A Network Application Programming Interface for Data Processing in Sensor Networks

Rice University Technical Report TREE0705

Raymond Wagner, J. Ryan Stinnett,
Marco Duarte, Richard Baraniuk
Dept. of Electrical and Computer Engineering
Rice University

David B. Johnson,
T. S. Eugene Ng
Dept. of Computer Science
Rice University

ABSTRACT

Since the inception of sensor networks, a wide variety of algorithms for in-network data processing have emerged. To enable practical implementation of a broad class of these proposed algorithms, sufficient application programming support for network communications is critical. While standard (but relatively low-level) network application programming interfaces (APIs) such as those implemented in TinyOS can provide a basis for flexible application-specific customization, experience in the past several years has shown that data processing algorithms in fact share similar higher level communication needs that can be better supported.

In this paper, we first identify common communication patterns through an extensive survey of data processing algorithms proposed over the past four years in the proceedings of the Information Processing in Sensor Networks (IPSN) conference. We then present the design of a higher level network API for sensor networks that is powerful, convenient to use, and compact. Many familiar issues in traditional networking such as addressing and reliability have vastly different solutions in the sensor network environment, and we carefully develop the rationales behind the design of the resulting network API, providing an in-depth discussion on the fundamental design decisions. We believe that the proposed network API serves as a starting point for the implementation of a more comprehensive sensor network communication middleware than what is found in currently available systems, enabling algorithm developers to quickly and easily implement their designs without having to handle low-level networking details themselves.

1. INTRODUCTION

Sensor networks pose an incredible variety of research challenges to the community that is working to make such systems a reality. During the relatively few years in which sensors networks have been actively studied, a great deal of progress has been made toward designing sensor node hardware, creating data processing applications to run on nodes, and developing the programming and networking interfaces to unite applications with hardware. There is, however, a growing disconnect between the latter two endeavors that must be addressed to sustain the growth of sensor networks into a mature technology.

It is widely recognized that the sensor network model, which consists of a wireless ad-hoc network of sensor nodes connected to one or more sink nodes (that are perhaps attached to the Internet), features a unique energy economics which encourages the development of applications with distributed components. That is, application designers often prefer localized collaboration among nodes and in-network processing of data to long-haul transport of data to a central location for conventional processing. And while an abundance of creative distributed algorithms have been proposed, a great deal still occupy the realm of theory. The in-network collaborations require by these algorithms induce a variety of network communication patterns, many of which are not supported by standard sensor network programming tools such as TinyOS [32]. And since many algorithm designers do not have the backgrounds necessary to develop the network communication protocols they need using TinyOS building blocks, many of their designs go untested in real sensor network environments. Thus, many algorithm developers are unable to ascertain the true practicality of their designs and identify and correct possible flaws exposed by the complex operating environments of real-world deployments.

What these researchers lack is a network programming abstraction to allow them to implement their designs without concerning themselves with the underlying network services to support the communications they require.

We believe the time is right to design a network application programming interface (API) that provides access to a suitable set of abstract network services for data processing in sensor networks. An API is *not* a specific implementation of these abstract services. Rather, an API provides a portal through which an application accesses the abstract services. Defining the API is a crucial intellectual exercise, since the API is the part of the system that is the most difficult to change over time; any change to the API requires application programs to be re-written. In contrast, the implementation of an abstract service is relatively changeable. For example, the implementation of a multi-hop datagram service can change from using link state routing to distance vector routing transparently.

Designing a suitable network API requires solid knowledge of the applications' communication needs. Fortunately, the plethora of sensor network applications and algorithms proposed to date provide strong guidance. Moreover, ad hoc wireless network algorithms and protocols have matured in recent years, so the basic technologies necessary for implementing the abstract services are available. Starting with these observations, this paper addresses the problem of designing a suitable network API for sensor networking. Our primary goals are to expose the technical trade-offs between different API design choices and to propose a specific API that is sufficiently powerful, convenient to use, compact, and realizable with existing technologies. We intentionally leave out implementation issues such as software architecture or system performance evaluation. We expect future work to explore and evaluate different implementation choices for realizing the proposed API.

Our contributions in this paper are two-fold. First, we conduct a survey of the distributed data processing algorithms proposed in the proceedings of Information Processing in Sensor Networks (IPSN) to extract a family of key communication patterns that recur across the algorithms. In Section 2 we present the results of this survey. Second, we design a network API to cover the classes of communication. We begin by carefully discussing the design of the API in Section 3, and we present the API calls and a brief discussion of possible implementation directions for each in Section 4. We provide a detailed treatment of four of the surveyed algorithms in Section 5, indicating the API calls necessary to implement the communications required by each. Finally, we conclude the paper in Section 6.

2. SURVEY OF APPLICATION REQUIREMENTS

To understand the communication requirements of typical sensor network applications, we conducted an extensive review of the papers proposing data processing algorithms in the proceedings of IPSN to date. We surveyed over 100 papers in all, but due to space limitations, we present the survey results for a set of 30 of them here, chosen to best represent the diversity of application classes and communication patterns. For the complete survey results, we refer the reader to [33]. In categorizing each paper, we carefully looked to the authors' description of the assumed networking environment, making as few assumptions as possible on our own part in order to accurately capture the authors' original intent.

The proposed applications span a wide variety of topics. Algorithms for data analysis include standard signal processing applications such as measurement compression [26, 34, 37], target tracking [3, 8, 23, 27, 35], and parameter estimation [25, 28, 38], as well as those more specific to sensor networks, such as query servicing [9, 13, 14, 19, 36] and aggregation [5, 10]. Network maintenance algorithms form another common application class, with protocols to enable node self-localization [16, 29], guide node placement, [18], schedule node sleep cycles while maintaining sensing coverage [1, 4, 39], detect network faults [30], and provide navigation assistance for mobile agents traversing a sensor network [2]. Finally, a number of papers extend their algorithm proposals to real-world implementations, that monitor environmental and structural phenomena [6, 11, 15, 31].

Despite great diversity in the kinds of applications, we find significant commonality in their node communication requirements. Some notion of address-based sending, either to a single destination or a set of destinations, is found in a majority of the proposed algorithms [2, 3, 5, 6, 8–10, 13, 14, 16, 19, 23, 25, 26, 28, 30, 31, 34, 36, 37, 39]. These addresses typically take the form of unique node identifiers, though a subset of applications address multicast groups to which nodes may subscribe [14, 23].

Region-based sending also emerges as a common feature. Broadcast of a message is common, either to all of a node's immediate neighbors within wireless transmission range [2–5, 10, 16, 18, 19, 29, 31, 34, 36, 38] or to all neighbors within a larger number of radio hops [2, 14, 29]. This notion is extended in several applications to sending a message to all nodes within a geographic radius of the sender [1, 4, 14, 34]. Finally, a number of applications wish to send a message to nodes within an arbitrary region of space not centered around the sender [13, 14, 23, 27, 37].

Communication based on hierarchies of more- and less-powerful devices is also common. In its simplest form, this consists of sending to one or more central data sinks in an otherwise homogeneous network of nodes [4, 14, 18, 19, 26, 30, 31, 34, 37]. For networks with multiple device classes below the level of the sink, this notion is extended to sending to a more powerful parent device and less powerful child devices [8, 11, 15, 35].

We note that most applications at least implicitly require some form of multi-hop communication, with only a handful specifically relying solely on single-hop transmissions [5, 10, 25, 28, 38]. Similarly, most applications imply at least some level of reliability in packet delivery, with a very few claiming complete robustness to unreliable links [5, 10, 25, 38].

Finally, while most applications concern themselves with reception of packets only at the intended destination, a few leverage the ability of nodes to eavesdrop on packets passing through their vicinity on the way to a different destination [14, 25, 35].

3. DESIGN DECISIONS

The survey results provide a starting point for specifying an API to cover common communication patterns; a number of issues, however, influence the shape of the final API. We motivate and detail key design decisions in this section, beginning with a brief overview of the state of the art in sensor network programming APIs.

3.1 Preliminaries

Over the past several years, a variety of programming models and support frameworks have been developed for sensor networks to overcome the unique challenges faced when trying to implement data processing applications of varying complexities on highly constrained hardware platforms. Of the available architectures, TinyOS has garnered the most research and attention. Its event-driven model is attractive because it allows for a direct translation of hardware interrupts from physical devices into handlers that run in response to these interrupts.

TinyOS also allows developers to program in nesC, an event-based extension of C created specifically for TinyOS, rather than forcing users to adapt to an entirely new language. TinyOS 2.0 contains a large selection of components to simplify typical tasks an application might want to perform, such as collecting sensor readings, managing overall device power, and communicating with other nearby devices [22].

While TinyOS has greatly reduced the work required of the application developer, its networking components are focused on supporting two basic communication types: single-hop unicast to a single node and single-hop broadcast to all nodes within radio range of the sender [21]. This provides a basis upon which more complicated transmission schemes can be built. For example, TinyOS 2.0 extends the single-hop support to include a tree collection protocol [12]. The communication classes directly supported by TinyOS, however, do not cover the bulk of the patterns enumerated in Section 2.

Moreover, while TinyOS's basic send and receive system is well designed for simple networking systems, it lacks several features useful to applications developers. For example, to expend extra effort sending a certain packet in order to increase its chances of reaching the receiver, a developer may take a number of platform-dependent actions, such as enabling automatic radio acknowledgements (ACKs) or increasing the radio's transmission power. There is, however, currently no transparent way for an application to specify increased reliability. Developers must directly access these low level controls, which can differ from one platform to another. As another example, to send data larger than the packet structure's fixed payload length, an application designer must custom-build a fragmentation and reassembly scheme. Implementing such a system can itself become quite complex and distracting for designers wishing to focus their efforts on novel applications.

3.2 Supporting Multiple Addressing Modes

In addition to the most basic form of unicast addressing for identifying the recipient of a message, the proposed API must also support a multicast group addressing mode as well as addressing modes based on physical regions, either centered around the sending node or around an arbitrary point in space. Efficiently supporting these families of send functions in the proposed API requires the following decisions:

Provide a dedicated API call for each addressing mode. Instead of having a generic *send*() API call and using a parameter to specify the intended addressing mode, we dedicate a separate API call to each addressing mode. This design allows the underlying execution environment, such as TinyOS, to selectively load only the code corresponding to the addressing mode(s) required by the application onto the possibly resource-constrained nodes. This design also removes the need to unnaturally force very different addressing modes such as unicast and region-based addressing to fit into a single API call mold.

Use a separate address space for multicast addressing. In today's IP network, unicast and multicast addresses co-exist within a 32-bit address space, where multicast addresses are identified by the prefix bit sequence 1110. While a similar strategy may be used for a sensor network API, sensor node addresses are typically drawn from a 16-bit address space [20]. To statically reserve a significant portion of those 2^{16} addresses for multicast addressing may be wasteful. Instead, the proposed API uses a separate 16 bit address space for multicast addressing. An address is evaluated as either unicast or multicast depending on the API call naming it as a destination.

3.3 Supporting Multiple Receive Modes

To enable an application running on a node to eavesdrop on passing messages for which the node is not the intended recipient, the proposed API supports overhearing receive modes in addition to the basic mode where the receiving node is the intended destination. The multi-hop nature of network traffic enables two very different kinds of overhearing receivers. The first merely observes traffic in the node's vicinity, while the second allows a node to intervene and modify passing message. Support for these two modes is qualified as follows:

Support an eavesdropping receive mode. A node is allowed to passively listen to all overheard messages for which the node is not the ultimate destination. This includes messages sent by neighbors for which the node is not the next-hop destination as well as those for which it is the next-hop destination. This is the multi-hop analogue to TinyOS's radio-channel eavesdropping.

Support an intervening receive mode. A node is notified of messages it is forwarding to another multi-hop destination and allowed to modify those messages before performing the forward (including optionally canceling the forward). This new capability can allow for novel implementations of distributed algorithms. Consider, for example, the algorithms proposed in [19, 26], both of which perform cluster-based aggregation of data where the node serving as the cluster head can change with time. Both algorithms expend effort to build and maintain a routing tree rooted at the head node, and child data is aggregated by parents in the tree as it flows to the root. Instead, each node in the cluster could address the cluster head directly, with all nodes utilizing the intervening-receive mode. Nodes would wait an amount of time inversely proportional to their hop-count to the head before aggregating all intercepted data with their own reading and addressing the result to the cluster head. Such a solution avoids re-building the routing tree when the cluster head membership changes or detecting and repairing links in the tree that have been broken due to changes in the wireless communication environment.

3.4 Giving the Application the Ability to Control Transmission Effort

Transmission effort is an important issue to consider in designing a network API for sensor networks as it impacts both the reliability of packet delivery and the amount of energy consumed in delivering a packet.

Among existing network APIs that provide enhanced transmission effort, the TCP socket API is the best-known. It provides an end-to-end reliable in-order byte-stream delivery service to the application, and data is retransmitted until an end-to-end acknowledgement is received. And while the TCP socket API serves the common case in the communication and content oriented Internet, it does not serve the common case in data-processing oriented sensor networks, where data tend to be much more time-sensitive — consider, for example, the sensor measurements that drive the target tracking applications proposed in [3, 8, 23, 27, 35]. A TCP socket-like API may retransmit a packet that the application no longer considers useful, wasting energy. Moreover, the in-order nature of the service causes the application to lose control over the timing of packet transmissions. When a packet is retransmitted past its deadline, the subsequent packet may be delayed sufficiently to miss its deadline as well.

Providing no transmission effort enhancement, however, does not sufficiently serve sensor networks — the chance of a packet being delivered successfully over a multi-hop wireless sensor network can, in some cases, become too small to be practical. These considerations led us to the following design decisions:

Give control to applications. The proposed API is designed to give applications control over the level of transmission effort required. Wireless sensor networks' reliability characteristics will depend on the physical environments in which they are deployed. Moreover, sensor network applications can have widely different tolerance to packet loss and sensitivity to timeliness of data delivery. Thus, only applications themselves can decide the level of transmission effort sufficient to achieve the desired performance.

Allow per-packet control. The proposed API allows the transmission effort to be controlled on a per packet basis. The API provides a datagram style service rather than a byte-stream style service like TCP. This allows applications to maintain finer control over the timing of packet transmissions. Packets of different importance to an application can be transmitted at different effort levels to realize different levels of reliability. The application can also adapt the transmission effort level at run-time to find the operational sweet-spot.

Provide an energy-based abstraction. The proposed API expresses transmission effort abstractly in terms of an energy factor relative to the amount of energy required for regular transmission. Thus, an energy factor of 1 implies no extra effort is needed, while a factor of 2 allows for up to twice as much energy to be used for transmission of the packet. Compared to an alternative where the transmission effort is expressed in terms of lower level notions, such as allowed number of re-transmissions, the energy-based abstraction has a number of advantages. First of all, applications on resource-constrained nodes can relate to energy consumption most meaningfully — with the energy-based abstraction, the application can pick a desired balance between reliability and energy consumption. Secondly, using energy as an abstraction allows a variety of techniques such as increased transmission power, decreased transmission rate, or acknowledgements and retransmissions, to be used transparently by the lower layer software and hardware depending on the situation.

Allow applications to manage congestion. Increasing transmission effort may potentially exacerbate congestion in the network; the proposed API explicitly leaves the management of network congestion to the applications. Sensor network applications should be designed correctly to avoid overloading the network. When necessary, an application can use a variety of available techniques such as rate-based and credit-based flow control [17] to help avoid transient congestion.

3.5 Providing Packet Fragmentation and Reassembly Service

Operational experience from the IP Internet suggests that packet fragmentation is a liability on network performance and software complexity that should be avoided when possible. However, sensor network radios in general allow only very small data payloads in packets. For instance, Chipcon's CC2420 radio currently supports a raw data size of 128 bytes [7], and TinyOS 2.0 uses a message structure with a default payload data size of just 28 bytes [20]. It is easy to find applications that send data units larger than such a small size.

Consider, for example, the Fractional Cascading query servicing algorithm [13], which returns the identifiers of all sensors in a region whose measurements fall in a given range — a set whose size can be arbitrarily large and require the payload of multiple packets. Consider also the distributed wavelet compression algorithm of [34], where transform data describing a node's roles at each scale of a multiscale transform must be sent by the sink before the transform can begin. The number of transform scales, and hence the size of the transform data, depends on the number of nodes in the network and cannot be guaranteed to fit in a single packet.

Our proposed API, therefore, provides a message fragmentation and reassembly service to reduce the burden on application programmers. Note, however, that applications should avoid packet fragmentation as much as possible.

The message fragmentation support of our API reflects the following design decisions:

Require hop-by-hop fragment reassembly. As mentioned in Section 2, some applications leverage packet eavesdropping [14, 25, 35], and others will likely benefit from intervening and modifying in-transit packets. To support such applications, fragment reassembly must be performed at each intermediate hop.

Require in-order fragment delivery at each hop. The primary implementation complexity for fragmentation lies in the reassembly of packet fragments when they may arrive out of order. The lower layer software should therefore provide an in-order fragment delivery service. A simple stop-and-wait protocol [17] is ideal for hop-by-hop in-order fragment transmissions. Although in many cases, a stop-and-wait protocol would sacrifice performance when compared to a sliding-window protocol (e.g., as used in TCP), this is not the case as used here, since there is no opportunity for pipelining of packets over the single wireless link before fragment reassembly at each hop.

3.6 Providing Flexible Memory Allocation and Management for Variable Sized Data

Both the application and the implementation of our sensor network API will in general need to deal with data objects for which the total size is not known in advance. For example, the message fragmentation and reassembly service described in Section 3.5 requires a node to collect the data from a variable number of packets (fragments) to reassemble the original application-level message; only once completely reassembled can the API pass the message to the application for processing. As another example, in a sensor network with nodes organized into hierarchical levels (Section 3.7), the application on some node may need a list of its immediate children nodes that are one level below it in the hierarchy; the number of such children included in the list may be a dynamic function of the network topology for which it may be difficult in advance to know the expected list size.

Our proposed API, therefore, provides a flexible memory allocation and management mechanism for such variable sized data. This mechanism entails the following design decisions:

Define a buffer chain data structure. Variable sized data are stored in a buffer chain data structure. A buffer chain is a linked list of memory chunks. Each memory chunk contains application data as well as meta data that facilitate the manipulation of the buffer chain. Multiple buffer chains containing different application messages can also be linked together to form a message queue. The idea is similar to the FreeBSD `mbuf` chain data structure [24]. The primary advantage of this design is that memory management is greatly simplified, since a pool of fix sized buffers can be pre-allocated by the system and used dynamically to store variable sized application data without heavy weight memory allocation and de-allocation operations.

Provide a buffer allocation service. The application can request fix sized buffers from the pool of buffers maintained by the system to create a buffer chain for storing its variable sized data. When a buffer is no longer needed by the application, it is returned to the pool for future use.

3.7 Supporting Self-Organized Device Hierarchies

The proposed API is designed to enable self-organization of the sensor network nodes into network hierarchies. Providing such network hierarchies is an important service since many applications rely on more powerful devices managing the data from less powerful devices — see, for example the target tracking applications of [8,35] and the environmental monitoring applications of [11,15].

Nodes in the network can be heterogeneous in many dimensions. Physically, nodes can have batteries with varying capacities or they may even be connected to a power grid. They can have varying computation capabilities, data storage resources, and communication bandwidth. In addition, nodes can also have different logical roles in a sensor network. For example, a sink node has the special logical role of a gateway between the sensor network and the outside world and is usually placed at the root of a network hierarchy. The proposed API is based on the following design decisions:

Do not assign static roles to device classes. We have explicitly decided against statically mapping different classes of devices (e.g. Stargates and Micas) to different fixed roles in a hierarchy, since the application should dictate how these resources are used. Suppose, for example, that some Mica nodes are connected to an external power source. With the proposed API, the application will have the flexibility of giving them a special role in the hierarchy to take advantage of their additional resources.

Support hierarchy level-based abstraction. The proposed API provides applications with the means to assign a node a hierarchy *level* number at run time. For example, the application can choose specific nodes to be the sinks and set them at level 1, set less powerful Stargate devices at level 2, and finally set least powerful Mica devices at level 3. The lower layer software then organizes the nodes to form efficient logical network hierarchies rooted at the level 1 nodes. Each node at level K is associated with a parent node at level $K-1$ whenever possible. Each node is also associated with one of the nodes at level 1 to enable it to send messages directly to the data sink. This approach allows applications to make very flexible decisions based on both the physical characteristics and the logical roles of network nodes.

Maintain self-organized logical overlays. It is important to note that the hierarchies constructed are logical overlays, so that a parent and its children need not be within physical radio range. This gives the lower layer software the flexibility to optimize the overlay structures to achieve good performance.

4. API DESCRIPTION

We present in this section the definition of our API for data processing algorithms in sensor networks, from the point of view of application developers who may use this API, and we also provide suggestions on how the underlying network protocols to support this API can be implemented efficiently in a real system. Our presentation here is divided between the API calls for sending messages and for receiving messages, and for send calls, it is divided between the calls for our three different addressing modes: address-based sending, geographic region-based sending, and sending relative to a device hierarchy. For each proposed API call, we also provide citations to representative example data processing algorithms that can directly use the call, based on our survey of application requirements presented in Section 2.

4.1 Address-Based Sending

There are three principle destination types for address-based sending. The first sends a message to a single node address. The second sends a message to a single address that is a multicast address to which several nodes may be subscribed. The third sends a message to each of a list of node addresses. The calls for each of these send types are specified as follows:

sendSingle(*data*, *address*, *effort*, *hopLimit*). The parameter *data* is a pointer to the buffer chain containing the message data. *address* is the single-address destination for the message, drawn from the physical node address space. *effort* is an integer specifying the transmission effort level to use at each hop (1 to *MAXLEVEL*). *hopLimit* is an integer specifying the maximum number of hops over which the message may be forwarded on its way to the destination address (1 to *MAXHOPS*). [2,3,5,6,8–10,13,14,16,19,23,25,26,28,31,34,36,37,39]

sendMulti(*data*, *address*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above, except that *address* is drawn from the multicast-group addressing space. [14,23]

sendList(*data*, *addList*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above, except that *addList* is a pointer to a buffer chain containing the list of destination addresses drawn from the physical node address space; the number of addresses in the list can be determined from the length of the data (in bytes) in the *addrList* buffer chain. [6,9,25,30,34,36,37,39]

Routing for *sendSingle*() can be done using any existing multihop wireless unicast routing protocol; this problem has been well studied in the literature. Likewise, for *sendMulti*(), routing may be done using any existing multihop wireless multicast routing protocol; although this problem has received less attention in sensor networks, it has been well studied in the multihop wireless ad hoc networking community.

Routing for *sendList*() can be done by leveraging the unicast routing protocol used for *sendSingle*(). For example, the sending node can determine the first hop toward each of the destinations listed in *addrList*. For all destinations with a common first hop, the sending node can forward a single copy of the packet to that first-hop node; this process is repeated for each distinct first-hop node needed for the routes to all destinations listed in *addrList*. In the header of the packet sent to each unique first-hop node in this way, the sending node includes a list of all destinations from *addrList* reachable through that specific first-hop node. Each of these nodes, upon receiving the packet, then repeats this process with the remaining address list.

Multi-hop transmissions are only allowed to propagate a limited number of hops (*hopLimit*) to allow applications to control the scope of their packets and as a safeguard against routing loops in the underlying routing protocols.

4.2 Region-Based Sending

Applications may often want to address all nodes within certain geographic constraints. This may include all nodes within a certain number of hops of a given node, all nodes within a certain radius of a given node, or all nodes within an arbitrary region of space. The API calls to support this are specified as follows:

sendHopRad(*data*, *hopRad*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above, except that *hopRad* is an integer specifying a hop-count from the sending node within which all neighboring nodes are intended to receive the message. *hopRad* can take a value ranging from 1, corresponding to immediate radio neighbors, to *MAXHOPS*, corresponding to a network-wide flood. [2–5, 10, 14, 16, 18, 19, 29, 31, 34, 36, 38]

sendGeoRad(*data*, *geoRad*, *outHops*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above, except that *geoRad* is a floating point number specifying a geographic distance (in standardized units) from the sending node within which all neighboring nodes are intended to receive the message, and *outHops* specifies the maximum number of hops that packets are allowed to propagate outside the specified region in order to route around voids inside the region, attempting to reach all intended nodes. [1, 4, 14, 34]

sendCircle(*data*, *centerX*, *centerY*, *radius*, *single*, *outHops*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above, except that *centerX* and *centerY* are floating point numbers that define the coordinates of the center of a circle, and *radius* is a floating point number specifying the radius from that point within which all nodes are intended to receive the message (all in standardized units). *single* is a boolean flag indicating how many sensors in the area must be reached: *single* = *1* specifies that only one sensor in the area must receive the message [23], whereas *single* = *0* specifies that all sensors in the area are intended to receive the message. [14, 23, 27]

sendPolygon(*data*, *vertCount*, *vertices*, *single*, *outHops*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above, except that *vertCount* is an integer specifying a number of polygon vertices (1 to some number *MAXVERTS*), and *vertices* is a pointer to an array of floating point numbers representing the spatial coordinate pairs of the vertices (in standardized units). *single* = *1* specifies that only one sensor within the convex hull formed by the vertex list must receive the message [13, 23], whereas *single* = *0* specifies that all sensors in this area are intended to receive the message. [13, 14, 23, 37]

The *sendHopRad*() API call can be implemented by any form of a flooding protocol (or using a spanning tree protocol); as a special case, if *hopLimit* = *1*, *sendHopRad*() can be implemented as a single link-layer broadcast transmission of the packet. The *sendGeoRad*() API call is similar, except that nodes forward the flood if they are still inside the specified *geoRad* radius around the originating node, or if they are within *outHops* beyond the first node encountered outside this radius. The use of *outHops* increases the chance of reaching all nodes inside the radius (at the expense of increased overhead, as controlled by the application), despite the presence of voids inside the circle.

For the *sendCircle*() call, routing can be done by adapting any existing geographic routing protocol; the problem of geographic routing has been well studied in the literature. The sending node routes the packet to geographic coordinates that are the center of the circle (*centerX*, *centerY*). However, once the packet is received by (or overheard by) the first node that is inside this circle (the node need not be at the center coordinates), geographic forwarding of the packet terminates, and this node instead initiates a form of the protocol used for *sendGeoRad*(), giving *centerX*, *centerY*, *radius*, and *outHops* to define the flood of the packet. Most flooding protocols use a unique identifier for multiple packets that are part of the same flood, to ensure that the flood expands efficiently in a well controlled manner; by assigning the unique identifier at the original sending node (the node initiating the *sendCircle*() call), the resulting flood will be well controlled, even if the packet under geographic forwarding is overheard by multiple nodes that all initiate copies of the flood — the different copies of the flood will in effect merge into a single flood.

For the *sendPolygon*() API call, we proceed similarly to the *sendCircle*() call, with the region boundary evaluated for flooding purposes as the convex hull of the vertex list.

4.3 Device Hierarchy Sending

In any sensor network, there will typically be a hierarchy induced by a central data sink and the nodes of the network. In a sensor network with multiple classes of non-sink devices (e.g., low power sensor nodes and higher power intermediate nodes), we support extending this device hierarchy to reflect these additional device classes. The API calls for sending in the device hierarchy are specified as follows:

setLevel(*level*). The parameter *level* is an integer specifying the level for this node in the hierarchy. We assume that each node, in an application-specific manner, has access to the level of the hierarchy it should occupy.

sendSink(*data*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above. The message is sent to the "best" available sink node in the network, where the choice of sink node is determined for the application by the API. [4, 11, 14, 15, 18, 19, 26, 30, 31, 34, 35, 37]

sendParent(*data*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above. The message is sent to the node's parent in the device hierarchy. [8, 11, 15, 35]

sendChildren(*data*, *effort*, *hopLimit*). The parameters here are as in *sendSingle*() above. The message is sent to each of the node's children in the device hierarchy. [11,15,35]

parent = getParent() returns the address of the node's current parent in the device hierarchy.

childList = getChildren() returns a pointer to a buffer chain containing a list of the addresses of the node's current children in the device hierarchy; the number of children in the list can be determined from the length of the data (in bytes) in the *childList* buffer chain. This API call supports, for example, a node sending to a subset of its children by calling *sendList*() using a subset of the return child list.

newParent(*parent*). This API call is invoked by the API implementation as an event into the application when the node's parent in the device hierarchy has changed. The parameter *parent* gives the address of the node's new parent.

newChildren(*childList*). This API call is invoked by the API implementation as an event into the application when one or more of the node's children in the device hierarchy has changed. The parameter *childList* is a pointer to a buffer chain containing a list of the node's current children; the number of children in the list can be determined from the length of the data (in bytes) in the *childList* buffer chain.

In order to form the device hierarchy, the API implementation can cause each level-n node (at all but the nodes with the largest level number) to broadcast a message announcing itself as a level-n device. Any level-$(n-1)$ node hearing this message may consider itself as a potential parent and contacts the level-n node with this information. The level-n node then chooses its parent from the responding level-$(n-1)$ nodes and informs that parent of its status as a level-n child. The lowest level of nodes (typically the lowest power sensor nodes) never advertise themselves and only look for messages from potential parents. The highest level (sink) node(s) never look for potential parents.

The API implementation maintaining this process can periodically update parent assignment to cope with changing network conditions. When a node's parent changes, the API implementation invokes the *newParent*() event in the application if the application had earlier requested knowledge of its parent through a *getParent*() call. Likewise, when a node's list of children changes (a new node becomes a child, or an old child is no longer associated with this node), the API implementation invokes the *newChildren*() event in the application if the application had earlier requested knowledge of its children through a *getChildren*() call.

We considered providing a *sendLevel*() call in our proposed API, as a more general form of the *sendParent*() and *sendChildren*() calls. Such a *sendLevel*() call, given a level number in the device hierarchy, would send to the respective node(s) at that level. For example, for a node at level n ($n > 1$), a *sendLevel*() to level $n-1$ would be equivalent to a *sendParent*() call, and to level $n+1$ would be equivalent to a *sendChildren*() call; a *sendLevel*() call need not be limited, however, to only sending to levels $n-1$ and $n+1$, creating an easy way to send to "grandchildren" (level $n+2$) and "great grandparents" (level $n-3$), for example. We did not include such a call in the API, though, since we did not find the need for this generality in our survey of application requirements, as presented in Section 2.

4.4 Receiving

The three main receive functions correspond to the three receiving modes. A node can (1) receive a message for which it is the target destination, (2) intercept and potentially modify a message for which the node is a forwarder on the path to a different target destination, or (3) passively eavesdrop (without modification) on all messages overheard by the node's radio receiver. The API calls for receiving messages are specified as follows:

receiveTarget(*data*, *metadata*). This API call is invoked by the API implementation as an event into the application when a message has been received. The parameter *data* is a pointer to a buffer chain holding the message data, and *metadata* is a pointer to a buffer chain holding the header information appropriate for the message packet type.

receiveForward(*data*, *metadata*). The parameters are the same as in *receiveTarget*(). This API call is invoked by the API implementation as an event into the application when a message has been received. If the application returns a zero result value in response to this call (the function return value), then the message will *not* be forwarded (forwarding stops at this node). If instead the application returns a nonzero result value, the message (possibly modified by the application) will be forwarded along to the next hop to the destination(s).

receiveOverhear(*data*, *metadata*). The parameters are the same as in *receiveTarget*(). This API call is invoked by the API implementation as an event into the application when a message has been received. Any message overheard by this node's radio will be received, regardless of the addressing of the message. [14, 25, 35]

Within any of these receive events within the application, the application handler for that event can then parse the metadata for any packet-specific fields it cares to extract. All packet types will support the following two calls:

type = getPacketType(metadata)
sender = getSender(metadata)

where *type* is an integer corresponding with one of the packet classes, and *sender* specifies the address of the sender in the physical address space. With this type information in hand, the application layer can then use similar calls to extract packet fields such as the destination address for *sendSingle*() or the (remaining) list of destinations for *sendList*(), the hop radius for *sendHopRad*() or the geographic radius for *sendGeoRad*(), or the center and radius for *sendCircle*() or the vertices for *sendPolygon*().

5. APPLICATION EXAMPLES

We now present detailed treatments of a selection of the surveyed papers [13, 14, 23, 34], chosen for the range of communication patterns they exhibit. We briefly describe the objective of each proposed algorithm and show how it can be implemented using the API calls outlined in the previous section.

5.1 TinyDB

In [14], the authors describe the TinyDB query engine for sensor networks. Basic aggregate queries — such as minima, maxima, averages, counts, and sums — are posed at the network's data sink. The sink broadcasts the query to the network with an incrementing hop counter that allows nodes to form an application-specific hierarchy to service the query. Each node receiving the query picks a parent one-hop closer to the sink than itself and re-broadcasts the query to potential children in its 1-hop neighborhood. This query dissemination is handled as a network-wide flood at the sink using the *sendHopRad*() call with *hoplimit* = *MAXHOPS*. As the query forwards, each node can extract its minimum distance to the sink from the decrementing *hoplimit* field.

Once the query has reached the entire network, each leaf node (a node that has no children) evaluates the query using its data and forwards the result to its parent. Each parent aggregates messages from its children with its own datum and forwards the result to its parent until the final aggregate is computed at the sink. Each child-to-parent message is sent using the *sendSingle*() API call.

TinyDB extends basic sink query servicing to support more advanced event-driven queries issued from nodes in the network. On detection of an event (say, a bird entering a nest monitored by a node), the node can query all neighbors within a specified radius for data (such as light and temperature) using the *sendGeoRad*() call. Data from the neighboring nodes returns to the querying node using *sendSingle*() calls, and a report is sent to the sink using *sendSink*() addressing. Alternatively, the report can be sent to a storage point within the network which is addressable by all nodes but does not occupy a fixed location. In this case, the querying node sends its report to a multicast group of storage nodes using the *sendMulti*() call.

As an example of the utility of TinyDB, the authors consider the problem of tracking a vehicle using magnetometer readings at each sensor. Nodes in the network initially start out in a low power mode, and those in the vicinity of the target power up when it first enters the sensor network using dedicated wake-up circuitry. Each active node then monitors the running average of its magnetometer and when a threshold is exceeded inserts its measurement and node identifier into a storage point (again using the *sendMulti*() call). The storage point estimates the target's location using the location of the node with the strongest reading. Nodes can also eavesdrop on their neighbors' messages to the storage point using the *receiveOverhear*() call and suppress their own transmissions when neighbors have a stronger reading. Note that use of the *receiveForward*() receiver would also allow nodes with stronger readings to suppress forwarding messages from nodes with weaker readings. Finally, when a sensing node detects that the target is moving out of range, it can wake up nodes in the next area to be traversed by the target using a region-based send such as *sendCircle*() or *sendPolygon*().

5.2 Distributed Multi-Target Tracking

The authors of [23] tackle the problem of tracking multiple maneuvering targets in a decentralized fashion. Maintaining a complete record of the joint state space of targets in any one component of the sensor network is impractical; it is far more efficient to keep state information for each target in tracking agents occupying nearby nodes. When targets' paths cross, however, their locations must be estimated jointly, and target identity upon track divergence becomes uncertain. Agents separately tracking each target following the split must maintain a dialogue to determine which of the targets they are actually tracking.

Before a merge, each target is tracked by an agent occupying a node designated as the track leader. To localize its target, the node collects measurements from other nodes in the vicinity of the target — a so-called "geographically constrained group" (GCG) defined as a circle or polygon. This process involves *sendCircle*() or *sendPolygon*() API calls, and each measurement returns to the track leader via a *sendSingle*() call. As the target moves, the track leader duty is passed to a new node in the target's path using *sendCircle*() or *sendPolygon*() with the single-node-only option. When two targets come within a threshold range of each other, the two GCGs are merged and serviced by a single track leader. As the two targets subsequently diverge, two new GCGs, each with its own track leader, are created. To disambiguate target identities, nodes serving as track leaders must maintain a dialogue for a time, so they form an "acquaintance group" (AG). Since tracking agents move from node to node to follow their targets, we represent the AG as a multicast group to which each leader subscribes. Messages passed between leaders to sort out target identities are then sent using the *sendMulti*() call.

5.3 Fractional Cascading

In [13], the authors present a framework called Fractional Cascading for answering range queries injected from anywhere in the network. Each node in the network has a global view of the measurement field that decays in resolution proportional to distance from the node — that is, a node knows much more about measurements from nearby nodes than from those far away. This allows queries posed at arbitrary locations, counting or enumerating all sensors in a rectangular area whose measurements lie in some range, to be efficiently routed to regions with relevant information.

The structure that enables efficient query processing is a virtual quadtree partition of the bounding box containing the sensor field. This square box is partitioned into four sub-squares, and each of these is recursively partitioned into four smaller sub-squares, and so on, until a minimum square size is reached. At each scale, the larger square giving rise to the four smaller squares is declared a parent and the smaller squares its children. To guide query routing, the maximum measurement value of any sensor node in a given quadtree square is stored in the nodes in all children of that square's parent — i.e., its sibling squares. This structure is built in a bottom-up fashion. One node in a square (which by induction has access to the maximum value in each of that square's children) computes the square's maximum value and sends the value to all nodes in the square's siblings, implemented using a *sendPolygon*() call.

The answer to each range query is compiled in three steps. First, the query is directed toward a sensor in the region of interest, which requires a *sendPolygon*() API call with the single-node-only option. Once in the region, the query sequentially visits each sub-region designated as a "canonical piece" — that is, a quadtree square which is completely contained inside the query region but whose parent square extends outside the region. This again necessitates a *sendPolygon*() API (single-node-only) call. Once the query has arrived at any sensor in the canonical region, it can exploit the distributed information structure to efficiently traverse the sub-tree for that region, recursively visiting each child square in a fashion appropriate to the query, again using a *sendPolygon*() API (single-node-only) call. For example, if the query wishes to compile a list of sensors with measurements above a threshold value, the query need not recur on any children or siblings of a square with a maximum temperature below the threshold. Since any sensor in each square records the maximum value in that square and its sibling squares, the query can efficiently traverse the squares of each canonical region's subtree.

Finally, the aggregated query data return to the querying node using a *sendSingle*() API call.

5.4 Distributed Wavelet Compression

In [34], the authors propose a distributed wavelet compression algorithm. This entails first computing a distributed wavelet transform of data within the network and then selectively streaming wavelet coefficients from nodes to the sink. First, however, the sink must learn of each node's self-localized position, sent using a *sendSink*() call by each node. Using this information, the sink computes a set transform data for each node and sends it to the node using a *sendSingle*() call.

Given this transform data, the multi-scale wavelet transform, based on the theory of wavelet lifting, proceeds in the network as follows. A subset of nodes at each scale are designated to compute wavelet transform values. Each node in this set computes its value — called a wavelet coefficient — using values from neighbors not generating wavelet coefficients. Each such neighbor knows which nodes will require its value — called a scaling coefficient – and transmits this value to those nodes, using an instance of the *sendList*() call. Each node which computes a wavelet coefficient collects these neighboring values, computes its coefficient, and sends the coefficient value back to the neighbors, again using an instance of the *sendList*() call. Each neighbor collects the list of wavelet coefficients sent to it and uses the information to compute a new coarser-scale scaling coefficient for itself. The process then repeats at the next scale on the remaining scaling coefficient nodes, with a subset giving rise to wavelet coefficients at the new scale and the remainder participating as scaling coefficients in further scales of the transform.

Once the transform has iterated to a final, coarsest scale, each node has a transform coefficient replacing its original measurement. This set of coefficients is much more sparse than the original measurement set — in other words, the energy of the measured signal is concentrated at far fewer nodes. To harvest a lossily compressed version of the measurement field, the sink broadcasts a threshold to all sensors using the *sendHopRad*() call with $hoplimit = MAXHOPS$. Upon receiving this threshold query, each node with a coefficient whose magnitude is above the threshold sends its coefficient to the sink using the *sendSink*() call. The process may repeat with subsequent threshold broadcasts from the sink and node replies until it has harvested enough coefficients to reconstruct the field to some desired fidelity.

To provide robustness to occasional node and routing failures, [34] also proposes a mechanism for nodes to repair transform data in a distributed fashion. If a wavelet coefficient-generating node cannot hear from a required neighbor, it can begin to search for new neighbors outward in an expanding radial neighborhood. Such a request is implemented using the *sendGeoRad*() call, and replies from potential neighbors return using the *sendSingle*() call. New and remaining neighbors must be informed of this change using further *sendList*() and *sendSingle*() calls.

6. CONCLUSIONS

We have presented a network API for sensor networks. The selection of the abstract services that the API provides has been informed by an extensive survey of over 100 data processing algorithms. Moreover, guided by the characteristics of sensor network hardware and software, we have carefully made a set of design decisions that covers a wide range of issues including addressing modes, receiving modes, transmission reliability enhancement, message fragmentation, system memory management, and device hierarchy formation.

By presenting a concrete API, we hope to stimulate further discussions in the research community on API design issues for sensor networks. In addition, we believe the API also opens new research directions. Specifically, the presented API frames two new challenges for future research to address: (1) how to optimally enhance transmission reliability given an energy budget and the current environmental conditions, and (2) how to optimally self-organize nodes dynamically into device hierarchies.

7. REFERENCES

[1] Z. Abrams, A. Goel, and S. Plotkin. Set K-cover algorithms for energy efficient monitoring in wireless sensor networks. In *Proc. of IPSN*, pages 424–432, 2004.

[2] M. A. Batalin and G. S. Sukhatme. Coverage, exploration, and deployment by a mobile robot and communication network. In *Proc. of IPSN*, pages 376–391, 2003.

[3] P. W. Boettcher and G. A. Shaw. Energy-constrained collaborative processing for target detection, tracking, and geolocation. In *Proc. of IPSN*, pages 254–268, 2003.

[4] Q. Cao, T. Abdelzaher, T. He, and J. Stankovic. Towards optimal sleep scheduling in sensor networks for rare-event detection. In *Proc. of IPSN*, pages 20–27, 2005.

[5] J. Y. Chen, G. Pandurangan, and D. Xu. Robust computation of aggregates in wireless sensor networks: Distributed randomized algorithms and analysis. In *Proc. of IPSN*, pages 348–355, 2005.

[6] K. Chintalapudi, J. Paek, O. Gnawali, T. S. Fu, K. Dantu, J. Caffey, and R. Govindan. Structural damage detection and localization using NETSHM. In *Proc. of IPSN*, pages 475–482, 2006.

[7] Chipcon. CC2420 Data Sheet. `http://www.chipcon.com/files/CC2420_Data_Sheet_1_4.pdf`.

[8] M. Coates. Distributed particle filters for sensor networks. In *Proc. of IPSN*, pages 99–107, 2004.

[9] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran. Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes. In *Proc. of IPSN*, pages 111–117, 2005.

[10] A. G. Dimakis, A. D. Sarwate, and M. J. Wainwright. Geographic gossip: Efficient aggregation for sensor networks. In *Proc. of IPSN*, pages 69–76, 2006.

[11] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: Enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proc. of IPSN*, pages 407–415, 2006.

[12] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis. Collection. `http://www.tinyos.net/tinyos-2.x/doc/html/tep119.html`.

[13] J. Gao, L. J. Guibas, J. Hershberger, and L. Zhang. Fractionally cascaded information in a sensor network. In *Proc. of IPSN*, pages 311–319, 2004.

[14] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Toward sophisticated sensing with queries. In *Proc. of IPSN*, pages 63–79, 2003.

[15] W. Hu, V. N. Tran, N. Bulusu, C. T. Chou, S. Jha, and A. Taylor. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *Proc. of IPSN*, pages 503–508, 2005.

[16] A. T. Ihler, J. W. F. III, and R. L. Moses. Nonparametric belief propagation for self-calibration in sensor networks. In *Proc. of IPSN*, pages 225–233, 2004.

[17] S. Keshav. An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Netowrk. Addison Wesley. 1997.

[18] A. Krause, C. Guestrin, A. Gupta, and J. Kleinberg. Near-optimal sensor placements: Maximizing information while minimizing communication cost. In *Proc. of IPSN*, pages 2–10, 2006.

[19] B. Krishnamachari and S. S. Iyengar. Efficient and fault-tolerant feature extraction in wireless sensor networks. In *Proc. of IPSN*, pages 488–501, 2003.

[20] P. Levis. message_t. `http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html`.

[21] P. Levis. Packet Protocols. `http://www.tinyos.net/tinyos-2.x/doc/html/tep116.html`.

[22] P. Levis. TinyOS Programming. `http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf`.

[23] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. Distributed state representation for tracking problems in sensor networks. In *Proc. of IPSN*, pages 234–242, 2004.

[24] M. K. McKusick and G. V. Neville-Neil. The Design and Implementation of the FreeBSD Operating System. Addison Wesley. 2005.

[25] M. Paskin, C. Guestrin, and J. McFadden. A robust architecture for distributed inference in sensor networks. In *Proc. of IPSN*, pages 55–62, 2005.

[26] S. Pattem, B. Krishnamachari, and R. Govindan. The impact of spatial correlation on routing with compression in wireless sensor networks. In *Proc. of IPSN*, pages 28–35, 2004.

[27] S. Pattem, S. Poduri, and B. Krishnamachari. Energy-quality tradeoffs for target tracking in wireless sensor networks. In *Proc. of IPSN*, pages 32–46, 2003.

[28] M. Rabbat and R. Nowak. Distributed optimization in sensor networks. In *Proc. of IPSN*, pages 20–27, 2004.

[29] A. Savvides, W. Garber, S. Adlakha, R. Moses, and M. B. Srivastava. On the error characteristics of multihop node localization in ad-hoc sensor networks. In *Proc. of IPSN*, pages 317–332, 2003.

[30] N. Shrivastava, S. Suri, and C. Tóth. Detecting cuts in sensor networks. In *Proc. of IPSN*, pages 210–217, 2005.

[31] A. Terzis, A. Anandarajah, K. Moore, and I. Wang. Slip surface localization in wireless sensor networks for landslide predection. In *Proc. of IPSN*, pages 109–116, 2006.

[32] TinyOS Community Forum. `http://www.tinyos.net/`.

[33] R. Wagner, M. Duarte, J. R. Stinnett, T. S. E. Ng, D. B. Johnson, and R. Baraniuk. A network API-driven survey of communication requirements of distributed data processing algorithms for sensor networks. Technical report, Rice University, 2006. `http://www.ece.rice.edu/~rwagner/IPSN-API-survey.pdf`.

[34] R. S. Wagner, R. G. Baraniuk, S. Du, D. B. Johnson, and A. Cohen. An architecture for distributed wavelet analysis and processing in sensor networks. In *Proc. of IPSN*, pages 243–250, 2006.

[35] Q. Wang, W. Chen, R. Zheng, K. Lee, and L. Sha. Acoustic target tracking using tiny wireless sensor devices. In *Proc. of IPSN*, pages 642–657, 2003.

[36] W. Wang and K. Ramchandran. Random distributed multiresolution representations with significance querying. In *Proc. of IPSN*, pages 102–108, 2006.

[37] R. Willett, A. Martin, and R. Nowak. Backcasting: Adaptive sampling for sensor networks. In *Proc. of IPSN*, pages 124–133, 2004.

[38] L. Xiao, S. Boyd, and S. Lall. A scheme for robust distributed sensor fusion based on average consensus. In *Proc. of IPSN*, pages 63–70, 2005.

[39] G. Xing, C. Lu, R. Pless, and J. A. O'Sullivan. Co-grid: an efficient coverage maintenance protocol for distributed sensor networks. In *Proc. of IPSN*, pages 414–423, 2004.