented the localization service as an extended Kalman filter. The localization service sends the location estimate to a notification service NOTIFY, which reports the estimated location to the user.

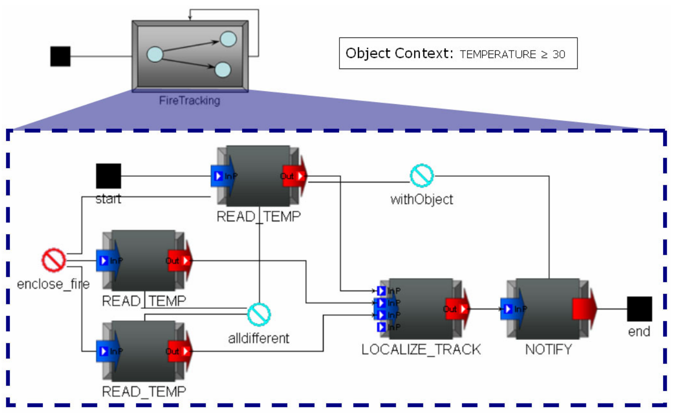

Fig. 5. Heat-source tracking application

Services have constraints associated with them. These constraints limit the number of allowable application configurations. The *alldifferent* constraint shown in Figure 5 on the three READ_TEMP services requires them to be on three different sensor nodes. The *enclose-fire* constraint requires the locations of the three READ_TEMP provider nodes to surround the heat source. By specifying such geometric constraints we can improve the probability of localization accuracy. Atomic constraints on all the services (not shown) require the provider nodes to have a minimum amount of energy, providing the application with a degree of power-awareness.

Other than property and resource-allocation constraints, quality of service (QoS) constraints can also be specified for an application. For our tracking example, the application-specific QoS constraint requires the variance of the location estimate from the localization service to be below a certain threshold.

Our goal is to demonstrate the feasibility and effectiveness of OASiS to create, maintain and migrate an object in the example object-centric tracking application. We present the number of messages communicated, which is an indicator of bandwidth utilization and energy consumption for the sensor node. We also present the delay for each stage which indicates the responsiveness of the application.

Experiment 1: Object creation and application execution

The heat source is originated at $[180, 180]$. Node 143 and 113 register temperatures higher than the detection threshold and start the object creation protocol. Since node 143 registered higher temperature, it is elected as the object-node and instantiates an object. The object then parses the object-code, retrieves the service graph for the current object mode, and initiates dynamic service configuration. Dynamic service configuration includes service discovery, constraint satisfaction and service binding. The application is configured by invoking instances of the READ_TEMP service on nodes 143, 113 and 109, the LOCALIZE_TRACK service on node 113, and the NOTIFY service on node 143. Once the service graph is configured, application execution commences. The number of message transmissions for object creation and application configuration is summarized in table III. 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 predefined timeout values; owner-election time-out for object creation and 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 III
EXPERIMENT 1 RESULTS

Experiment 2: Object migration

Once the physical object goes out of the *enclosure* of nodes 143, 113 & 109, 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 143 starts a new owner election 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 other services. In this case, node 143 sends the object to node 109. Table IV 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 IV
EXPERIMENT 2 RESULTS

Our 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 admissible time delays that are dependent on various timeout intervals exhibit the responsiveness of our ambient-aware OASiS.

VII. RELATED WORK

Recently, the WSN community has seen the emergence of a diverse body of macroprogramming languages, frameworks, and middleware which provide solutions for various aspects of sensor network programming [3].

SONGS [13] is a declarative service-oriented programming model which dynamically specifies application functionality in response to user-generated queries. While this technique works well as an information retrieval system, SONGS was not designed to alter its behavior based on a change in environmental conditions.

The object-centric paradigm has been successfully used in the EnviroSuite programming framework [9]. EnviroSuite and OASiS provide a similar layer of abstraction to the application developer, however by employing a SOA, OASiS is able to incorporate aspects of modular functionality, resource utilization, and ambient-awareness more efficiently.

Ambient-awareness is the primary focus of AmbientTalk [2]. The AmbientTalk programming language provides primitives for ambient-aware application behavior, however its use is intended for mobile ad hoc networked devices that are more resource-constrained than the motes.

Agilla [14] adopts a mobile agent-based paradigm. Autonomous agents, each with a specific function, are injected into the network at run-time, a technique referred to as *in-network programming*. For this approach, the underlying network middleware is uploaded once onto the node hardware, after which applications can be swapped out or reconfigured at any time, even after the nodes have been geographically dispersed in the field. In-network programming in Agilla is realized using a virtual machine instruction set architecture based on that of Maté [15].

The Abstract Task Graph (ATaG) [16] 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.

VIII. Discussion

The ability to specify application behavior from an object-centric viewpoint in terms of modular dataflow blocks makes OASiS a powerful tool for WSN programming. The use of a service-oriented architecture benefits all aspects of the application development process. Application functionality is neatly distributed across the network as services with well-defined interfaces which are easily discovered and invoked. Similarly, middleware functionality is bundled into services which use the same invocation mechanisms as their domain service counterparts. The service-oriented architecture provides efficient solutions for many WSN programming challenges. Dynamic network topology, heterogeneity, and data aggregation issues are simplified due to service modularity and autonomy. By following the object-centric programming paradigm, OASiS provides the application developer with a layer of abstraction that simplifies programming and reduces the opportunity for deploying error-prone code. Object-centric programming is a logical approach because most WSN applications are driven by the state of the physical object. The object's internal functionality such as creation, migration, and destruction are handled by the run-time system, leaving the developer to focus solely on the application's overall behavior. Programming from the viewpoint of the object also facilitates real-world integration and maintaining quality of service guarantees. This is because the application is *driven* by the object, which monitors and reacts to changes in the environment. The ambient-aware OASiS middleware supports application development by providing protocols for service discovery, constraint satisfaction, and communication. Furthermore, these protocols ensure efficient resource utilization and were designed to function in the presence of a dynamic network topology.

At present, the OASiS middleware only provides support for managing a single object at a time. This functionality can be greatly enhanced by enabling the detection of multiple objects, of multiple object contexts. Such a mechanism should be capable of disambiguating between two similar objects in close proximity to each other. The volatility of communication links in WSNs necessitates reliable communication failure detection and recovery mechanisms. Failures can occur for various reasons including network congestion, insufficient transmission power, transmission interference, and node dropout. Because most WSN applications must exchange data in an efficient and timely manner, handling these types of failures gracefully is essential. Although the OASiS middleware currently tackles certain aspects of dynamic network behavior, it requires a robust failure detection component to allow an application to continue execution in the presence of unpredictable network behavior. This is the subject of our current work.

References

[1] E. Cheong and J. Liu, "galsC: A Language for Event-Driven Embedded Systems," in *DATE*, 2005.

[2] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter, "Ambient-Oriented Programming," in *OOPSLA*, 2005.

[3] S. Hadim and N. Mohamed, "Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks," in *IEEE Distributed Systems Online*, vol. 7, no. 3, 2006.

[4] Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces. IBM Research. [Online]. Available: http://www.research.ibm.com/hyperspace/

[5] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," in *Proc. IEEE*, vol. 91, no. 1, 2003.

[6] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "TinyGALS: A Programming Model for Event-driven Embedded Systems," in *SAC*, 2003.

[7] P. Engelstad and Y. Zheng, "Evaluation of Service Discovery Architectures for Mobile Ad Hoc Networks," in *WONS*, 2005.

[8] I. Amundson, M. Kushwaha, X. Koutsoukos, S. Neema, and J. Sztipanovits, "Efficient Integration of Web Services in Ambient-aware Sensor Network Applications," in *BaseNets*, 2006.

[9] L. Luo, T. Abdelzaher, T. He, and J. Stankovic, "EnviroSuite: An Environmentally Immersive Programming System for Sensor Networks," in *TECS*, 2006.

[10] TinyOS Hardware Designs. U.C. Berkeley. [Online]. Available: http://www.tinyos.net/scoop/special/hardware#mica2

[11] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS," in *NSDI*, 2004.

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," in *PLDI*, 2003.

[13] J. Liu and F. Zhao, "Towards Semantic Services for Sensor-Rich Information Systems," in *Basenets*, 2005.

[14] C.-L. Fok, G.-C. Roman, and C. Lu, "Rapid Development and Flexible Deployment of AdaptiveWireless Sensor Network Applications," in *ICDCS*, 2005.

[15] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," in *ASPLOS X*, 2002.

[16] A. Bakshi, V. Prasanna, J. Reich, and D. Larner, "The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems," in *EESR*, 2005.