Efficient Integration of Web Services in Ambient-aware Sensor Network Applications[1]

Isaac Amundson, Manish Kushwaha, Xenofon Koutsoukos, Sandeep Neema, Janos Sztipanovits
Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, Tennessee 37235
{isaac.amundson, manish.kushwaha, xenofon.koutsoukos, sandeep.neema, janos.sztipanovits}@vanderbilt.edu

Abstract— **Sensor webs are heterogeneous collections of sensor devices that collect information and interact with the environment. They consist of wireless sensor networks that are ensembles of small, smart, and cheap sensing and computing devices that permeate the environment as well as high-bandwidth rich sensors such as satellite imaging systems, meteorological stations, air quality stations, and security cameras. Emergency response, homeland security, and many other applications have a very real need to interconnect such diverse networks and access information in real-time. While Internet protocols and Web standards provide well-developed mechanisms for accessing this information, linking such mechanisms with resource-constrained sensor networks is very challenging because of the volatility of the communication links.**

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

I. Introduction

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

Sensor nodes may have different and varying capabilities, be manufactured and operated by different vendors, and be accessed by multiple clients exercising different functionalities [1]. A *service-oriented architecture* (SOA) offers flexibility in the design of WSN applications since it provides accepted standards for representing and packaging data, describing the functionality of services, and facilitating the search for available services which can be invoked to meet application requirements. SOA deployment has already proved successful on the World Wide Web, however Web service technologies have been developed assuming standard Internet protocols and are not realizable in resource-constrained sensor networks.

In this paper, we present an object-centric, ambient-aware, service-oriented programming model for WSN applications. In the *object-centric* paradigm, the application programmer is presented with a layer of abstraction in which an event detected by the sensor network is represented as a logical object which then drives the application. Furthermore, our model is *ambient-aware*, enabling the application to adapt to network failures and environmental changes by employing a dynamic service discovery protocol. In this paper, we emphasize the service-oriented and ambient-aware aspects of our model, and do not discuss object-centric behavior in detail.

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

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

Our programming model can be used to build a wide variety of dataflow applications such as mobile vehicle tracking, fire detection and monitoring, and distributed gesture recognition. As proof of concept, we have developed a simplified indoor

[1]This work is partially supported by Microsoft Extended Research and by NSF Grant CCR-0225610

tracking experiment, which monitors a heat source as it travels through the sensor network region. We compose a service graph containing temperature-detection services, a localization service, a notification service, and a Web-based hypothetical wind velocity service. The case study demonstrates the feasibility and utility of a service-oriented WSN programming model. Furthermore, by providing access to Web services, we can incorporate functionality into our WSN applications that would otherwise be unavailable.

This paper is organized as follows. Section 2 describes our programming model. In Section 3, we give a detailed explanation of our method for performing constraint satisfaction in the context of dynamic service configuration. Our middleware implementation is discussed in Section 4. Section 5 presents the details of our case study. In Section 6, we compare our research to similar work that has recently appeared in the literature. Section 7 concludes.

II. Programming Model

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

A. Services in Sensor Networks

In an *object-centric* application, an *object* is a unique logical entity corresponding to a physical phenomenon under observation. The object is unique because it resides on only one node at a time. This is enforced by means of an object-owner election algorithm, similar to that of [2], when multiple nodes detect the same object context. This does not imply the object is confined to a single node over its entire lifetime; it has the ability to migrate between nodes as it follows its real-world counterpart. The network application is then driven by the object; that is, its behavior reflects the object's current state.

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

In our service-oriented architecture, the object contains a *service graph* whose constituent services provide the application with its functionality. Specifically, a service graph contains a set of services, a set of bindings, and a set of constraints, where a service is represented by a service ID and a port ID, a binding is a connection between two services, and a constraint is a restrictive attribute relating one or more services. In this work, we assume that an object's service graph is known a priori.

Figure 1 depicts the service graph used in our tracking application example. Our localization algorithm requires sensor data from three nodes surrounding the source event, in addition to the current wind velocity in the region. Therefore,

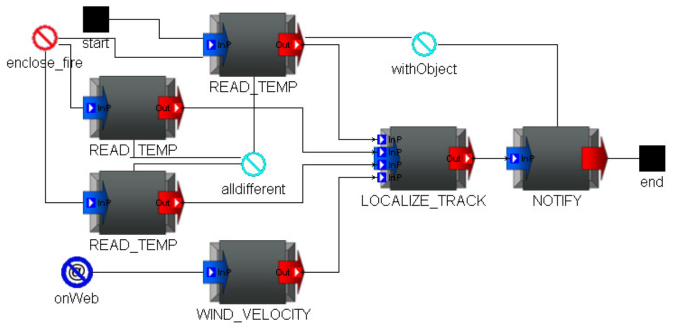

Fig. 1. Service Graph for Tracking Application

the service graph consists of the three Sensing services and one Wind Velocity service whose outputs are wired to the inputs of a Localization service. The Localization service is wired to a Notification service, which informs us of the source event's current position.

Services are resources capable of performing tasks that form a coherent functionality from the point of view of provider entities and requester entities [3]. They have a well-defined interface which allows them to be described, published, discovered, and invoked over the network. Furthermore, services are modular and function autonomously, providing an excellent mechanism for application reconfiguration during runtime. Each service can have zero or more *input ports* and zero or more *output ports*. For example, the Localization service in our tracking example has four input ports, three for sensor readings and one for wind velocity, and one output port, on which the position estimate is placed.

One necessary property of services is their ability to communicate asynchronously with each other. This is accounted for in our programming model by means of the globally asynchronous, locally synchronous (GALS) model of computation [4]. GALS guarantees that communication between services will occur asynchronously, while intra-service communication such as method calls will exhibit synchronous behavior. As such, GALS is an important and desirable feature of sensornet-based service-oriented applications.

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

B. 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 an important aspect of service graph creation.

The 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 discussed in more detail in Section III. 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.

These constraints are defined mathematically as follows, and our methods for solving them are presented in Section III.

(1) *Atomic property constraint:*

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

where, p is a node property, **op** is a relational operator ($\mathbf{op} \in \{>, \geq, <, \leq, ==, \neq\}$), and K is some constant value. For example, the constraint that service a must be provided by a node with power level greater than or equal to 85% is written as,

$$a.provider.\text{POWER} \geq 85$$

(2) *Compositional property constraint:*

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

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 height of nodes providing services a, b, and c must be one meter, the compositional property constraint is written as,

$$\mathbf{average}(provider.\text{POSITION.Z}) == 1000 \ \mathbf{over} \ \{a, b, c\}$$

(3) *Atomic resource-allocation constraint:*

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

where, $\mathbf{op} \in \{==, \neq, \in, \notin\}$. For example, suppose we want to ensure that service a does not run on a set of nodes with particular IDs. We can then say,

$$a.provider.\text{ID} \notin \{NODE_1, NODE_2, NODE_3\}$$

(4) *Compositional resource-allocation constraint:*

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

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

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

C. Service Discovery and Composition

Before an object can start executing a service graph, a *Service Discovery Protocol* (SDP) is invoked to determine which nodes in the network provide which services. Our model employs a passive discovery protocol in which a provider advertises a service only when a request for that service has been received [5]. This optimization minimizes the overall number of message transmissions, thus also minimizing power consumption. The SDP maintains two service repositories, a *Local Service Repository* (LSR), which catalogs the application services running locally, and a *Discovered Services Repository* (DSR), which catalogs remote application services that have been discovered in the past. Should an entry become stale due to communication failure or node dropout, for example, or a new service request arrives, the SDP locates a provider for that service by performing the steps outlined in Algorithm 1. Note that the algorithm searches both service repositories in order to obtain a complete list of known providers.

Algorithm 1 Service Discovery Protocol

```
1:  Input: Service ID
2:  search the Local Service Repository
3:  if Service ID is found in LSR then
4:      send local Service Info to Composer
5:  end if
6:  search Discovered Services Repository
7:  if Service ID is NOT found in DSR then
8:      compose Service Discovery Message
9:      broadcast Service Discovery Message
10:     receive Service Discovery Reply message
11:     record service provider node ID in DSR
12: end if
13: send remote Service Info to Composer
```

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

It is the Composer's job to produce a set of services and service providers that satisfy the constraints specified in the service graph. These services are then *bound* and eventually invoked. The Composer's behavior is outlined in Algorithm 2. The ID of each service in the service graph is passed to the SDP (lines 3-5). Because several instances of the same service could be residing on multiple nodes across the network, the Composer can expect multiple replies. As replies arrive, the Composer checks to see that any atomic service graph constraints are satisfied, and if so, the node information is stored (lines 6-9). Compositional constraint satisfaction commences after all replies have been received. Finally, the connections between the services in the service graph are examined, and a *service binding message* is created for each (line 12). The binding message simply 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 (line 13) so that it may properly direct the output of its service to the input of the service specified by the connection destination.

Algorithm 2 Composer

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

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

III. Ambient-aware Programming

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

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

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

A. Constraint Satisfaction

Service graph instantiation can be modeled as a constraint satisfaction problem, where services in the *abstract* service graph are the constraint variables, and the nodes that provide a particular service constitute the domain. The constraint satisfaction problem (CSP) is formally defined in [6].

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

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

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

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

1) Atomic Constraint Satisfaction: Atomic constraints are straightforward to satisfy. Because each atomic constraint is defined on a single variable, pruning the domain of that variable will leave the domain consistent, and hence satisfy the constraint. In Algorithm 3, the resulting pruned domain $\tilde{D}_i$ for constraint variable x_i is consistent.

Algorithm 3 Atomic Constraint Satisfaction

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

2) Compositional Constraint Satisfaction: Algorithm 4 outlines the process of compositional constraint satisfaction.

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

Many tracking applications employ localization algorithms which require measurement data to come from multiple sensors surrounding the physical phenomenon of interest. The quality of the localization estimate often depends on how well the spatial configuration of these sensors is described. We have therefore defined an additional composition function called ENCLOSE which is useful for specifying the spatial configuration of sensor nodes. For example, in our tracking application we use ENCLOSE to specify that we would like to have at least three different sensor nodes enclosing the tracked phenomenon at all times. The constraint ENCLOSE(L) *over* $\mathcal{S} =$

Algorithm 4 Compositional Constraint Satisfaction

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

$\{s_1, s_2, s_3\}$, specifies that the location L must be enclosed by the sensor nodes which provide services s_1, s_2 and s_3. The enclosure location, L can be specified as a fixed location or as a node ID. For example, ENCLOSE(s_4.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 [7]. We model the ENCLOSE constraint based on the AM_I_SURROUNDED query described in [7]. 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 the following: ENCLOSE(L) *over* $\{s_1, s_2, s_3\} \Rightarrow$ CCW(L, s_1, s_2) & CCW(L, s_2, s_3) & CCW(L, s_3, s_1), where CCW(a, b, c) specifies that locations a, b, and c form a counter-clockwise-oriented triangle in 2-D. The geometric constraint CCW(L, s_3, s_1) is easy to satisfy by simple computation.

The definition of ENCLOSE varies for different sensor domains. For example, one domain can define an *enclosed* region to be the overlap of member sensing ranges. Consider another example of camera sensors with orientation and limited field-of-view. The enclosed region in this case is the intersection of fields of view recorded by all member cameras. Figure 2 illustrates various enclosed region definitions.

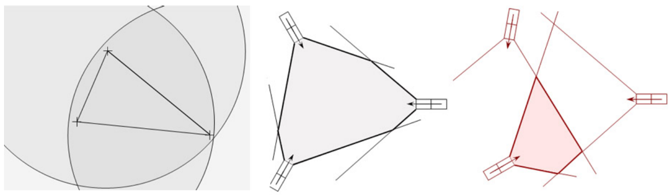

Fig. 2. Enclose Constraint

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

IV. Middleware

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

A. Services

The middleware services, which include a *Node Manager*, *Service Discovery Protocol*, and *Composer*, provide support to the object and application services. Figure 3 illustrates the relationship between our middleware and the sensor network, while Figure 4 illustrates the relationship between the the different types of middleware and application services at the sensor node level.

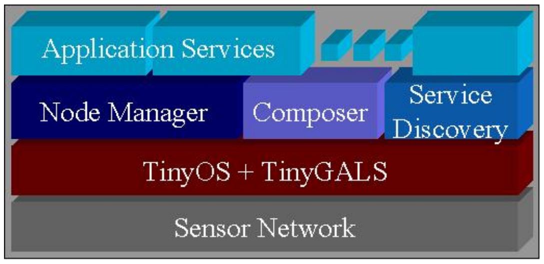

Fig. 3. Middleware

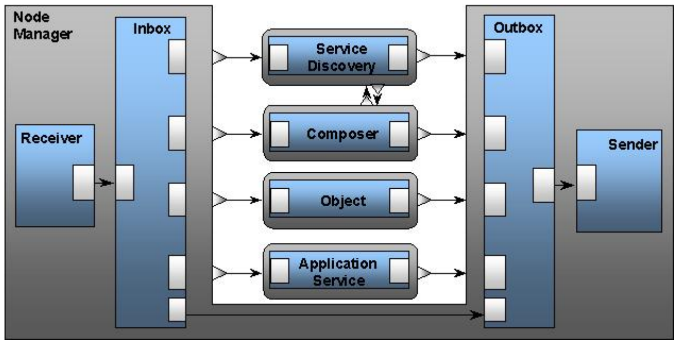

Fig. 4. Middleware node architecture

The Node Manager is responsible for message routing between services, both local and remote. 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), and message type (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 Service Discovery Protocol and Composer are wired into the Node Manager and function as described in Section II. Because these two services often operate on the same data, they share a dedicated channel, allowing them to bypass the Node Manager when communicating with each other. 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.

B. WWW Gateway

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

As such, it is also the job of the Gateway to speak the language of Web services. When a service discovery message arrives, the Gateway must locate this service on the Internet. This is accomplished by using the *Universal Description, Discovery and Integration* (UDDI) protocol [8], a Web service standard used for locating and accessing services. Given the proper keys, a UDDI inquiry returns the access point for a specific service as an URL string. Service access is achieved by means of XML-based *SOAP* [9] messages. If the service returns a value, it is also enclosed in a SOAP message. It is up to the Gateway to compose and parse these various XML messages and marshal the data appropriately when translating between the sensor network and the World Wide Web.

To return to our tracking example, suppose we could improve our localization estimate if we knew the present wind velocity. However, our sensor nodes are not equipped to take wind measurements, so instead we rely on an Internet-based *WindVelocity* service. The service interface definition is provided in a *Web Service Definition Language* (WSDL) [10] file available on the host. This provides us with the information necessary to access the Web service, including input and output parameters and their data types.

Service	Program memory (bytes)	Required RAM (bytes)
Node Manager	4100	330
Service Discovery Protocol	3846	183
Composer	3114	416
GALSC queues & ports	406	582
All	27962	2079

TABLE I
IMPLEMENTATION CODE STATISTICS

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

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

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*.

C. Implementation

Our middleware was implemented on the Mica2 mote hardware platform [11] running TinyOS [12]. The code was developed using galsC [13], a GALS-enabled extension of nesC [14]. The Gateway application was developed in Java. Our Web service implementation was realized using a suite of Apache services [15], including the Tomcat 5.5 web server, Axis 1.4 SOAP implementation, and jUDDI 0.9rc4, a Java-based UDDI implementation. In addition, MySQL 5.0 was used for the UDDI repository.

Table I 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 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.

V. CASE STUDY

Our programming model and middleware allow users who may not be WSN experts to develop robust sensor network applications. This permits emergency response teams, for example, to deploy a chemical cloud or fire tracking application on a sensor web. We demonstrate the features of our programming model and middleware by developing such an application.

Our setup consists of a simplified indoor sensor network application for tracking a heat source, as shown in Figure 5. The application takes periodic temperature readings from thermistor-equipped sensor nodes. Simultaneously, a Web service is accessed and the current wind velocity obtained. For purposes of this experiment, the wind velocity service returns predetermined values based on the location of the object. At each iteration, these data are processed by a Localization service which estimates the position of the heat source. This estimate is then sent to a Notification service, which reports the location estimate to the user. The service graph contains six services, as depicted in Figure 1. We have three instances of the Temperature-sensor service, and specify that each must reside on a different node and in a specific spatial configuration. Our Wind-velocity service is a Web service, which we specify via an atomic *onWeb* constraint.

The Localization service in this application was implemented using an extended kalman filter (EKF) [16]. The system state is a vector of heat source coordinates, $\mathbf{x} = [x\ y]^T$. The measurement vector is the collection of measurements from three temperature-sensor services. The system model that we use is represented by the equation,

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

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

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

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

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

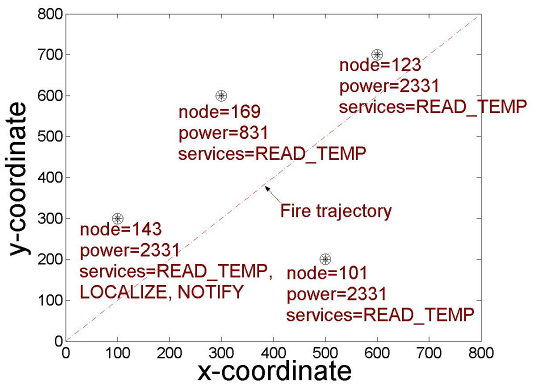

Fig. 5. Experimental setup

Operation	Response Time (ms)	Standard Deviation (ms)
Service discovery	4092	113
Service discovery w/o Web service	1.4	0.01
Constraint satisfaction	15	0
Service graph execution w/o Web service	81	13
Web service access	502	65
Localization service access	11	0

TABLE II
OPERATION RESPONSE TIMES

Application performance was evaluated by comparing the actual heat source trajectory with the tracked trajectory. The tracking accuracies for cases with and without wind velocity data ($u_x = u_y = 0$) was also measured. Figure 6 (a) and (b) shows the tracking results for tests with and without wind velocity data. Figure 6 (c) and (d) shows the tracking results with varied system and measurement noise parameters.

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

Response times for various operations were also obtained, and are displayed in Table II. The service discovery response time is provided with and without the Web service. Additionally, Web service access is not included in the service graph execution time, but is provided separately. This is to illustrate the overhead imposed on the system by adding Web service capability. It should be noted that our Web service implementation is not optimized for speed, however the current service discovery and constraint satisfaction latency is quite acceptable for performing dynamic service configuration. Similarly, the current access latency is acceptable for tracking slower-moving, wide-area phenomena such as chemical clouds and fire. Applications that require service graph execution at higher frequencies should not include Web service access in each iteration.

VI. Related Work

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

The object-centric paradigm has been successfully used in the EnviroSuite [2] programming framework. Envirosuite provides a high level of network abstraction, however its modularity can be enhanced by following a service-oriented approach for adding software components.

Ambient-aware computing [17] 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

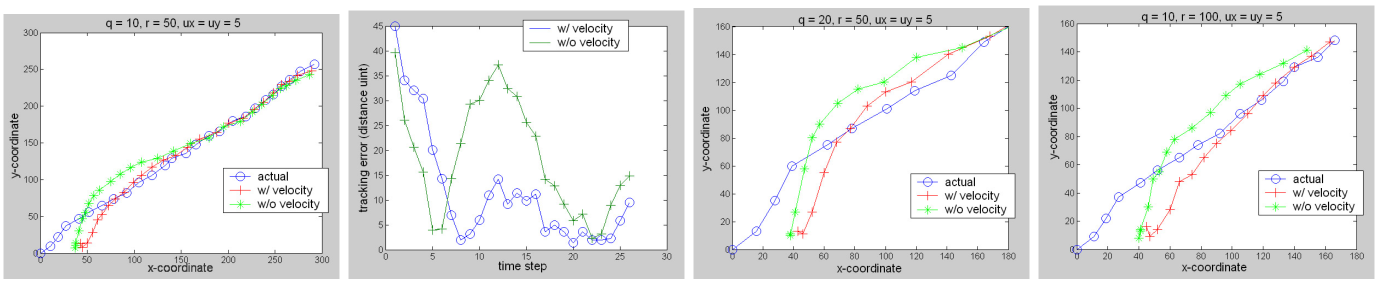

Fig. 6. Tracking results

networks consisting of resource-constrained nodes, communication with neighboring devices is often costly. Hence a tradeoff exists between the rate at which a node can update its understanding of the surrounding environment and the amount of time the node can run before depleting its power supply.

Bridging a sensornet-based SOA with the Internet has been realized in the CodeBlue project [18] in which sensors used for healthcare monitoring are able to relay data to a Web service. This provides a convenient mechanism for transferring a patient's vital signs, obtained through an embedded sensor device, to a medical records system or monitoring alert center. CodeBlue's gateway application is similar to our own, with the exception that it translates sensor data into the HL7v3 format, a standard used for communicating medical information.

Dynamic software reconfiguration in sensor networks has been achieved in [19] by expressing system requirements as constraints on design space quality-of-service parameters. A run-time search of the design space is made possible by situating the reconfiguration controller on a powerful base station, a strategy which cannot be realized in resource-constrained sensor nodes such as the motes.

VII. Conclusion

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

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

The object-centric behavior of our programming model can be further developed to respond to the different states of an object. This can be achieved by implementing the object as a finite state machine, each mode containing its own service graph. When the object transitions modes, the system undergoes dynamic service configuration based on the specification of the new service graph. Currently, dynamic service configuration only takes into account services provided by an object's one-hop neighbors. Clearly, there is a benefit to expanding the range to *n*-hops. However, the actual number of hops must be chosen carefully in order to minimize power consumption resulting from increased transmissions.

References

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

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

[3] Web Services Architecture. [Online]. Available: http://www.w3.org/TR/ws-arch/

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

[5] P. Engelstad and Y. Zheng, "Evaluation of service discovery architectures for mobile ad hoc networks," in *WONS*, 2005.

[6] J.-C. Regin, "A filtering algorithm for constraints of difference in CSPs," in *AAAI*, 1994.

[7] L. J. Guibas, "Sensing, tracking, and reasoning with relations," in *IEEE Signal Processing Magazine*, March 2002.

[8] Universal Description, Discovery, and Integration. [Online]. Available: http://www.uddi.org

[9] SOAP. [Online]. Available: http://www.w3.org/TR/soap/

[10] Web Service Description Language. [Online]. Available: http://www.w3.org/TR/wsdl/

[11] U.C. Berkeley. [Online]. Available: http://www.tinyos.net/scoop/special/hardware#mica2

[12] 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.

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

[14] 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.

[15] Apache Web Services. [Online]. Available: http://ws.apache.org/

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

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

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

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