Design of an Application-Cooperative Management System for Wireless Sensor Networks

Gilman Tolle
EECS Department
University of California, Berkeley
Berkeley, CA 94720
Email: get@cs.berkeley.edu

David Culler
EECS Department
University of California, Berkeley
Berkeley, CA 94720
Email: culler@cs.berkeley.edu

***Abstract*— This paper argues for the usefulness of an application-cooperative interactive management system for wireless sensor networks, and presents SNMS, a Sensor Network Management System. SNMS is designed to be simple and have minimal impact on memory and network traffic, while remaining open and flexible. The system is evaluated in light of issues derived from real deployment experiences.**

I. Introduction

During the spring of 2004, 80 mica2dot sensor network nodes were placed into two 60 meter tall redwood trees in Sonoma, California. These nodes would collect environmental readings every five minutes, and forward them through a multihop network to a base station located between the two trees. One month later, initial examination of the gathered data showed that the nodes in one tree had been entirely unable to contact the base station. Of the 33 remaining nodes, 15% returned no data. Of the 80 deployed nodes, 65% returned no data at all, from the very beginning.

This problem was not detected on the day of the deployment for two reasons: the algorithm used to form the collection tree was designed to converge slowly, and with a 5 minute wait between opportunities to contact each node, the deployers were unable to adequately test the network in their allotted time. We suggest that the deployed application was incompatible with the needs of human managers, due to the tightly constrained power budget. However, we argue that every sensor network, running any application, should be able to present a human manager with the ability to quickly determine whether a deployed network is functioning.

One week into the Sonoma deployment, another 15% of the nodes died, likely due to premature battery exhaustion caused by a time synchronization failure that prevented the node from entering sleep mode. But, this was later inferred from the collected data, and no records exist of the events that may have caused this failure. Authors of TinyOS components have the ability to check for these failure conditions arising from lower-level systems, and combine this data to determine higher-level failure conditions. But, this data is rarely available to a human manager. We argue that every sensor network should be able to record these events to permanent local storage for post-mortem analysis, and that this event record should also be accessible to a human manager in real time.

II. SNMS Design Principles

This paper presents SNMS, an application-cooperative management system for wireless sensor networks. SNMS provides two core services, as suggested by the scenarios above:

- a query system to enable rapid, user-initiated acquisition of network health and performance data
- a logging system to enable recording and retrieval of system-generated events.

Because SNMS is intended to operate alongside any sensor network application, we place the following restrictions upon our design:

1) SNMS must occupy a minimal amount of RAM.
2) SNMS must generate network traffic only in response to direct human action, and should generate no network traffic in the steady state.
3) SNMS must be simple and robust.
4) SNMS must depend on the application as little as possible, to ensure that it will continue to function even when the application fails.

This last requirement gives rise to a third core service provided by SNMS:

- a lightweight network layer that can operate in parallel with the application's networking layer.

In addition to operating in an application-cooperative mode, we also envision that SNMS will also be used as a fallback system. In conjunction with the Deluge [1] network programming system, a compiled binary containing the core SNMS functionality can be loaded in response to a runtime command or an application failure that causes a watchdog timer to reboot the node. Thus, even if the application cannot admit SNMS, SNMS could still provide management services during the initial deployment, and during catastrophic application failure.

By working with hardware developers to preload SNMS onto sensor network nodes, we also hope to establish an "out-of-box experience" for wireless sensor networks. Using SNMS in this way may encourage more advanced research, by lessening the amount of time spent trying to get the network to "say hello". A preloaded program could also assist in calibration of the nodes prior to deployment, and lessen the burden of assigning unique addresses, but we leave this for future work.

To meet our requirements of simplicity and robustness, we have designed SNMS to clearly separate mechanism from policy. The mechanisms provided by SNMS include marshalling, transport, and unmarshalling for commands, data, events, and schema attributes. The policies of what can be managed, what will be managed, and which nodes will participate are decided by the users of the system. Furthermore, SNMS clearly separates the responsibilities of policymaking. The author of a component must decide which attributes it will export and which events it will generate, in addition to the names of the attributes and the descriptions of the events. The network manager must decide which attributes and events should be monitored, and then must interpret the results.

SNMS does not impose any semantics of its own. The system does not depend on a fore-ordained taxonomy of attribute types, like "tree parent" and "number of dropped packets", nor does it categorize and prioritize events by any means other than the natural ownership boundaries of components. This open architecture will ensure that SNMS is usable by the largest number of developers.

We begin by detailing the SNMS network architecture in Section III, and then examine the two core SNMS management services in Section IV and Section V. We describe a collection of higher-level management services built atop SNMS in Section VI. Finally, we evaluate the networking performance of SNMS in Section VII, and suggest a number of ways in which an application-cooperative management system could improve the quality of data obtained from a real deployment.

III. Network Architecture

Because SNMS is intended to cooperate with an application, but continue functioning when the application fails, we argue that a management system should contain its own networking stack that can run in parallel with the application's. The alternate approach of allowing SNMS to use the application's stack would require less RAM, less code, and would prevent redundant network maintenance traffic, but these drawbacks can be minimized with careful design of the networking layer. Thus, our exploration of the design space has been shaped by a decision to deliberately create redundancy in an environment that has previously admitted none. However, SNMS is structured as a collection of separate components, and if the application designer would prefer to eliminate this redundancy, then SNMS could cooperate with any layer providing the proper interfaces.

The SNMS networking architecture supports two traffic patterns: Collection and Dissemination [2]. Collection is required to obtain health data from the network, and Dissemination is required to distribute management commands and queries. The need for these traffic patterns reflects the absence of point-to-point connections between individual nodes.

The first contribution of the SNMS networking stack is a collection tree construction protocol that minimizes state requirements by not requiring a neighbor table, and minimizes network traffic by requiring explicit initiation of tree construction. The tree construction protocol does remain adaptive to changing network conditions, and makes no assumptions about the underlying topology.

The second contribution is an interface and stack for transport-level reliable dissemination of messages. Other implementations of reliable dissemination have so far been application-specific [1] [3] [4]. Currently, this networking layer is the only one supported by SNMS, but it has been separately developed so that future work can study how well SNMS would perform atop an application's network stack, and how well the SNMS network stack would support other applications.

A. Collection

The initial task of network management is the collection of operational data from the network under study. In order to retrieve data from the managed nodes, and transport it to a management workstation, SNMS requires a protocol that will generate a network structure for data collection.

Many such protocols have been developed for the sensor network space. The MintRoute protocol organizes a collection of nodes into a tree rooted at a specific node that acts as a base station. Each node in a MintRoute network periodically sends an announcement message, which is used for link quality estimation by every neighboring node. Performing this link estimation requires each node to maintain a table containing every other neighboring node, which greatly increases the storage cost of using the MintRoute routing layer. These link quality estimates are then used to form a tree that minimizes the expected number of transmissions necessary to forward a message to the root [5].

The interest diffusion process of Directed Diffusion also creates a tree rooted at the node generating the interest, called the sink. Instead of using periodic beacons to maintain a constant tree, each node selects the neighbor from which it first received a flooded interest message [6]. Other studies have shown this approach to be problematic in the presence of widely varying link qualities. To improve the quality of this link-reversal tree, Directed Diffusion also uses a large amount of storage to maintain link qualities in a neighbor table, and only considers sufficiently well-connected nodes to be neighbors.

1) Tree Construction and Refinement: SNMS uses a collection tree construction protocol that combines these two approaches. Because SNMS runs alongside other sensor network applications, we argue that SNMS should not generate any maintenance traffic when the network is not being actively managed. Thus, our tree construction protocol cannot use periodic beacons as MintRoute does, and must only construct a tree in response to a message sent from the root. However, to construct a higher-quality tree than the one produced by selecting the neighbor that sent first, we continually update the parent selection as new messages arrive. We avoid contention while flooding the construction message by randomly staggering the retransmission time for each node.

Because SNMS does not have periodic traffic to estimate the quality of each potential parent, and does not have a neighbor table in which to store this information, SNMS must use an estimator that can produce an estimate immediately upon receipt of a tree construction message. The protocol estimates link qualities between nodes by measuring the received signal strength of the incoming tree construction message. Upon receipt of each tree construction message, the node adds the message's RSSI to the cumulative RSSI value in the message and selects the sender if the new total is less than the current parent's path cost. By only maintaining the single best parent, we remove the need for a RAM-consuming neighbor table. Future work for the protocol will examine the benefits of maintaining k best parents instead of the single best parent.

Minimizing the sum of RSSI values along a path will produce a path with maximally strong signal. In addition, the tree constructed by this protocol will contain more hops over stronger links. This may not be desirable when the number of forwarding nodes must be minimized, but minimizing energy consumption is not required for short-lived interactive management applications. Additionally, this approach may result in more contention, but quantifying the exact properties of this RSSI tree construction technique will be left for future work.

However, this protocol may construct trees containing asymmetric links because the RSSI value only indicates the strength of the inbound link. The SNMS tree construction protocol accounts for link asymmetry by combining RSSI with a real-time link quality metric. This metric requires link-layer acknowledgement messages to be sent in response to every forwarded message. After selecting a new parent, a node begins to maintain a windowed moving average of the acknowledgement rate for messages forwarded to that parent. The inverse of this rate is then scaled to the same range as the RSSI value, and added to the current estimate. For purposes of comparison, the message success rate of a potential parent is assumed to be 50%. Again, maintaining this estimate only adds a constant amount of state to the routing layer.

2) Management-Specific Tree Enhancements: Most deployed wireless sensor networks connect to an external network at a single point. Future sensor networks, however, may be designed with multiple interconnection points. In addition, a sensor network could be managed from a mobile workstation or PDA-class device located within the physical boundaries of the network itself. To accommodate these multiple-root scenarios, our collection tree protocol includes the identifier of the root node in the construction message. The parent selection process then doubles as a root selection process.

Multiple trees may be constructed simultaneously or sequentially, which introduces the problem of tree selection. When construction of one tree from one root ceases, and construction of a new tree begins at a new root, the protocol must allow nodes to leave the old tree and join the new. Nodes close to the original root, if allowed to maintain their most recent path cost estimate, would not shift to a new root because it would require an increase

in the path cost.

We address this problem by setting a minimum rate for tree construction messages. If enough messages are missed in sequence, the node will begin to increase its link cost estimate to the currently selected parent and will eventually select a new parent from an active tree. This requirement does place a non-zero lower bound on the amount of traffic required, but this traffic is only necessary during active maintenance of the management tree. If no data is being collected, the tree can be allowed to remain static, and the aging timer can be stopped.

B. Dissemination

The Simple Network Management Protocol [7] relies on point-to-point transport that will deliver control messages and queries to an individual device being managed. In contrast, wireless sensor networks act in aggregate, and thus, a wireless sensor network management system must be able to manage in the aggregate. Aggregate management requires a dissemination protocol that can deliver messages reliably to a set of nodes within a sensor network. The underlying algorithm used by our dissemination layer is the Trickle algorithm [8]. Trickle uses periodic retransmissions to ensure eventual delivery of the message to every node in the network. To minimize the number of required messages, retransmissions can be suppressed by prior transmissions of similar messages, and randomization is used to prevent permanent suppression. Our dissemination layer takes the Trickle retransmission algorithm and builds a transport-layer interface atop it.

1) Unnamed Reliable Dissemination: The SNMS dissemination protocol, named Drip, provides a transport-layer interface to multiple channels of reliable message dissemination. Implemented as a TinyOS component, Drip provides a standard message reception interface. Each component wishing to use Drip registers a specific identifier, which represents a reliable dissemination channel. Messages received on that channel will be delivered directly to the component. Each node is responsible for caching the data extracted from the most recent message received on each channel to which it subscribes, and returning it in response to periodic rebroadcast requests. In our implementation, space for this cache is allocated by the subscribing component, and data is retrieved from the cache in response to an upcall [9] issued prior to retransmission. The Drip protocol uses a sequence number with half-space wraparound to determine whether a received message is new, and upon receipt of a new message, the data is delivered to the subscribing component for required caching and optional action.

The Drip protocol uses the message as the unit of reliability, and the component as the unit of caching. This design allows Drip to function as a standard transport-layer protocol. But, it does introduce extra complexity for a component that has several independent variables which must be reliably synchronized among all nodes in the network. To solve this problem, the component must collect the current value of each variable into a single reliably disseminated message. This method will produce independent reliability for each variable, as long as every node stores the same value of every variable.

This problem could also be solved by selecting the variable as the unit of reliability and caching, and by associating a unique key with each variable instead of associating a channel with each message. However, this approach would require a significantly larger keyspace, and would blur the boundary between protocol and data storage.

2) Named Reliable Dissemination: To implement a useful command layer using the unnamed dissemination provided by Drip, we must be able to name individual nodes or subsets of the network that should act upon the command. This ability is provided by a separate component implementing a naming interface. The SNMS naming component places three extra header fields into each Drip message: destination address, destination group, and Time To Live. The naming component is called by the client component after receiving and caching each Drip message, and if the message is destined for the node or for the group to which the node belongs, the client component acts on the message. During the upcall issued before retransmitting the cached Drip message, the naming component can prevent retransmission if the TTL has expired. Our particular implementation of the Naming interface is only one of many potential implementations. These methods could support attribute-directed dissemination, or any other naming scheme which can distinguish between intermediary nodes and endpoint nodes.

The combination of Naming and Drip does slightly lessen the reliability guarantees provided by Drip. If message *A* intended for node set *S* is injected, and before *A* has reached every node in *S*, message *B* is injected for node set *S'*, some nodes in set *S* and not in set *S'* may never receive value *A*. Solving this problem would require independently caching each distinct message and recipient set indefinitely, which we rule out due to limited space available on the nodes.

3) Commands: The Drip dissemination layer as described above has proven itself to be a useful service for controlling sensor network nodes. Node commands are interpreted by individual components responsible for a well-defined aspect of the TinyOS program, and the Drip layer itself is used to dispatch incoming commands to the appropriate components. Due to the eventual consistency provided by Drip, commands should take the form of idempotent assignments to state, and not modifications or actions.

IV. Query System

Understanding the health of a sensor network requires a combination of automated data gathering and human interpretation. One of the two key contributions of SNMS is a middleware layer that allows programmers of TinyOS components to easily expose "interesting" attributes of their components to human eyes, over a deployed multihop network. Because the number of potential attributes in all components is large, SNMS provides a runtime query system by which subsets of these attributes may be selected for collection. This system is intended to be simple and robust.

Network export of interesting parameters is not a new process in sensor networks, but prior to SNMS, it has often performed in an ad-hoc fashion. For example, the MintRoute collection tree component can periodically send a specific Debug message, which contains the current parent, current link estimates, and current path costs. This data can be used to graph a network in real-time and study its stability. But from a developer's point of view, MintRoute's debugging protocol is lacking in several ways. The Debug message can only be enabled at compile time, and once enabled, is constantly sent at a fixed period. In addition, a monitoring application must understand the specific format of the message used by MintRoute in order to display the data, and if more fields are added to the Debug message, then the monitoring application will break. Under the SNMS query system, current parent and link quality become separate attributes that can be selected for remote monitoring at a user defined time and period, placing management at the user's control, not at the compiler's.

A. Attribute Export

Because a TinyOS program is constructed from a collection of components, created by many different developers, the set of potentially queryable attributes may change with each compilation, and will grow as more components are developed. This process, applied by a highly active TinyOS developer community, suggests that a static taxonomy of attributes that representing classes with well-defined semantics will quickly become outdated.

In designing SNMS, we have chosen to leave interpretation to the human posing the query, and taxonomy to the component developer. The author of each component chooses which attributes to export, and gives them human-readable names indicating their meaning. Each programmer-selected name then becomes the canonical name of its associated attribute. To prevent namespace collisions, we prefix each human-readable attribute name with the name of its enclosing component. The set of attributes is then formed by taking the union of every attribute exported by every component.

Once an attribute has been exported, the component is then responsible for providing the attribute in response to an upcall. Because the variable is accessed with an upcall instead of by copying from a memory location, the component can perform an arbitrary computation before copying the value into the buffer. Potential computations include rollover detection for counters, or access to external data. If the computation may be long-running, the component can return `FAIL` to the original upcall, and call a separate function once the value has been computed. Additionally, the storage location passed as an argument to the upcall can be transient, which enables the value to occupy no RAM at all if it can be computed or obtained from external sources.

B. Schema Construction

Because the canonical name of an attribute is a human-readable string, query by name is infeasible due to the small message size and limited bandwidth of nodes in a sensor network. This stands in opposition to the TinyDB and Tiny Diffusion query systems, which directly send short strings through the network as queries. Instead, SNMS uses a compact representation of each name. At compile-time, each exported attribute is assigned a small integer key, which is meaningful only in terms of the set of components comprising the TinyOS program. It is this key which will be placed into later queries. In order to establish a mapping between canonical names and local keys, the source code is scanned for all exported attributes, and then a schema is generated and stored as a file containing a list of canonical names and local keys, as well as the size of the value in bytes.

We could have required the programmer to select a fully global key for each exported attribute. This could be drawn from a flat space, as is current practice for

selecting Active Message identifiers, or it could be drawn from a hierarchical space by selecting a global key for each component and creating a locally unique key for each attribute exported by the component. This method would obviate the need for the schema file, but it also presents several drawbacks. Requiring each programmer to select a unique integer, without a central controlling authority, would be much more likely to result in collisions. The programmer already assigns a unique human-readable name to each component, the selection of which can be made easier by the programmer's knowledge of other existing components. So, we argue that the extra complexity caused by the need to map from long names to short names is outweighed by the much lesser probability of collisions in name selection. Additionally, the programmer would most likely desire to describe each attribute to a human querier, which would require a schema file mapping keys in the reverse direction – to names.

Using a variable-length key instead of our fixed 2-byte integer keys would use less message space and not unnecessarily bound the size of the keyspace. But, the earlier decision to use local keys naturally supports a smaller keyspace, and we believe that variable-length keys would create unnecessary complexity.

The decision to require a schema file in order to generate queries does raise additional problems. When the program executing within a sensor network is changed, new queries must use a schema specific to the new program to ensure that names are translated into the correct local keys. In future deployments, different nodes may be executing different application codebases simultaneously, which would require retrieving a specific schema for each node to be queried, and then constructing a query may actually require constructing different queries for each codebase.

We suggest that this problem may be solved by storing the schema in the persistent storage of each mote at program install time, thus associating it permanently with the appropriate program. The schema file or files could then be retrieved prior to query construction. This schema could be stored as an opaque data object and retrieved in its entirety, or stored as a programmatically-accessible structure that supports remote translation of individual names to keys. However, our current implementation uses offline schema synchronization, by requiring the querier to specify an appropriate file.

C. Query Processing and Response

A SNMS management query is composed of a list of attribute keys and a sample period. Upon receipt of a new active query, the node sets a response timer with the period specified in the query, and a random phase. When the timer fires, a message buffer is allocated and a pointer is initialized to the beginning of the payload portion. For each attribute in the query, the query system signals the component exporting the event, and the component writes the current value of the attribute into the buffer at the location of the pointer. This creates a list of values in the same order as the attributes in the original query, without the need for temporary storage. The system waits until all attributes have been filled, then sends the response message through the collection tree. If the query is repeating, the node resets the timer, and if it is marked as one-shot, the query is cleared.

Using the query engine to pack the results into a single message buffer creates less preamble and message header overhead than returning values individually. Additionally, it guarantees atomicity for a collection of results: for every sample period, either all values will be returned or none will. To keep the packet processing simple and robust, we require the user to submit multiple queries in order to obtain more values than will fit into a single message.

To retrieve continuous results with a one-shot query, the base station must re-inject the query every period. Using a one-shot query in this fashion acts as a safeguard against wasted network traffic caused by results from a continuous query being sent to a base station that has stopped receiving, or into a network that has become disconnected.

The packed list of values in each query result is not a self-describing data structure. Correctly parsing the result requires knowledge of the keys in the originally submitted query, their order, and the sizes of their returned values. We chose to make the results non-self-describing to lower overhead, at the cost of being unable to interpret results returning from a query when the list of keys contained in the original query has been lost. If this is necessary, then SNMS will respond to a command message that returns the details of the currently active queries.

Effectively, we are dynamically generating a response message structure for each injected query. This is as efficient an encoding as the earlier pattern of creating individual structures for each component's debugging message, but comes with the benefit of runtime control.

Note that nothing in our method precludes statistical in-network aggregation of an attribute's values, as they can be returned within the same slot that would have stored an individual node's value. However, aggregation techniques that are neither Distributive nor Algebraic according to the TAG terminology [4], techniques requiring more space to contain the aggregate than to contain the value produced by a single node, are not possible within this system.

V. Event Logging System

While the Query System supports the continuous user-driven monitoring of known parameters, post-mortem analysis and real-time monitoring of unexpected events require a second fundamental management system. The SNMS Event Logging System supports program-driven notification of one-time events. This event logging system is also structured according to the SNMS design principles of minimal footprint, simple and robust design, programmer-initiated direct naming, dynamic schema generation, and user control.

A. Programmer Interface

We draw inspiration for the SNMS Event Logger from the TOSSIM TinyOS simulation environment [10]. TOSSIM supports debugging calls, which are embedded within a nesC [11] application and display output to the terminal during execution of the program. A debugging call is effectively a call to `printf`, combining a human-readable string with the contents of multiple variables. Because TOSSIM is only a simulation environment, these debugging calls are stripped out during compilation for actual mote hardware. The SNMS Event Logger enables the programmer to use debugging calls like TOSSIM, but on mote hardware at runtime.

The event logging system, like the query system, does not embed meaning within the messages, nor does it interpret them in any way. Instead, every event is represented by a programmer-created string, which is intended to be meaningful to a human manager of the system. This string is associated with a set of values to be captured from variables at runtime. This aspect of the design was also inspired by the UNIX "syslog" facility, which has the benefit of being time-tested.

At compile-time, each call to the event logger is transformed. A unique key is assigned to each different event message, and inserted into the code. This key is drawn from a flat local keyspace, as in the key generation process for exported attributes. The strings are removed from the code, and the each variable is translated into a command that pushes the value into a message buffer. A second schema is then stored in a file. This schema contains the human-readable strings, the list of parameters and their sizes, and the unique keys.

When sending or storing a log event, the values are packed into a single message as is done in the responses to management queries. This reduces the cost of sending multiple messages and provides an atomicity guarantee for events. When a log event message is received by a workstation, the associated key is used to select the correct schema entry, which provides the parameter size information necessary to unpack the parameters from the message. The `printf` placeholders in the string are then replaced with the parameter values, and the string is output for interpretation.

B. Event Storage and Delivery

In TOSSIM simulation, log events have only one destination: standard output. Log messages generated on a running mote have three potential destinations: persistent local storage, the radio, or the local serial connection. The third is feasible only in testbed deployments, but we do provide it as an option. The second, sending log event messages directly to the radio, does not provide a strong enough guarantee that the event will be received, given the commonly observed packet loss rates across multihop sensor networks and the lack of totally reliable delivery without custody transfer. Thus, we have focused on logging events to persistent local storage, commonly realized as flash RAM.

Each event record contains a fixed-length header and a variable length data payload, and is written sequentially to persistent storage. The event record header contains the key, a sequence number, and a timestamp. This timestamp may be a local time, measured in milliseconds since the program began execution, or it may be a globally synchronized time. The event logging system can be made to use any available time synchronization component. The variable-length data payload is necessary because the number of parameters is not fixed, and each parameter may be of a different size. The event logger supports variable-length data payloads by including a length field in the header.

To retrieve the event log, the manager sends a remote playback command using the command dissemination layer, which directs a set of nodes to begin reading the log and sending the events over the multihop collection tree. Events are returned with a user-specified period, which can be used to bound the network bandwidth consumed by log events. When many nodes are sending

events simultaneously, this can be used to prevent event traffic from overwhelming the network. Additionally, this playback interface can enable real-time retrieval of events immediately after they are stored, by moving the "play head" to the head of the log.

The event log as described above is a sequence of independent events. SNMS also provides the option to associate a component identifier with each log event, transforming the event stream into a collection of interleaved streams, one per component. The user can then include a list of component identifiers in the log playback message, which will filter the events prior to network delivery.

The current version of the SNMS Event Logger delegates the responsibility of defining log events to the component author. However, a component author might not define a program event for every actual event in which a manager may be interested. Additionally, this model does not admit events dependent on information from multiple components, unless a higher-level component explicitly aggregates this information and generates its own event. Future work for SNMS includes remote specification of user-defined log triggers, which will use the attributes exported by the query system as input, and output user-defined events. We are interested in using the Maté application-specific virtual machine as a language and interpreter for these triggers.

VI. Applications

As described above, SNMS supports node control, attribute query, and event logging. To test the viability of SNMS and provide a core set of management services, we have built a collection of node control components that use the command layer, and instrumented components that use the attribute query system.

A. Identification

The changing connectivity and node failure seen in sensor networks suggests that enumeration of running nodes should be a core management service. SNMS enables the managing user to enumerate the network address, network group identifier, and unique serial number for each node in a network, in real time.

SNMS also supports identification of the program executing on each node, with the human-readable name of the program and a globally-unique identifier assigned at compile-time.

To provide basic enumeration capabilities even without an active multihop collection tree, this information can also be gathered with a special "ping" message, which prompts a link-local response. These link-local identification messages are also sent at startup, providing immediate assurance that a node has begun to execute normally.

B. Remote Sleep and Wakeup

Because sensor nodes have a limited supply of energy, SNMS provides remote power management as a core service. A command can place a set of nodes into a fully-awake state, a sleep state that can be woken with a disseminated message, or a hibernation mode from which a node must be awoken with a physical reset.

Entering the sleep state requires cooperation from a basic power management system, and waking from the sleep state requires cooperation with the network stack. The SNMS power management system places the radio stack into low-power listening mode, as defined by B-MAC [12], disables upper-level components, and lets the CPU enter sleep mode. The network stack remains active, but listens only for messages destined for the Drip dissemination layer, and the Drip layer listens only for the specific command channel used to control the power state.

Because each sleeping node samples the channel with a very long period, the wakeup message must be preceded by a very long preamble. Disseminating this message requires a small amount of special-case logic, which is easily supported by the upcall system used by Drip. During the upcall prior to periodic rebroadcast of the wakeup message, the power management component places the radio stack into long-preamble transmission mode before sending the message. This ensures that even after a node has woken from sleep, it can wake up its sleeping neighbors.

C. Physical Parameters

Because the split-phase attribute interface supports reading from physical sensors, we can use SNMS to monitor physical parameters of the node's environment. We have implemented a small component that measures battery voltage, which can be used to predict node failures [13]. Temperature and humidity around the mote also function as predictors of upcoming failure, and could be measured by appropriate components and then exported to the query system.

D. Task Queue

TinyOS computations are executed in interrupt context or in task context. Tasks are scheduled using a queue, in which the current task must finish executing before the

next task can begin. The size of this queue is fixed at compile-time, but tasks can be entered into this queue at any time and in any number. Thus, a program with a large number of components that frequently post tasks may experience queue overflows, and some tasks may be prevented from executing. Failure to post a task may lead to an upcall that never arrives at a higher-level component, which could leave that component locked forever.

SNMS provides monitoring of the task queue as a core service, by including a component that tracks the number of times the task queue has overflowed, and stores the memory address of the function for which posting has most recently failed. The counter will make the manager aware of queue overflows, and the stored pointer can assist the manager in identifying the affected component.

E. Reboot Counters

TinyOS motes often protect themselves from software failures using watchdog timers, which forcibly reboot the mote if the application does not periodically reset a flag. Another type of protection from failures is the grenade timer, which reboots the mote after the program has been allowed to execute for a fixed amount of time. The TinyOS bootloader can distinguish between these resets, in addition to resets caused by power cycling the node, resets caused by low-voltage brownouts, resets in response to a flash error detected while transferring a new program image from external to internal flash, and resets initiated by the system or user. The TinyOS bootloader, in cooperation with SNMS, keeps separate counters for each type of failure-initiated node reboot and another counter for reboots manually initiated by the TinyOS application. These counters, as well as a history of the k most recent reasons for node reboot, are accessible through SNMS.

F. Radio Stack

Performance of sensor network algorithms is commonly measured by the number of messages sent and received. Contention can be measured directly by the number of corrupt packets received, instead of indirectly by tracking missed packets across nodes as is done in the simulation analysis of Deluge [1]. Additionally, contention between TinyOS components for the radio stack can result in performance degradation and lost messages. In deployments, nodes can fail and emit streams of spurious packets (jabbering), which can be detected by neighboring nodes.

To assist the user in analyzing these aspects of radio stack performance and failure, SNMS includes an instrumented stack that provides a rich set of remotely queryable counters. The stack counts incoming and outgoing message notifications, several types of receive and send failures, and actual message deliveries in both directions. This instrumented stack provides information previously available only in simulation, allowing for profiling of algorithms in real deployments.

The counters in this instrumented radio stack actually consume more than double the RAM occupied by the rest of the stack. This highlights the tension between management and resource consumption, but it is not the management system consuming the resources. Future work includes developing a system by which developers can select an appropriate level of monitoring for standard system components.

G. Node Binding

To support fine-grained management of large sensor networks with multiple interconnection nodes, SNMS includes a system for dividing a network into subgroups which can be used to limit the propagation of commands and data. The actual selection of nodes for each subgroup can be assisted by the tree construction protocol described earlier.

When multiple base stations construct trees in parallel, each node will join the tree presenting the smallest total path cost to a base station. Tree membership, then, will divide the network into a number of groups that will cover the deployment area. After every node has selected a root, the manager can send a group establishment command that sets the group ID to x for all nodes with tree ID y. This message solidifies the dynamic divisions into static divisions, which can be used to limit the scope of newly disseminated messages. However, a command can still be disseminated across groups by addressing it to the broadcast group. This group establishment command, when sent to the broadcast group, can be used to re-establish, modify, or remove previously created divisions.

Future work will include establishing divisions by setting a common radio frequency for a collection of nodes. Because messages sent on different frequencies are rejected in hardware, the frequency equivalent of the broadcast group may require modification of the radio driver to listen on both a current frequency and a well-known hailing frequency.

TABLE I
DELIVERY PERFORMANCE OF SNMS COLLECTION

		Basic RSSI	RSSI + ACK
Msgs/Node	Mean	25.5	25.6
	Max	29	30
	Min	4	12
Yield/Node	Mean	91%	92%
	Max	100%	100%
	Min	15%	52%
Xmits/Msg	Mean	3.6	3.1
	Max	7.7	6.5
	Min	1	1

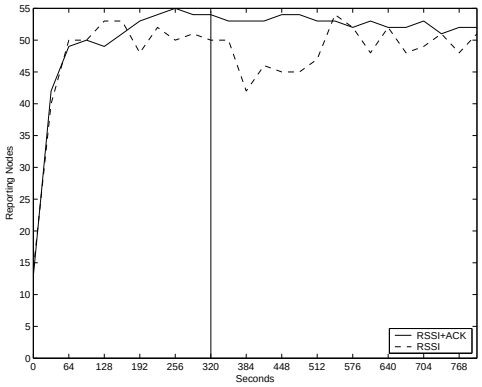

Fig. 1. Tree construction and failure response

VII. EVALUATION

To truly evaluate SNMS, we must study both the performance of the underlying network architecture, and the effects on the deployment process that may result from having an integrated management system. The first can be analyzed in the laboratory. We choose to analyze the second by inferring the effect that SNMS would have had on previous deployments.

A. Networking Evaluation

We evaluated the performance of SNMS on a 55-node subset of our in-house 78-node testbed. Each node is a mica2dot mote, connected to an Ethernet channel for reprogramming and data collection. As a feasibility test for SNMS, this Ethernet channel was not used during the following evaluation, and all data was collected over the network as though the nodes were deployed *in situ*.

1) Collection: The SNMS collection layer must meet three requirements: fast tree construction, high-yield data retrieval, and successful adaptation to node failure. To determine whether the ACK estimator made the tree construction more adaptive and selected higher-quality links without affecting the initial construction time, we examined the performance of the collection layer with and without the ACK estimator. A query was disseminated to every node, and each node responded every 32 seconds for 15 minutes. This period was chosen to be high enough to prevent contention effects, but could easily be adapted to the specific network at hand.

Table I presents a summary of the data retrieval quality results. The data indicate that basing the link quality on both the signal strength and an estimate of the acknowledgement rate improves the worst-case paths, and improves the overall quality of the paths as measured in the number of transmissions required to retrieve a message.

We then measured the time-variant behavior, including tree construction and response to node failure, in a second set of experiments under the same conditions as the first set. The tree construction message was initiated at the 32 second mark, after every one-hop neighbor of the base station had returned one query result. Because we are monitoring the responses to a query with a period of 32 seconds, the tree appears to require over a minute of construction time. In fact, the tree was likely constructed more quickly, but we chose to focus on the functional metric of query response over the abstract metric of tree construction. After 10 samples were received and all nodes had joined the tree, a node that had been selected as the parent by the most other nodes was disabled.

Figure VII-A.1 shows the number of nodes reporting in each period over time. We see that the basic RSSI estimator takes longer to re-establish a full tree, as compared to an estimator that measures quality based on acknowledged packets.

We found that the ACK estimator did not affect the time necessary to construct the tree, but greatly increased the speed of adaptation following a node failure.

2) Dissemination: We tested the performance of the dissemination layer on 88 nodes deployed in an office environment. The SNMS power management system requires dissemination of a sleep message, which we use as the normal case for Drip, and dissemination of a wakeup message that must be transmitted with an extremely long preamble because sleeping nodes sample the channel with a much lower frequency. This wakeup message represents a special challenge for the suppression mechanisms within Drip, because of its length. We compare the time to completely disseminate a message using a simple high-speed flood with no suppression or retransmissions, Drip using retransmissions but without suppression, and Drip using both retransmissions and

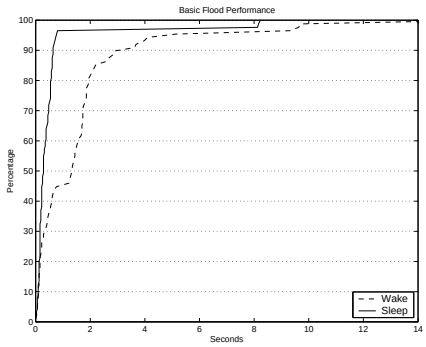

Fig. 2. No retransmissions, no suppression

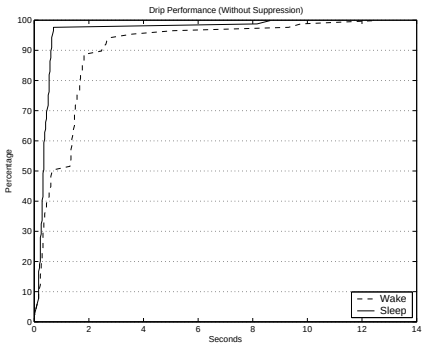

Fig. 3. Retransmissions, no suppression

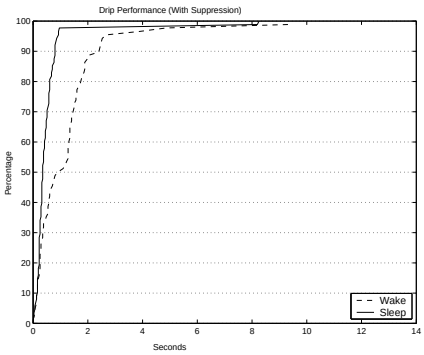

Fig. 4. Retransmissions and suppression

suppression. (Figs. 2-4).

As expected, we see that the dissemination takes longer when the messages have longer preambles. We also see that using suppression resulted in a shorter total time to complete dissemination, and a lighter tail in the reception CDF. The reception rate was 100% in all three tests, even without retransmissions. We attribute this to the high density of our testbed, and will study the performance under a wide range of more adverse conditions in future work.

3) Memory Footprint: One of the key SNMS design requirements was a small RAM footprint. Table II gives a detailed breakdown of the SNMS RAM requirements by component. Because SNMS is a cooperative system, there are many parameters that may be modified to trade off RAM footprint for program features.

The static allocation system used in TinyOS requires us to allocate buffers for each potentially active query ahead of time. We have chosen to allocate space for four simultaneously active queries, but this could easily be reduced to one. We have chosen to allocate separate buffers for the basic identification system and the input half of the event logger, to decrease the likelihood of temporary failure. We chose to finely instrument several components, but the space occupied by instrumentation counters could be reduced by accepting less detailed data gathering.

When added to the underlying components necessary to access the radio, UART, and flash memory, as well as the standard system support components, the total RAM usage of the null application plus the full SNMS configuration rises to 1281 bytes out of 4k of available RAM. With a trimmed version of SNMS, or one that makes use of the application's networking components, the amount of RAM available to the application could be increased even further. As a point of comparison, the Tiny Application Sensor Kit, which also provides multihop query processing and collection, requires 2870 bytes of RAM. This is not a feature-to-feature comparison, because SNMS does not provide in-network aggregation of query results, but TASK does not provide event logging support, scoped dissemination for node control, or dynamic schema generation.

B. Management Evaluation

SNMS, and the components built atop it, could assist in the process of sensor network deployment and health monitoring. In the Great Duck Island deployment [14], nodes that were overhearing and forwarding traffic were found to have shorter lifetimes. The instrumented network stack of SNMS, in conjunction with the radio duty cycle control, can provide enough information to estimate power consumption due to the radio and gather that data in real time for use in failure prediction. Future deployments could also estimate CPU power consumption by instrumenting the scheduler, and sensing power consumption by instrumenting the sensor drivers. SNMS provides a flexible system upon which to build these health monitoring facilities.

A lesson learned from GDI is the desirability of persistent data logging for post mortem analysis. The SNMS event logging system provides just such an ability, using an application-cooperative API that will allow integration into existing system with little programmer effort.

During the deployment of the GDI network, real-time information on the network neighborhood and reachable nodes would have assisted the deployers in constructing a more reliable network. This information should be accessible from both a base station and a field tool. The SNMS multiple-root data collection system, coupled with epidemic dissemination for commands and queries, would allow this data to be gathered from both places simultaneously.

TABLE II

RAM USAGE OF SNMS COMPONENTS (IN BYTES)

Component	Fixed Cost	Variable Costs	Chosen Variable Cost	Instrumentation	Total
Identification	11	43 × 1 msg buffer	43	0	54
Collection	15	45 × 2 msg buffers	90	14	119
Dissemination	1	8 × 9 channels	72	0	73
Management Attrs	0	1 × 43 attrs	43	0	43
Management Query	10	25 × 4 query slots	100	0	110
Event Logger	18	43 × 1 msg buffer	43	0	61
Sleep & Wake	5	0	0	0	5
Node Binding	8	0	0	0	8
Reboot	9	0	0	0	9
Radio Stack				24	24
Task Queue				1	1
Timer		10 × 7 timers	70	0	70
Shared Msg Buf		43 × 1 msg buffer	43	0	43
TOTAL	77		504	39	620

VIII. CONCLUSION

We have described SNMS, a simple and robust application-cooperative Sensor Network Management System. SNMS provides a core set of services to enable management: query-based health data collection and persistent event logging. These services occupy a minimal amount of RAM and code size, and can be rapidly integrated into TinyOS applications. To ensure that these services are usable for management and will continue to function in the event of application failure, SNMS also includes a new lightweight network architecture for collection and dissemination. Finally, SNMS provides a number of specific management components for selective inclusion into sensor network applications. The networking layer has been shown to perform acceptably, and the management functionality meets several needs derived from prior TinyOS sensor network deployments.

REFERENCES

[1] J. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[2] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The emergence of networking abstractions and techniques in tinyos," in *Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[3] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks." in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[4] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, dec 2002.

[5] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*. ACM Press, 2003, pp. 14–27.

[6] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proceedings of the International Conference on Mobile Computing and Networking*, 2000.

[7] J. Case, R. Mundy, D. Partain, and B. Stewart, "Introduction to version 3 of the internet-standard network management framework (rfc 2570)," Internet Engineering Task Force, April 1999.

[8] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks," in *Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[9] D. D. Clark, "The structuring of systems using upcalls," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1985.

[10] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: Accurate and scalable simulation of entire tinyos applications," in *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[12] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[13] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[14] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.