Service-Oriented Sensor-Actuator Networks

Abdelmounaam Rezgui and Mohamed Eltoweissy, Virginia Tech

ABSTRACT

Most approaches developed to query sensor-actuator networks (SANETs) are either application-specific or generic. Application-specific SANETs provide limited reusability, are not cost effective, and may require extensive reprogramming efforts to make the network able to serve new applications. Generic SANETs usually require that a sizeable code be deployed on the nodes regardless of the specific requirements of the application at hand. More important, they may not be optimized to fully exploit the specific characteristics and query patterns of a given application. In this article we introduce service-oriented SANETs (SOSANETs) as a novel approach to building customizable SANETs. SOSANETs provide the benefits of both application-specific SANETs (e.g., energy efficiency, scalability) and generic SANETs (e.g., reusability) and avoid most of their limitations. We implemented our approach in TinySOA, a SOSANET developed on top of TinyOS. We conducted an evaluation of TinySOA that included a comparison with TinyDB, an established query processing system for sensor networks. The obtained empirical results show that TinySOA outperforms TinyDB in many aspects including energy consumption, scalability, and response time.

INTRODUCTION

Sensor networks have enabled a range of applications where the objective is to *observe* an environment and *collect* information about the observed phenomena or events. In many cases appropriate actions must be taken upon the occurrence of a given event (e.g., switching the light of a room off when it has been empty for more than five minutes or switching the light on when the presence of a human is detected). This has led to the emergence of a new generation of sensor networks, called sensor-actuator networks (SANETs), that have sensor nodes and *actuator* nodes.[1] Sensors and actuators communicate and collaborate to perform distributed sensing and acting tasks. Sensors gather information about the physical world, while actuators make decisions and perform actions that affect the environment [2]. Actuators are able to change parameters in their environment (e.g., temperature, light) as well as their intrinsic properties (e.g., location, speed, volume). Applications of SANETs include environmental applications (e.g., forest fire detection), business applications (e.g., inventory management), health applications (e.g., patient monitoring), home automation, and entertainment (e.g., interactive museums).

For years, SANETs have been *closed* networks deployed for *specific* applications with specific sets of characteristics. Typically, a single party (e.g., a government agency, research institution, private company) owns, maintains, and uses the SANET. As a consequence, in most early SANET deployments an ad hoc, application-specific architecture was adopted. In recent years the need to decouple SANETs from the applications using them has led to the emergence of *generic* SANETs, an alternative design model where an application-independent query system is deployed on the SANET. In this model the query system is designed to answer queries from any application. As SANETs evolve, they are expected to become open, ubiquitous, interoperable, multipurpose infrastructures. This would translate into new requirements not supported by existing architectures. We argue that next-generation SANETs require *customizable* architectures that provide developers the ability to select individual software components from several SANETs and integrate them in new applications that achieve higher levels of efficiency and scalability. We next elaborate on the inadequacy of current architectures and then introduce the proposed alternative of customized SANETs.

Application-specific SANETs: In application-specific SANET deployments, the application consists of a distributed code installed on some or all of the nodes of the network. In simple applications the same code is installed on all nodes. In more complex applications different code modules are installed on different nodes. This approach has several drawbacks. First, SANETs deployed for one or a few applications are often of limited reusability and are therefore inherently not cost-effective. This translates into a low return on investment. Another drawback is the tight coupling between the application and the underlying SANET. The application is often designed as a monolithic code of tightly coupled modules where each module implements a spe-

[1] *Some literature uses the term "actor" instead of "actuator."*

cific functionality, such as user interface, data access (i.e., retrieving sensor readings), or actuator activation. To develop these modules, programmers invoke functionalities at several layers in the SANET's architecture. The reason behind this monolithic application-specific design is often optimization. By enabling programmers to manipulate parameters and mechanisms at different layers, the code may be tailored to achieve better efficiency for the application at hand. A consequence of this tight coupling between the application and the SANET is that considerable reprogramming efforts are often necessary to make the network able to serve new applications.

Generic SANETs: Generic SANETs are not intended to be used by a specific application. They usually require that a generic code (i.e., the query processing system) be installed on all nodes of the network. Examples of generic query systems include Cougar [4], TinyDB [8], and REED [1]. This approach also has a number of limitations. First, the same code is installed on each node. A particular node may not need or be able to support all the functionalities of the installed query system. For example, a typical query system would include code for in-network data aggregation, collaborative event detection, actuation coordination, and so on. The latter functionality, for example, is not needed at a node with no actuation capabilities. As a result, a sizeable query processing system has to be installed on all nodes of the network regardless of their capabilities. A more important drawback of generic SANETs is that they must often trade efficiency for genericity. Typically, a generic query system may not be optimized to fully exploit the specific query patterns of a given application. Also, nodes in a SANET do not necessarily have the same hardware configuration. A generic query system may not be optimized to efficiently manage the hardware resources of specific nodes. As a consequence, generic SANETs may not scale to handle high query loads typical in next-generation, potentially Web-accessible SANETs.

As sensor technologies mature and new applications proliferate, current design models for sensor-actuator systems seem increasingly unable to cope with the requirements of the next generation of open, ubiquitous, interoperable, multipurpose SANETs. Architectures for future sensor systems will have to be able to serve different applications and adapt to different post-deployment query patterns. Networks from different providers will have to be individually programmed, yet able to interoperate efficiently. Both application-specific and generic architectures are obviously unable to satisfy these requirements. To enable next-generation sensor-actuator systems, new customizable architectures are needed.

Customizable SANETs: We define customizable SANETs as SANETs that are readily configurable, after they are deployed, to serve different types of applications with arbitrary query patterns. A node in a customizable SANET would expose its capabilities as identifiable resources that may be accessed by any entity that may communicate with the node and not necessarily by other nodes from the same network. Customizable SANETs would provide developers the flexibility to combine the resources provided by nodes in one or more (existing) SANETs to meet the requirements of new applications and yet expect the same levels of performance that would result from an application-specific deployment.

A possible alternative to building customizable SANETs is to use generic SANETs as their backbone and develop additional software layers that customize the functionalities of generic SANETs to satisfy the requirements of the given application. This, however, would only lead to further lower performance and memory availability. In this article we introduce *service-oriented* SANETs, or SOSANETs, as a novel approach to building customizable SANETs. In SOSANETs nodes' sensing and actuation capabilities are exposed to applications in the form of a collection of programmatic abstractions called *services*. A service deployed on a node is a lightweight code unit that provides some functionality supported by the node. These services may be individually invoked or combined in complex ways to form a *virtual* SANET with far richer sensing and actuation capabilities. In the proposed approach we deploy services directly on top of the operating system, and services are accessible directly by applications. We implemented this approach in TinySOA (service-oriented architecture), a prototype SOSANET built on top of TinyOS. Our evaluation of TinySOA shows that the proposed service-oriented approach is a viable alternative for building customizable SANETs.

We implemented this approach in TinySOA, a prototype SOSANET built on top of TinyOS. Our evaluation of TinySOA shows that the proposed service-oriented approach is a viable alternative for building customizable SANETs.

SERVICE-ORIENTED QUERY MODEL

In this section we present our service-oriented query model for SANETs. We consider a sensor-actuator network where nodes have heterogeneous sensing and actuation capabilities. Each node exposes its capabilities as services. Conceptually, a service is a computational component that:
- Has a unique network-wide identifier
- May be invoked asynchronously
- May have one or more parameters
- Produces one or more values as a result of invocation

A service may have multiple *service instances*, each running on a given node. The SOSANET has one or more base stations. Users query the SOSANET by submitting queries to one of its base stations or directly to individual nodes. Queries may be of two types:

Task queries: In a task query the application requests a *reading* to be retrieved from one or more sensors or an *operation* to be performed by one or more actuators. The result of a task query is the value of the reading or a code indicating the outcome of the actuation operation.

Event queries: In an event query the application requests to be notified when an *event* of interest occurs. Typically, the result of such an event query is a *notifier* message that informs the application of the occurrence of the event. A notifier may include additional information such

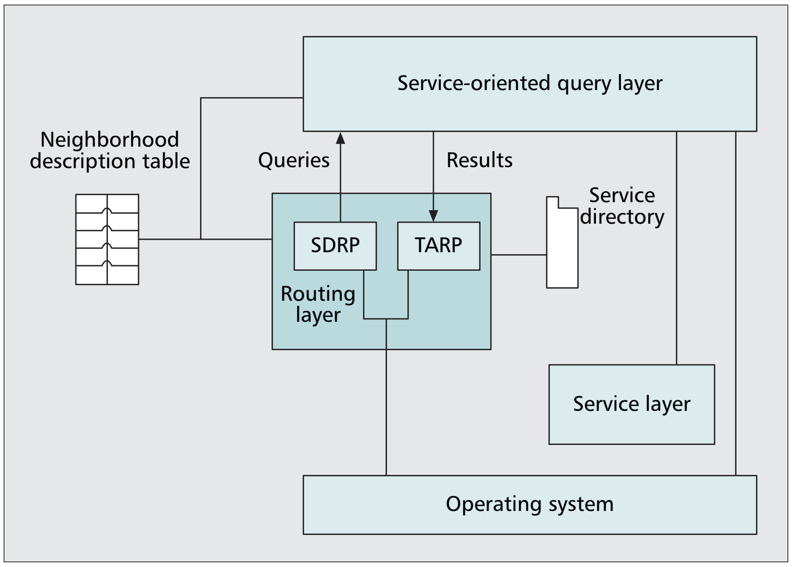


Figure 1. *Overview of a node's architecture in SOSANETs.*

as the time the event occurred, the geographical location where the event occurred, and so on.

Both types of queries may be one-time or recurrent. A recurrent query is a query an application submits to request that a sensing/actuation task be carried out or an event be detected repetitively with a given *frequency* and for a given *duration*.

Query Specification

We adopt an extended Event-Condition-Action (ECA) model for query specification. In general, queries specify five elements: an event, a condition, an action, a spatial scope, and a temporal scope. We use the acronym Event-Condition-Action-Spatial scope-Temporal scope (ECAST) to refer to our query model. A query in the ECAST model has the following syntax:

```
Query :: event <event>;
condition <condition>;
action <action>;
space <spatial scope specification>;
time <temporal scope specification>
```

where:

• <event> is the event that triggers the execution of the query's action. An event is a condition expressed in terms of attributes that may be sensed. For example, `event temp > 70 or light > 110` is an event that occurs when one of the given conditions becomes valid.

• <condition> is a condition that must be satisfied to execute the query's action. Note that, in queries that specify both an event and a condition, the condition is evaluated only when the event occurs; the event condition must be evaluated continuously to detect the occurrence of the event. The query's condition is also specified in terms of attributes, for example, `condition temp > 65`.

• <action> is an invocation of a service. Each service invocation specifies the service identifier and the values for the service parameters. For example, `action getLight()` requests that nodes involved in the query read and return the value of their light sensors.

• <spatial scope specification> specifies the geographical area where the sensing/actuation is to take place. For example, `space Circle` (n_i, 10ft) specifies all the nodes within 10 feet from node n_i.

• <temporal scope specification> specifies the start and end times for the query and, for recurrent queries, the query's frequency. The start (resp. end) time is the time at (or after) which the user wants the query execution to begin (resp. end). The frequency specifies the time that must elapse between two successive executions of the query. If the frequency is not specified, a new iteration of the query is started immediately upon completion of the current iteration; for example,

```
Query :: event temp >65 ;
condition humidity > 50%;
action getLight();
space Circle(n5, 10 ft);
time start 10:00am
end 5:00pm
frequency 100s;
```

This query requires all nodes within a 10 foot distance from node n_5 to repetitively read their light sensor when their temperature reading exceeds 65 if the condition humidity > 50 percent is true. The query also specifies a start time of 10:00 a.m. and an end time of 5:00 p.m. as well as a frequency of 100 s.

Service-Driven Query Routing

Query routing in SOSANETs is a distributed process in which several nodes cooperate in routing queries requesting services toward nodes providing those services. Typically, a query is initiated by a base station[2] and requests the invocation of a given service by a given subset of nodes. A fundamental idea in the proposed SOSANETs is to use service-driven routing (SDR) to efficiently deliver queries to their appropriate destinations. In SDR each node perceives another node's capabilities in terms of the services it provides. Each node maintains a data structure called a service directory (SD) that stores information about services provided by reachable nodes. Service directories are used much as routing tables are used in networking protocols. When a node receives a query requesting the invocation of a given service, it makes its routing decision for the received query based on the content of its service directory. We elaborate on this later, where we present SDRP, a service-driven routing protocol that routes queries efficiently in SOSANETs.

An Architecture for SOSANETs

Figure 1 shows an overview of a node's architecture that supports the proposed service-oriented query model. The software running on top of the operating system at each node is organized into three layers.

Service-Oriented Query Layer

The service-oriented query (SOQ) layer receives queries from the service-driven routing layer,

[2] *Note, however, that any node may initiate queries.*

interprets them, invokes the appropriate services specified in the queries, collects the results from the services, packages these results into query result messages, and submits those messages to the service-driven routing layer to send them to the query issuer. The SOQ layer consists of two main modules (Fig. 2).

Service invocation scheduling module (SISM): This module monitors the node's query load and schedules service invocations while considering the frequency and expiration time of the different queries. The SISM maintains a list of services to be invoked and the times of invocation in a service invocation schedule (SIS). This module also conducts multiquery optimization by exploiting any relationships that may exist between several queries. For example, a single service invocation may be relevant to several queries.

Event detection module (EDM): This module detects the events that are relevant to the current query load at the local node. As defined earlier, an event is a predicate expressed in terms of one or more attributes. The EDM maintains an *event list* of all the events relevant to the current query load. It also maintains a mapping between attributes and events. Each time a change is detected in the value of an attribute *a*, the EDM evaluates the event predicates whose value depends on the value of the attribute *a*. The EDM then activates all the queries whose event clause evaluates to true.

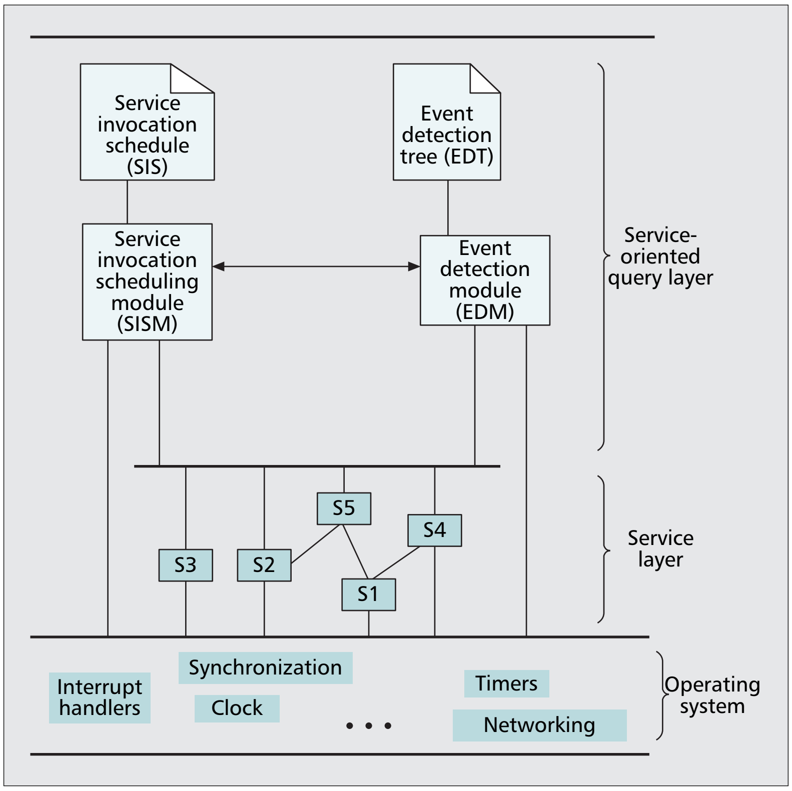
To understand the role of these modules, we describe how queries are handled by the SOQ layer. When the SOQ layer receives a query `Q` from the SDR layer, it first determines `Q`'s type (i.e., whether it is a task or an event query). It then processes `Q` as follows.

Processing task queries: Let `Q(s)` be the query the SOQ layer receives from the SDR layer requesting invocation of service `s`. The SOQ layer submits the query to the SISM to request the scheduling of the invocation of service `s`. If `Q` is a one-time query, the SISM invokes the service `s` and returns the results to the routing layer (Fig. 1). If `Q` is a recurrent query, the SISM adds a new entry in the service invocation schedule. It then activates a timer T_Q to trigger future invocations of `s`. Each time T_Q fires, the SISM invokes `s` and returns the results to the routing layer.

Processing event queries: Let `Q(e,s)` be an event query that requests the invocation of service `s` when event `e` occurs. The SOQ layer first submits query `Q` to the EDM. The EDM inserts the predicate corresponding to `e` in its event list. When the EDM detects the occurrence of `e`, it submits query `Q(s)` to the SISM. The SISM then processes this query as a regular task query (as previously explained).

Service Layer

The service layer is a collection of lightweight services. Each service is a software module that carries out some sensing, actuation, or control function. A service may interact directly with OS components (e.g., sensor controllers, timers) of its local node. These components interact with the node's hardware modules (e.g., actuation unit, clock).



Service invocation schedule (SIS)
Event detection tree (EDT)
Service invocation scheduling module (SISM)
Event detection module (EDM)
Service-oriented query layer
S5
S4
S3
S2
S1
Service layer
Synchronization
Interrupt handlers
Clock
Timers
• • •
Networking
Operating system

Figure 2. *Service-oriented query layer.*

Without loss of generality, we illustrate our discussion through sensing services that accept no parameters. We adopt the syntax `getAttribute()` to express invocations of sensing services. For example, an invocation of a sensing service that samples and returns temperature would be noted `getTemperature()`. Similarly, we consider actuation services that set the single value that corresponds to the new value to be assigned to a given parameter. We adopt the syntax `setAttribute(value)` to express invocations of actuation services. For example, `setLight(on)` and `setLight(off)` are invocations of an actuation service that set the attribute light on and off. We also assume that all services return a single scalar value.

Routing Layer

This layer is responsible for:
- Delivering incoming queries to the SOQ layer of the local node
- Sending out query results produced by the SOQ layer
- Forwarding received queries and query results to neighbors

The routing layer consists of two protocols: Service-Driven Routing Protocol (SDRP) and Trust-Aware Routing Protocol (TARP). SDRP routes queries from the base station to the nodes in the network, while TARP routes query results from the network's nodes to the base station. In this article we focus on SDRP and refer the reader to [9] for ample details on TARP.

Existing routing schemes for SANETs generally do not exploit the semantics of the queries. As a result, a query may be routed through several hops to end up at nodes that do not provide

The purpose of path learning is to let each node know whether it is on the path to nodes providing any given service. Path learning takes place at bootstrapping time. It also takes place when new nodes are deployed or when new services are deployed on existing nodes.

the sensing/actuation capability it requests and do not have paths to nodes that provide the requested capability. This obviously incurs excessive communication and processing overhead that is not strictly required to deliver the query to its recipients. Consider a node n that receives a query message m requesting some sensing/actuation capability c. In most current deployments of SANETs, n forwards m to (at least some of) its neighbors regardless of whether or not this gets the message closer to some node that provides c. In this scenario it is likely that m traverses a large number of hops only to reach *dead ends* (i.e., nodes that do not provide the capability requested by m). The key idea behind SDRP is to avoid *aimless routing* — forwarding query messages to nodes that neither provide the requested service nor are on the path to nodes that provide the requested service. Each node determines the capability requested by the received query and forwards a query *only* if it determines it is on a path to one or more nodes that provide the requested capability.

SDRP builds and maintains two data structures: a *service table* and a *service directory*. A node n_i's service table (ST_i) contains information about n_i's services. In particular, each entry in ST_i contains the service identity (`serviceID`) of each service n_i provides and the service class of that service (e.g., `getLight()`). The second data structure each node builds and maintains as part of the SDR protocol is the service directory. A node uses its service directory to store information about services provided by reachable nodes. An entry in SD_i corresponds to a service class (e.g., `getLight()`) for which the local node has determined that there is at least one reachable provider. The service directory may be thought of as the routing table of SDRP. Using their respective service directories, nodes cooperate to route query messages via paths with no dead ends.

The basic SDR protocol consists of two concurrent activities: *path learning* and *query routing*.

Path Learning — The purpose of this phase is to let each node know whether it is on the path to nodes providing any given service. Path learning takes place at bootstrapping. It also takes place when new nodes are deployed or new services are deployed on existing nodes. After the network is deployed, all nodes enter a service dissemination phase through which each node advertises its sensing and actuation services to its neighbors. Service dissemination is an incremental process in which nodes that become aware of new services further advertise this service in their neighborhood.

At bootstrapping, each node n_i broadcasts a `MyServices` message that contains the list of service classes it provides. When a node n_j receives this message, it iterates through each of the service classes included in the message. For each service class, n_j checks whether there is an entry for that class in SD_j. If not, n_j simply adds a new entry corresponding to that class to its service directory SD_j. In basic SDRP, n_j then broadcasts a message `ServiceUpdate` to its neighbors informing them that node n_i provides the given service. Each node n_p (other than n_i) that receives the message `ServiceUpdate` from n_j updates its service directory as follows: For each entry in the message `ServiceUpdate` that corresponds to a service not already in SD_p, n_p adds an entry for the service to its service directory SD_p. In a more efficient version currently in development, n_j waits until k new entries are added to its service directory before it broadcasts the message `ServiceUpdate`. k, the number of entries each `ServiceUpdate` message contains, is called the *update threshold*. Using this threshold, however, may prevent nodes from disseminating routing updates in a timely manner where adding k new entries to the service directory takes an excessively long time. To prevent this situation, we use a time limit on how long a node may wait before it sends routing updates. If this limit is reached while there is at least one new entry in the service directory, the node broadcasts a `ServiceUpdate` to its neighbors.

Query Routing — The second activity SDRP performs is routing queries. Query routing in SDRP is a distributed process in which several nodes cooperate in routing queries requesting services toward nodes providing those services. Typically, a query is initiated by a base station, which requests the invocation of a given service by a given subset of nodes. Let `Q(s)` be a query the base station issues requesting the invocation of service `s`. First, the base station broadcasts `Q` to its immediate neighbors. When node n_i receives `Q` from another node, n_j, SDRP first determines whether `Q` contains a `space` clause. If so, SDRP determines whether n_i is involved in the query (i.e. whether n_i is included in `Q`'s spatial scope). If so, SDRP looks up `s` in ST_i to check whether n_i provides the service requested by `Q(s)`. If `s` is in ST_i, SDRP simply passes the query to the local node's service-oriented query layer (Fig. 1). SDRP then looks up `s` in SD_i. If an entry is found, there is a path from n_i to one or more nodes that provide `s`. In this case SDRP forwards the query `Q(s)` to n_i's neighbors.

IMPLEMENTATION OF TINYSOA

We implemented the proposed approach in TinySOA, a prototype service-oriented query processing system built on top of TinyOS 1.1.15. Depending on the nature of the function to be provided, services in our implementation may be coded as one of three types of TinyOS processing units: asynchronous commands, synchronous commands, and tasks. These types derive directly from TinyOS's constraints. Services coded as asynchronous commands (using async) may be executed at any time (preempting other code). Asynchronous commands are therefore used for services that have time constraints on their invocation time and whose execution is of short duration. Synchronous commands do not preempt other code. They are used for services with less stringent constraints on their invocation time and whose execution may be of longer duration. Tasks in TinyOS are used to perform long processing, such as background data processing, and can be preempted by hardware event handlers. They are therefore used for services that are not very critical. Note, however, that in TinyOS, tasks may not take parameters and do not return results. Services coded as tasks are therefore used only when no input values are needed and no output is expected.

The service-driven routing protocol was implemented as a separate nesC module, called `SDRP.nc`, that may coexist (within a given application) with TinyOS's standard communication mechanisms. The purpose of providing this flexibility is to enable programmers to select the routing layer to use when routing a given class of queries. For example, for queries that must reach all or most of the nodes, TinyOS's default communication primitives would probably be more efficient.

EVALUATION OF TINYSOA

In this section we present an evaluation of TinySOA. We first study the scalability of SDRP, TinySOA's query routing mechanism, in terms of energy consumption. We then conduct comparative experiments between TinySOA and TinyDB [8]. We also compare TinySOA to some existing systems where the concept of service is adopted.

ENERGY CONSUMPTION AND SCALABILITY

To evaluate energy consumption in TinySOA, we developed an evaluation benchmark that uses the PowerTOSSIM simulator [12] integrated in the TinyOS package. The benchmark enables a wide spectrum of simulation scenarios. In particular, users may specify configuration parameters such as the total number of different services, maximum number of services per node, and update threshold. The benchmark also enables users to specify executions where any number of queries is injected in the network, any values for the parameters of the queries may be selected, and the time between the injection of two consecutive queries may be varied.

Scalability in the Number of Nodes — In the first experiment we considered SDRP's scalability with regard to the number of nodes. Specifically, we measured the energy required to set up SDRP as reflected in the average energy consumed by the nodes' radios and CPUs. We ran experiments on networks with a number of nodes varying from four to 961 nodes. We assumed a lossy radio model and used TinyOS's LossyBuilder tool to generate the probabilities of incorrect bit reception for each considered network topology. For example, in the first iteration we generated a probability file for a grid of 2 × 2 nodes in an area of 6 ft × 6 ft. In the last iteration we generated a probability file for a grid of 31 × 31 nodes in an area of 93 ft × 93 ft. Note that we kept the same node density (1/9 node/ft^2) for all iterations. Each node runs a number of services selected randomly between 1 and NBS_{max}. NBS_{max} was kept constant at 5 in this experiment. The identities of the services running on each node were also selected randomly from a set of `NbServices` (kept at 50) services. The experiment ends when no message remains in transit; that is, when all messages sent as part of SDRP are either received or lost. Figure 3a shows the results of the experiment.

The figure shows that energy consumption increases almost linearly until we reach about 30 nodes. From that point until the number of nodes reaches 700, energy consumption remains constant. At that point increases slightly and then remains constant. The key conclusion from this experiment is that for topologies of a given size (30 nodes or more under this experiment's conditions), energy consumption in SDRP becomes almost independent of the number of nodes. Indeed, SDRP depends only on the number and distribution of services on the nodes. This makes SDRP particularly suitable for large networks.

Scalability in the Number of Queries — In the second experiment we kept the number of nodes constant at 100 nodes and measured TinySOA's energy consumption when the number of queries injected in the network varies from 10 to 150. The other parameters were similar to the ones used in the first experiment. The base station submits a query which requires that each node providing the service `getLight()` invoke it and then send the result to the base station.[3] Here again, the experiment ends when no message remains in transit. We measured the average energy consumed by the nodes' radios and CPUs from the time the base station submitted the query until the end of the experiment.

The experiment shows two important results. First, the setup cost in terms of energy (i.e., running SDRP before queries may be routed) is very low. For example, Fig. 3a shows that at 100 nodes, SDRP's setup requires about 330 mJ. Figure 3b shows that the total cost of executing 60 queries or more is slightly less than 1500 mJ. This makes SDRP's setup cost only about 22 percent of the total cost of executing 60 or more queries. A more important result is that beyond a certain number of queries (60 in this case), the cost of query processing in SDRP increases very slowly with the number of queries. This is due to the key characteristic (i.e., service-driven routing) of SDRP, which reduces a query's scope in terms of the number of nodes that receive the query. This eliminates much of the unnecessary traffic that would normally occur in traditional routing protocols.

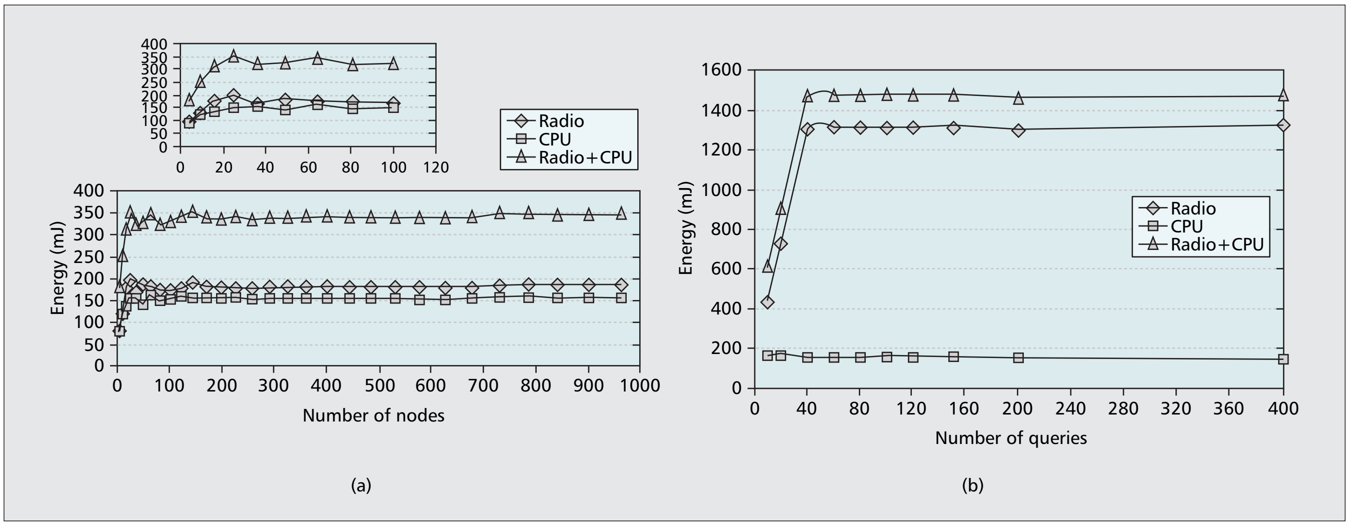
TINYSOA VS. TINYDB

In this section we compare TinySOA to TinyDB. We focus on three criteria:
• Energy consumption
• Scalability
• Response time

To conduct our comparison, we considered that querying for the values of attributes in TinyDB is the equivalent of invoking simple services in TinySOA. However, mapping TinySOA's services into TinyDB's attributes is not straightforward. TinyDB has a number of specificities and limitations that had to be considered. In TinySOA arbitrary services may be deployed on nodes. In TinyDB, however, this is not possible. Indeed, TinyDB uses a static set of attributes (temperature, light etc.). This set is specified in a file called `catalog.xml` that is loaded at bootstrapping. TinyDB then makes *all* of the

The benchmark enables a wide spectrum of simulation scenarios. In particular, users may specify configuration parameters such as the total number of different services, the maximum number of services per node, the update threshold, etc.

[3] *As we said earlier, we focus here only on the routing of queries, not query results.*



■ **Figure 3.** *Energy consumption and scalability of SDRP: a) varying the number of nodes; b) varying the number of queries.*

attributes defined in the catalog available, and hence queryable, on *all* nodes. The second constraint was that in TinyDB, populating nodes with different sets of attributes may not be done programmatically. TinyDB provides a mechanism to add an attribute for a specific mote on the fly (i.e., after TinyDB starts). This mechanism, however, is only available through TinyDB's Java-based GUI. Moreover, this is only possible for constant attributes. To reflect a similar setting, we set the number of services available on each node in our TinySOA network to 19, which is the number of attributes in TinyDB's default catalog.

We considered a scenario where the base station submits a single query and waits until it receives a given number of results. In TinyDB we wrote a Java class that interacts directly with the TinyDBApp nesC application and injects the following query into the network:

```
select nodeid,light
from sensors
sample period 30000
```

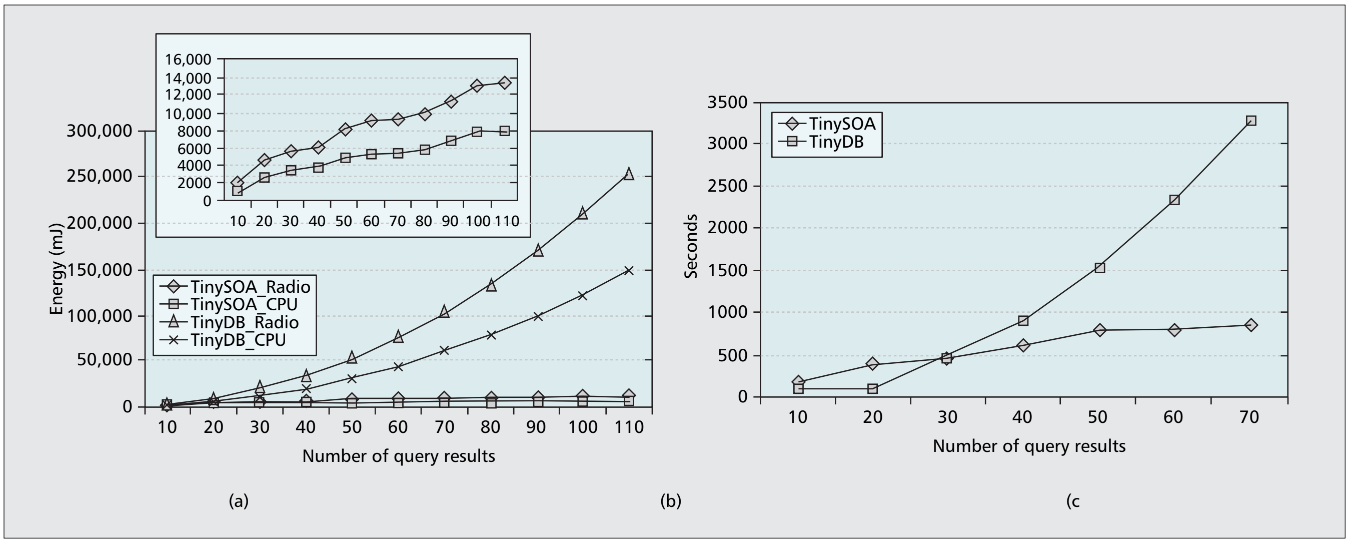
The equivalent TinySOA query is

```
action getNodeID(), getLight();
time frequency 30000 ms;
```

In both cases the experiment ended when the base station received a certain number *qr* of query results that was varied from 10 to 110.

Energy Consumption — Figure 4a shows the average energy consumed by the nodes' radios and CPUs with TinySOA and TinyDB. The figure shows that in the case of TinyDB, energy consumption increases almost exponentially with the number of query results received at the base station. Figure 4b is a closer view that shows energy consumption in the case of TinySOA. Energy consumption in this case increases almost linearly with the number of results the network generates. The difference between TinySOA and TinyDB in terms of energy consumption may be explained by the fact that TinySOA cuts significantly the number of messages required to get a query to all nodes that must contribute to its evaluation. The length of the paths between the base station and the target nodes depends almost entirely on the distribution of services in the network and not much on the size of the network. If only a few nodes of the network provide the service requested by a given query, only a proportional number of nodes are involved in routing the query to those nodes, regardless of how many other nodes exist in the network. This is not the case in TinyDB, where the number of nodes that contribute to routing a given query increases systematically with the network's size. As a result of the difference between TinyOS's query routing and TinyDB's query routing, and given the size of the network (100 nodes), queries get to their destination a lot sooner in TinySOA than in TinyDB. As it takes longer for TinyDB to get a query to its destination, some nodes start generating results and sending them to the base station while some other nodes have not even received the query. This seems to create a significant amount of opposite traffic. This traffic translates into a large number of collisions, which in turn results in many failed transmissions and retransmissions. This, in fact, is confirmed in the next experiment.

Response Time — We also conducted experiments to compare TinySOA and TinyDB from the perspective of response time. The purpose was to measure the time taken by both networks to route query results from the nodes where they are generated to the base station. This metric is important when assessing the suitability of both networks to support real-time applications. When conducting this experiment in TinySOA, we made each node that generates a result for a query simply broadcast it to its neighbors. Each neighbor that receives a query result forwards it until the query result reaches the base station. We set the number of nodes at 100 nodes, and measured energy consumption of nodes' radios and CPUs until the base station received a given number of results in both TinySOA and TinyDB. Figure 4c shows this experiment's results. Here



■ **Figure 4.** *Comparison between TinySOA and TinyDB: a) energy consumption, TinySOA vs. TinyDB; b) energy consumption in TinySOA; c) query processing time, TinySOA vs. TinyDB.*

again, we can see that in TinyDB this time increases exponentially. In contrast, in TinySOA the increase is linear. While using simple broadcasting to route query results to the base station is certainly less efficient than in the case of TinyDB, TinySOA outperforms TinyDB mainly because of the savings in time TinySOA achieves when routing queries from the base station to the nodes using SDRP.

Ad Hoc Network Deployment —A TinyDB network must be completely defined prior to deployment. In particular, the attribute catalog must be defined before the network is deployed. In addition, all nodes must run TinyDB's code. This makes TinyDB unsuitable for the next generation of sensor systems where different networks from different providers run different operating systems. In contrast, in TinySOA services are dynamically discovered and used. In addition, when new services are discovered, SDRP is able to automatically update service directories without any side effects on the normal operation of the network.

On-node services: Almost all existing *service-oriented* architectures for sensor systems introduce services as off-network programs running on computers, not sensor nodes. Examples include: [3, 6, 7, 11, 13]. TinySOA deploys on-node services that may be advertised, discovered, and invoked by entities within or outside the network. This makes TinySOA networks open environments that may readily interoperate with each other as well as with other types of client entities. For example, consider a user roaming an area where different TinySOA networks are deployed. This user may be able, using a *generic* mobile device such as a laptop or cell phone, to discover and invoke services provided by nodes that belong to different networks. Interaction would be possible without any prior configuration of the mobile device to query any given TinySOA network.

Service-based optimization: Both in-network and off-network optimization have already been proposed in a number of existing service-oriented sensor systems. Examples include [6, 10, 13] for systems capable of off-network optimization and [5] for systems capable of in-network optimization. These systems, however, do not exploit services in optimization; they simply provide traditional application-level forms of in-network optimization. None of these systems have considered the idea of exploiting services in low-level mechanisms such as routing. In TinySOA services are both a design paradigm and an optimization means. They enable new forms of in-network and off-network optimization:

- In-network service-based optimization: TinySOA enables in-network service-based optimization at the application level (e.g., multiquery optimization, result aggregation) as well as at lower levels. An example of the former is TinySOA's ability to associate a single service invocation with several queries. An example of the latter is TinySOA's SDRP discussed earlier.
- Off-network service-based optimization: From the perspective of TinySOA's clients (e.g., base station, mobile user), TinySOA streams query results in ways similar to existing sensor systems. Traditional off-network optimization techniques (caching, query rewriting, reprocessing previous results to answer new queries, etc.) are also applicable in the context of TinySOA.

Application independence and application awareness: Application awareness refers to the ability to exploit the specific characteristics of a given application to improve the overall efficiency of the network. Several prior efforts have proposed application-aware solutions. However, these solutions are often too specific to the considered class of applications. In contrast, TinySOA exposes "neutral" services any application may use with the same efficiency expectations. TinySOA is therefore application-independent while still able to exploit any specific characteristics of a given application.

While existing literature has already explored some aspects of service-oriented design in sensor networks, our work is a fundamentally novel

Future SANETs will require new architectures. We anticipate that taking the service-oriented design paradigm to wireless networks will likely extend the immense impact that the concept of services has had on wired computing.

approach where services are selectively deployed on top of the bare operating system controlling sensor-actuator nodes. Another major difference is that services in our approach are not only a means for better expressivity but, more important, a key element in query optimization.

Conclusion

We have motivated the need for and proposed a new service-oriented architecture for sensor-actuator networks. In contrast with current SANETs, SOSANETs expose their sensing and actuation capabilities in the form of *services* that may be invoked by *any* application. We show the potential of SOSANETs in addressing the limitations of current SANET architectures. We have implemented our approach in TinySOA, a SOSANET developed on top of TinyOS. Empirical results show that TinySOA achieves significant improvements over existing architectures in many aspects, including energy consumption, scalability, and response time.

Future SANETs will require new architectures. We foresee service-oriented architectures as a highly viable candidate to support the requirements of tomorrow's sensor-actuator networks. We anticipate that taking the service-oriented design paradigm to wireless networks is likely to extend the immense impact the concept of services has had on wired computing.

Acknowledgment

This work is sponsored in part by NSF award 0721523.

References

[1] D. J. Abadi, S. Madden, and W. Lindner, "REED: Robust, Efficient Filtering and Event Detection in Sensor Networks," *Proc. 31st Int'l. Conf. Very Large Databases*, Trondheim, Norway, Aug. 30–Sept. 2, 2005, pp. 769–80.
[2] I. F. Akyildiz and I. H. Kasimoglu, "Wireless Sensor and Actor Networks: Research Challenges," *Ad Hoc Networks*, vol. 2, no. 4, 2004, pp. 351–67.
[3] F. Coimbra Delicato *et al.*, "A Service Approach for Architecting Application Independent Wireless Sensor Networks," *Cluster Comp.*, vol. 8, no. 2–3, 2005, pp. 211–21.
[4] A. J. Demers *et al.*, "The Cougar Project: a Work-in-Progress Report," *SIGMOD Rec.*, vol. 32, no. 4, 2003, pp. 53–59.
[5] F. Golatowski *et al.*, "Service-Oriented Software Architecture for Sensor Networks," *Proc. Int'l. Wksp. Mobile Comp.*, 2003, pp. 93–98.
[6] J. Liu and F. Zhao, "Towards Semantic Services for Sensor-Rich Information Systems," *Proc. 2nd IEEE/CreateNet Int'l. Wksp. Broadband Advanced Sensor Networks*, Oct. 3 2005.
[7] K. A. Hua, R. Peng, and G. L. Hamza-Lup, "WISE: A Web-based Intelligent Sensor Explorer Framework for Publishing, Browsing, and Analyzing Sensor Data over the Internet," *Proc. 4th Int'l. Conf. Web Eng.)*, July 2004, pp. 568–72.
[8] S. Madden *et al.*, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Sys.*, vol. 30, no. 1, 2005, pp. 122–73.
[9] A. Rezgui and M. Eltoweissy, "TARP: A Trust-Aware Routing Protocol for Sensor-Actuator Networks," *4th IEEE Int'l. Conf. Mobile Ad Hoc and Sensor Networks*, Pisa, Italy, Oct. 2007.
[10] P. Schramm *et al.*, "A Service Gateway for Networked Sensor Systems," *IEEE Pervasive Comp.*, vol. 3, no. 1, Jan.–Mar. 2004, pp. 66–74.
[11] M. Sgroi *et al.* "A Service-Based Universal Application Interface for Ad Hoc Wireless Sensor and Actuator Networks," *Ambient intelligence*, W. Weber, J. Rabaey, and E. Aarts, Eds., Springer Verlag, 2005.
[12] V. Shnayder *et al.*, "Simulating the Power Consumption of Large-Scale Sensor Network Applications," *SenSys*, J. A. Stankovic, A. Arora, and R. Govindan, Eds., ACM, 2004, pp. 188–200.
[13] L. Zhuang *et al.*, "Power-aware Service-oriented Architecture for Wireless Sensor Networks," *Proc. 31st Annual Conf. IEEE Industrial Elect. Soc.*, 2005.

Biographies

Abdelmounaam Rezgui [M] (rezgui@vt.edu) is a Ph.D. candidate in the Department of Computer Science at Virginia Tech. He received his M.S. in computer science from Purdue University. His research interests include service-oriented computing, routing, and query optimization in sensor networks, and reputation and trust. He has written more than 30 book chapters, journal papers, and conference papers. He is a member of Upsilon Pi Epsilon.

Mohamed Eltoweissy [SM '05] (eltoweissy@vt.edu) is an associate professor of elecrical and computer engineering at Virginia Tech. He also holds a courtesy appointment in computer science. His research is in ubiquitous networking, sensor and ad hoc networks, information assurance and trust, and network cognomics and autonomics. His funded research exceeds $10 million. He is a senior member of ACM.