SWARMS: A Sensornet Wide Area Remote Management System

Charles Gruenwald, Anders Hustvedt, Aaron Beach, Richard Han
Department of Computer Science, University of Colorado, Boulder
{charles.gruenwald, anders.hustvedt, aaron.beach, richard.han}@colorado.edu

***Abstract*—Our experiences deploying a wide area wireless sensor network (WSN) in the wildfires of Idaho motivate the need for a software middleware system capable of remotely managing many sensor nodes deployed in widely disparate geographic regions. This requirement is unlike the localized focus of many traditional WSN middleware systems, which manage a group of sensor nodes deployed in a single small region, e.g. a warehouse or lab. We describe in this paper SWARMS, a wide area sensor network management system. The SWARMS architecture is designed for scalability and flexibility, while providing an infrastructure to manage in situ sensor nodes, e.g. upload code images, retrieve diagnostics, etc. To demonstrate its flexibility, we present two deployments of SWARMS, the first in a wide area weather sensor network, and the second in a local area testbed that was used by a class of graduate students. To demonstrate its scalability, we analyze the performance of SWARMS when the middleware is subject to sensor data loads of thousands of packets per second.**

I. Introduction

Our experiences with deploying a wide area WSN called FireWxNet [1] to monitor weather conditions surrounding wildland fires motivate the need to develop a software middleware system capable of managing disparate far-flung WSNs. Our sensor nodes were deployed in the Bitterroot National Forest of Idaho in summer 2005 across three different mountain peaks, separated by tens of kilometers. Each individual WSN was connected to a backhaul WiFi network. What our deployment lacked was a software system capable of systematically collecting, integrating, and managing the sensor data in one place while also communicating with each of the geographically distinct subnets in our far-flung WSN.

Our belief is that FireWxNet represents the future of many WSNs that will evolve towards more and more widespread deployments. Global sensor networks will be needed to understand planet-wide environmental effects in the land, sea, and air, e.g. to better understand the impact of global warming on our ecology. For all of these scenarios, it is important to develop software systems that scale to meet the needs of wide-area deployments, yet remain flexible to accommodate the desire to tailor sensor networks to application-specific domains.

Prior work in the area of managing WSNs and WiFi networks is largely focused on managing localized groups of sensor nodes, e.g. in indoor testbeds [2], [3] and outdoor testbeds confined to a single field [4], [5]. While the software developed in such systems can scale to hundreds of nodes, it is not designed to manage and integrate data from diverse geographic sources. Issues encountered in wide-area WSNs such as service discovery and addition/subtraction of occasionally connected or disconnected subnets are not addressed.

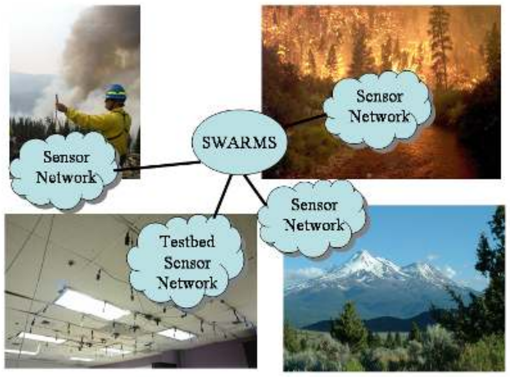


Fig. 1. SWARMS: Wide Area Sensor Network Management

As shown in Figure 1, the SWARMS system is designed to provide an infrastructure to manage *wide area* in situ sensor nodes. WSNs can be distributed in different geographic areas, yet are jointly managed by the same SWARMS software. SWARMS enables users to be able to specify that certain code images be uploaded to different user-defined groups of sensor nodes via either a user-friendly Web interface or a scripting language interface. At the same time, SWARMS logs both data and diagnostics emerging from these multiple WSNs.

Our approach is to provide these basic features while designing the architecture of SWARMS to balance scalability with flexibility. The modularity of SWARMS is key in accomplishing the dual goals of flexibility and scalability. The different aspects of the system (e.g., user interface, job management, programming nodes, communicating with nodes) have been abstracted so as to achieve both scalability and flexibility. The SWARMS system consists of seven components: sensor nodes, software node mates, cluster controllers, head servers, the database, the web server, and the logger. The organization of the system is hierarchically networked and distributed such that individual components can be offloaded to different hosts, enhancing scalability. For example, if the Web server or database are found to be too much of a bottleneck, then they can be easily separated to execute on parallel hosts. In addition, node-mates are designed to be customizable so that application-specific data protocols can be supported in SWARMS. The SWARMS software is so flexible and generic that it can be configured to support two very different yet common WSN scenarios: wide area in situ WSN applications; and local area sensor testbeds.

In the following, Section II discusses related work and how SWARMS contrasts with other wireless middleware and network testbeds. Section III details the design features of the SWARMS architecture. Section IV discusses how the system was implemented in practice. A wide-area application deployment of SWARMS is described in Section V. A local area testbed deployment of SWARMS is described in Section VI, along with an evaluation of the SWARMS testbed's scalability and performance in the face of increased message traffic. The paper finishes with conclusions and future work in Section VII.

II. Related Work

The Motelab [2] WSN testbed manages hundreds of motes and was developed by Harvard to serve as a community resource for WSN researchers. SWARMS differs in a variety of ways. The focus of SWARMS is on supporting wide area sensor networks that span diverse regions, while the focus of Motelab is on a local area testbed. Also, SWARMS is agnostic to the sensor operating system uploaded into the motes, e.g. TinyOS, the multithreaded Mantis OS [6] or the modular SOS [7] are all admissible, while Motelab is tightly integrated with TinyOS. SWARMS borrows some design ideas from Motelab, e.g. a web interface, direct access to sensor nodes and a user based locking mechanism.

Other local area WSN management systems include MistLab and GNOMES. MistLab is a system similar to Motelab that consists of 60 Mica2/Cricket nodes, each connected to central power and ethernet connections. The GNOMES testbed was designed to showcase low-cost hardware/software and explore properties of heterogenous wireless networks [8]. The entire system, from serial/USB programmers to interface, maintains an idea of different kinds of nodes, possibly even virtual nodes.

EmStar [9] takes network heterogeneity in an interesting direction allowing for an integration of a simulation environment with a real-world network testbed. This approach is also called emulation. Its focus is not on wide area management.

Several local area testbeds have explored the notion of scale. The ExScale project [4] and Trio [5] have both deployed on the order of 500 nodes. Each has developed a software middleware system to manage this large scale of sensor nodes. Again, there is no integration of wide area sensor networks into their infrastructure nor an exploration of some of the issues of managing in the wide area.

Simulators have been designed to explore scaling issues in WSNs [10], [11]. However, complex interactions and variability within real-world wireless mediums necessitate real-world testing and verification. SWARMS evaluates its scaling performance in a real-world environment.

The Roofnet project at MIT has developed software to collect and measure real world effects on wireless data transmission [12] in a topology that is not quite wide area but somewhat beyond a single LAN. The management backbone itself is not tested for scalability.

III. Architecture

The architecture for SWARMS is designed to provide basic programming and diagnostic functionality while achieving scalability, flexibility, and extensibility. The core of the middleware system inhabits the dashed outline in Figure 2, and exists between the sensor nodes and the clients receiving the data. Clients at the top use SWARMS for programming, controlling and communicating with the sensor nodes at the bottom, which we term gateway sensor nodes. This involves transferring new images to the gateway sensor nodes and routing messages through SWARMS passed from nodes to the connected clients and vice versa. Broadly speaking, in tier 1 Node Mates control a gateway sensor node; in tier 2 multiple node mates funnel information back and forth between a Cluster Controller; and similarly in tier 3 multiple cluster controllers funnel information back and forth to the Head Server, which controls the entire system.

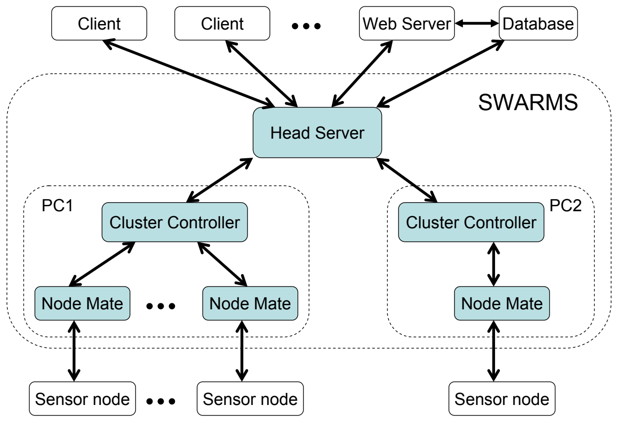


Fig. 2. Swarms Architecture

The three-tiered hierarchy of SWARMS is scalable, flexible, and extensible in the following ways. The head server, database, and Web server can all be distributed to execute on separate computers. This scaling feature was found to be especially useful during the initial performance evaluation of the system, when the database began to bog down the entire system when run on the same machine as the Web server. Because of SWARMS' modular design, offloading this process to another PC was a simple procedure, and effectively eliminated this performance bottleneck.

Another scaling feature arises from the design of the cluster controller. One cluster controller Daemon runs on any computer that has sensor nodes attached to it. A "cluster" is then defined as any computer that has nodes attached to it, the cluster controller daemon process, and the nodes managed by the cluster controller. This architecture supports scaling in that if a particular cluster-controller PC becomes overloaded, more computers can be added to function as additional clusters, with some of the sensor nodes then physically offloaded for management to the new clusters. For example, this feature was found to be quite useful when handling scaling issues while building out a testbed variant of SWARMS. After experiencing a bottleneck in USB bandwidth when using only one PC to manage the testbed, SWARMS made it straightforward to add a new PC and its associated cluster controller, which was then easily configured to take over management of about half of the sensor nodes.

Perhaps the most prominent feature of SWARMS is its wide-ranging flexibility to remotely manage both sparse wide-area WSNs, e.g. sensornets distributed across a city or even the globe, and dense local-area WSNs, e.g. testbeds within a room in a lab. Conceptually, the two seemingly different paradigms are handled in a similar manner within the SWARMS framework. In the simplest case, SWARMS maintains a one-to-one correspondence between a node mate and a hardware sensor node. In this case, to realize a wide-area WSN, SWARMS just maintains a one-to-one correspondence between a cluster controller and a node, i.e. each PC cluster controller manages only one sensor node. Each sensor node can be placed anywhere in the world where there is a PC to run a cluster controller. It is thus simple to build a wide-area WSN. Each new user can add a new node to an existing wide-area sensor network managed by SWARMS by starting one local node mate process and one local cluster controller process, which only needs to be told where to connect to the appropriate head server.

To realize a local-area WSN testbed, SWARMS generalizes to inherently support a many-to-one correspondence between each cluster controller and many sensor nodes. Instead of the special case of managing just one node as in the wide-area application, each PC cluster controller in the testbed is capable of managing many sensor nodes. These nodes can all be placed within a single room to form a dense local area testbed.

In this manner, the same SWARMS software architecture is capable of flexibly managing both local area WSN testbeds as well as wide area WSNs. In this paper, we present two application deployments of SWARMS corresponding to separate wide area and local area realizations. However, the SWARMS framework accommodates additional flexibility: a single SWARMS server can simultaneously manage a hybrid combination of dense and sparse WSNs; further, the mapping between node mate and sensor node can be relaxed to accommodate many-to-one mappings between a node mate and sensor nodes, i.e. to support multi-hop WSNs that connect through a single node mate to a cluster controller. We elaborate on these possibilities in future work.

SWARMS' flexibility is enhanced by supporting a disconnected mode. In many practical deployments, nodes may not be persistently connected to the remote middleware managing the network. As a result, SWARMS' design allows disconnection between the local software, e.g. the cluster controller, and the remote software, e.g. the head server. Upon observing that a connection has broken with the head server, the local cluster controller drops into disconnected mode and begins caching sensor data. After observing that a connection has been reestablished with the head server, all the cached data is flushed back to the head server. This feature came in handy during deployment of the wide-area SWARMS application, when the connection between one sensor node at a home in the mountains and the head server at the university would break intermittently over a poor-connectivity modem line. The local cluster controller appropriately cached sensor data when the modem line was down, and reported it so that no data was lost when the line was back up.

Extensibility in SWARMS is supported in at least two ways. First, the node mates themselves are designed to be customizable, so that application-specific filters can be placed into node mates to translate incoming data to an appropriate application-specific format. SWARMS is agnostic to what format this data takes, and simply routes the data to the endpoint, whether that is the database or the client. Second, a goal of this architecture was to be agnostic to the underlying node hardware and software thus making the system extensible to new nodes. Node-mates can be configured to accommodate new hardware as new sensor nodes are introduced.

A. Node Identification

Before discussing the details of how data and control messages pass through the system, the identification of nodes will be explained. Aside from the processes depicted in Figure 2 exists a database containing all the meta-data required for identifying each node. There is a node table that contains information like the type of node, the cluster controller it is connected to and a unique ID used internally to identify each node. This information is required to uniquely identify where a message originated if it is coming from a node or where it is destined if it is going to the node. This information also helps identify which nodes are expected to be connected to SWARMS at any time. This database also contains information regarding grouping of nodes and assignments between program images and these defined groups. It is this grouping structure that allows us to program many gateway sensor nodes at once with the same image, repeatedly with ease.

B. Types of Communication

In the sections below, two types of communication are identified which are treated differently in the SWARMS system. The system separates the communication into a control plane and a data plane. This allows each flow of data to be handled separately as their use is quite different in functionality.

1) Control Messages: Control messages are messages normally initiated from an end user to perform a particular action on a node or group of nodes. These actions include programming a new image, starting and stopping execution. These messages use an RPC mechanism known as Distributed Ruby Objects (or DRb). These messages are instantiated strictly by the client and pass down to the node mate. These messages do not get passed onto each node and never travel in the other direction.

2) Data Messages: Data messages are bi-directional in that they can either be generated by an attached node or by a client. The SWARMS system encapsulates the data in a generic packet and routes it accordingly. If the message is from a node, it is broadcast to each client by the head server. If the message is destined for a node, it is routed by SWARMS to the appropriate node mate. The benefit of this encapsulation process is that it allows a custom protocol to be built on top of SWARMS with little modification required to the internals of the system. Additionally, clients are able to have live interaction with the nodes through this path.

C. Node Mate

The node mate has several responsibilities including programming the node, controlling the node (starting / stopping the running execution) and translating messages between the gateway sensor node and SWARMS, each of which is explained below. The node mate is a process that consists of parts that are specific to each node hardware/software. As such, only the node mate knows how to get an image onto the node or what packet format a node will be sending messages in. There is exactly one node mate process for each node connected to a computer.

The largest factor affecting the extensibility of this system is the ability to add a new type of node. As such, this part of the system was designed to be as modular as possible with the different responsibilities clearly separated. A node mate is then comprised of a generic layer and several specific layers depicted in Figure 3.

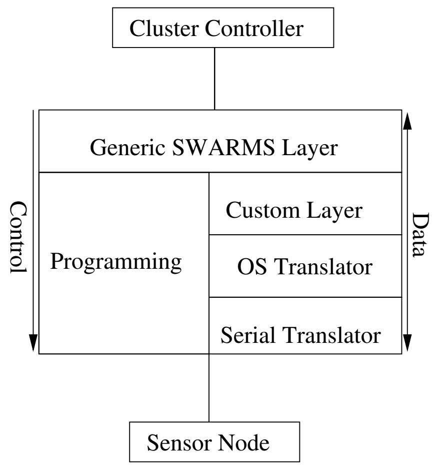


Fig. 3. Node mate Architecture

The Generic SWARMS Layer is responsible for establishing a control and data connection to the cluster controller. Next control messages are passed to the programming interfaces upon request by the cluster controller. Likewise, data messages are passed through the stack on the right side. The Generic SWARMS Layer takes SWARMS packets from the cluster controller and passes them down the stack to the node. Additionally, this layer takes discrete packets, adds the node's id, creating a SWARMS packet and passes it on to the cluster controller.

Below the Generic SWARMS Layer in Figure 3 are two sections that are truly specific to the particular node. The left side provides the control interface to the node while the right side provides the data interface. The programming interface is specific to the given hardware of a node. This layer provides a function to take an image that resides on the local computer and program the node with that image. It is additionally responsible for stopping and restarting the currently running program. This piece is currently implemented for the MICA2/MICAz and TelosB nodes.

The Serial Translator listens on a given serial port and provides a byte stream interface to the OS Translator. The serial port is passed down as control information from the cluster controller and stored by the Generic SWARMS Layer.

The OS specific portion of the packet translation is provided by the OS Translator layer. This layer takes discrete packets from the Generic SWARMS Layer and converts them into OS specific packets and sends them to the Serial Translator. Additionally, it reads the byte stream provided by the Serial Translator and passes discrete packets to the Generic SWARMS Layer.

It is also easy to see how a custom layer could be inserted into this design to perform additional manipulation on the packets before being sent to the cluster controller. In addition to traffic shaping mechanisms, additional operations such as compression / encryption could be performed at this layer.

D. Cluster Controller

The cluster controller is a process that manages the many node mate processes on a given computer, saves images to the local drive, and acts as a conduit for the node mate and head server to exchange data. As such, there is only one cluster controller process on each computer participating in SWARMS with nodes connected. One or more cluster controllers can connect to the head server. This organization allows us to distribute the cluster controllers and node mates across different computers that are an arbitrary distance apart, as shown in Figure 2. As a consequence, the cluster controllers can exist across the Internet, thus giving this design flexibility for managing a wide area distribution of sensor nodes. When a user selects to program an image that they've uploaded, the head server transfers the image to each cluster controller, then sends a control message notifying each cluster controller of the job to be started. The cluster controller then notifies each node mate (spawning new node mate processes as necessary) to program the given image.

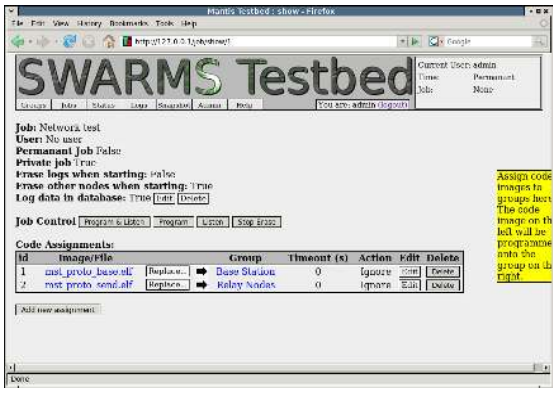
The other main responsibility of the cluster controller is routing messages between each node mate and the head server. As such, each cluster controller must be configured with the appropriate head server to connect and transfer messages with. It opens a TCP/IP connection to the head server. Then, each message sent from the head server is inspected and sent to the appropriate node mate process. Any message sent from the node mate is transferred to the head server over this connection as well.

E. Head Server

The head server process is the central point of access for external clients to the SWARMS system. When the head server receives messages from a cluster controller, it distributes that message to each client connected. Likewise, when the head server receives a message from a client, it inspects the message, performs a look-up in the meta-data to determine the appropriate cluster controller and routes the message appropriately. Control messages are passed onto the cluster controllers in a similar fashion.

F. Clients

Clients, as mentioned in previous sections, are simply applications that wish to control nodes, communicate with nodes, or both. A DRb connection is made with the head server for control messages while a TCP/IP connection is established for node communication. When a client wishes to send data to a node, it encapsulates that data in a specific structure that contains that node's unique id as established in the SWARMS meta-data. Next, it sends this data to the head server to be routed to the node. As a result, the client only needs to know the unique id given to the node(s) it wishes to communicate with and a TCP/IP connection. This property allows custom clients to be created to handle the actual data sent from the node. This also allows the client to format data destined for the node in any fashion possible and the system will route the packet regardless of the underlying data.


Mantis Testbed : show - Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1/job/show/1
SWARMS Testbed
Groups Jobs Status Logs Snapshot Admin Help
You are: admin (logout)
Current User: admin
Time: Permanent
Job: None

Job: Network test
User: No user
Permanant Job False
Private job True
Erase logs when starting: False
Erase other nodes when starting: True
Log data in database: True Edit Delete

Job Control Program & Listen Program Listen Stop Erase

Code Assignments:

Id	Image/File			Group	Timeout (s)	Action	Edit	Delete
1	mst_proto_base.elf	Replace...	➡	Base Station	0	Ignore	Edit	Delete
2	mst_proto_send.elf	Replace...	➡	Relay Nodes	0	Ignore	Edit	Delete

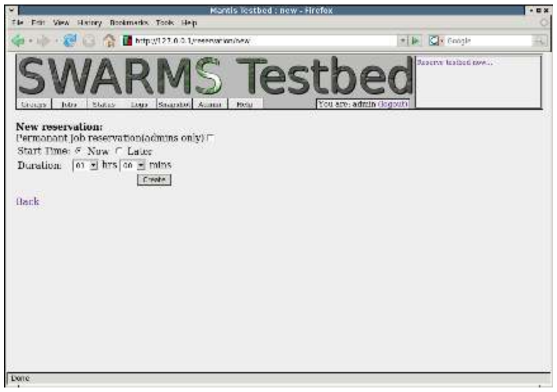
Add new assignment

Assign code images to groups here. The code image on the left will be programmed onto the group on the right.

Done

Fig. 4. Remote Job Management Interface

1) Web Server: One unique client that is included with the SWARMS system is the Web Server. This client provides an interface for remotely managing WSNs, i.e. reserving SWARMS, creating groups, job assignments (Figure 4), managing groups, and even reserving time on the testbed (Figure 5). Additionally, the Web server provides access to a database of logged data from the nodes (Figure 6). As such, it is possible to use SWARMS without any specific software installed on a user's computer besides a Web browser.


Mantis Testbed : new - Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1/reservation/new
SWARMS Testbed
Groups Jobs Status Logs Snapshot Admin Help
You are: admin (logout)
Reserve testbed now...

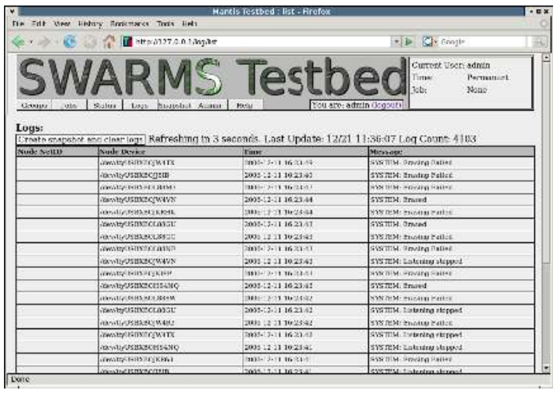
New reservation:
Permanent job reservation(admins only) ☐
Start Time: [✓ Now] Later
Duration: 01 hrs 00 mins
Create

Back

Done

Fig. 5. Interface for testbed reservations - resource management

2) Database Logger: Another client included in the SWARMS system is a Database Logger. This (which may be selectively enabled for any given job) simply stores each message from all nodes in a database for later retrieval and inspection. This provides persistence of data regardless of whether or not an external client is connected. Also, since the data resides in a database on the server, it can quickly be indexed and managed through the Web interface.


Mantis Testbed : list - Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1/log/list
SWARMS Testbed
Groups Jobs Status Logs Snapshot Admin Help
You are: admin (logout)
Current User: admin
Time: Permanent
Job: None

Logs:
Create snapshot and clear logs Refreshing in 3 seconds. Last Update: 12/21 11:36:07 Log Count: 4183

Node NetID	Node Device	Time	Message
	/dev/ttyUSBXCQW4TX	2005-12-11 16:23:4?	SYSTEM: Erasing Failed
	/dev/ttyUSBXCQJ5IB	2005-12-11 16:23:45	SYSTEM: Erasing Failed
	[illegible]	[illegible]	SYSTEM: Erasing Failed
	/dev/ttyUSBXCQW4VN	2005-12-11 16:23:44	SYSTEM: Erased
	[illegible]	2005-12-11 16:23:44	SYSTEM: Erasing Failed
	/dev/ttyUSBXCOL8GU	2005-12-11 16:23:43	SYSTEM: Erased
	/dev/ttyUSBXCOL8GU	2005-12-11 16:23:43	SYSTEM: Erasing Failed
	[illegible]	2005-12-11 16:23:43	SYSTEM: Erasing Failed
	/dev/ttyUSBXCQW4VN	2005-12-11 16:23:43	SYSTEM: Listening stopped
	[illegible]	[illegible]	SYSTEM: Erasing Failed
	/dev/ttyUSBXCHS4NQ	2005-12-11 16:23:43	SYSTEM: Erased
	[illegible]	2005-12-11 16:23:42	SYSTEM: Erasing Failed
	/dev/ttyUSBXCOL8GU	2005-12-11 16:23:42	SYSTEM: Listening stopped
	/dev/ttyUSBXCQW4B2	2005-12-11 16:23:42	SYSTEM: Erasing Failed
	/dev/ttyUSBXCQW4TX	2005-12-11 16:23:42	SYSTEM: Listening stopped
	/dev/ttyUSBXCHS4NQ	2005-12-11 16:23:41	SYSTEM: Listening stopped
	[illegible]	2005-12-11 16:23:41	SYSTEM: Erasing Failed
	/dev/ttyUSBXCQJ5IB	2005-12-11 16:23:41	SYSTEM: Listening stopped

Done

Fig. 6. Summary of log messages

Due to the fully distributed design of SWARMS, both the Web server and database need not be co-located with the head server. Either or both software entities can execute at remote locations and connect via TCP/IP and DRb, which is useful for scalability. In our implementations, we had full flexibility in choosing where to locate and set up the head server, the database, Web server, and cluster controllers.

IV. IMPLEMENTATION

The programming language used throughout the system was Ruby. This was chosen for its cross platform support and ease of maintainability. An end-user of the system may either employ the Web interface or develop their own custom client. A testbed administrator may need to develop software for a new node mate. Using ruby allows us to minimize the amount of code required to do this, thus simplifying the task.

The DRb library was leveraged to provide the control plane between the clients and the nodes. This RPC mechanism nicely matched our goals, simplifying the separation between data and control planes.

Access to the database is provided by Active Record, a part of the Web framework called Ruby On Rails. This provides a generic interface which has many database backends. As a result, the system can work with PostgreSQL, MySQL and possibly others as the database to store the meta-data about the testbed. We have implemented versions of SWARMS using either PostgresSQL or MySQL.

The Web interface is constructed based on the Ruby On Rails framework. SWARMS' Web interface provides cross-platform support for uploading and controlling the nodes, alleviating the client software from having to provide these functions as well. Leveraging Ruby On Rails provided a convenient framework to build this user interface. Snapshots of the Web interface can be referenced in Figures 4, 5, and 6.

The specific procedure for accessing a SWARMS-controlled sensor network through the Web is straightforward. A user first logs in, or establishes an account, by clicking on the appropriate button in the SWARMS Web page. After logging in, the user may reserve the set of nodes that is being managed for a specific day and specific hours, as shown in Figure 5. SWARMS enforces that only one user at a time can be occupying the set of nodes. Next, the user can define a new job by going to the Jobs menu, as shown in Figure 4. The user can separately define groups and assign subsets of nodes to each group by clicking on the Groups menu. Then, within a job, the user can define an image and assign this image to be uploaded to a given group of sensor nodes. The normal method for starting the job is "Program and Start" in the Job menu, which programs the nodes, and then starts all of the nodes when they are finished programming. The total time to program and start all of the nodes is usually about 1 min, with the nodes starting in about the last 2 seconds of that time. To obtain diagnostic data while the job is running, clicking on the Status menu of the Web page gives a near real-time readout of the last 30 messages received from the collective set of nodes, as shown in Figure 6. All the data being generated by the sensor nodes and sent back as packets through SWARMS is logged in the database. Once the user is done with the job, a snapshot from the database can be obtained by clicking on the menu below the "Snapshot" tab, which will download all of the log messages as a CSV file for easy post processing.

The current clients are implemented in ruby, thus giving them access to the control plane as well as the data plane. The clients have a Generic SWARMS Layer as well. This allows clients to quickly be built with only a few lines of code. All of the details of connecting to the cluster controller and establishing the two planes are provided by this generic layer. Using ruby, *scripts* can easily be created to interact with SWARMS. This includes controlling groups, nodes, uploading images and talking directly to the node. The scripting interface provides a more powerful mechanism than the Web interface for clients to specify complex jobs that require the definition of many groups, the uploading of many images, and the ordered starting and stopping of various images.

V. A Wide Area Deployment Of SWARMS

To demonstrate the capability of SWARMS to manage a wide area WSN deployment, we constructed a simple outdoor weather sensor network that spanned about 10 miles of Boulder, Colorado. Five nodes were deployed around the city of Boulder, with one at the university, three in the town of Boulder, and one a few miles outside of Boulder. The TelosB nodes were put just outside windows, and were not weather sealed in any form. The temperature, humidity, and light were reported every minute to the central SWARMS head server at CU, with the temperature and dewpoint being calculated and binned over 1 hour periods. The layout of the nodes is shown in Figure 7.

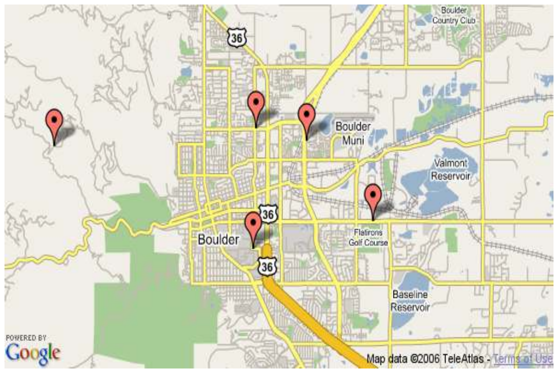


Fig. 7. Node locations in our real world wide area deployment of SWARMS.

The wide area sensing was up and running during a blizzard that hit the Denver area after Christmas in 2006. The dewpoint and temperature are shown in Figure 8, with a normal December day included before the storm to show the contrast, where the front moving in and rain starting at noon on the second day in the graph. With SWARMS, we were able to centrally monitor temp/RH data from a widely scattered collection of sensor nodes, and thus capture the passage of the blizzard, as evidenced in particular by the dew points (lower lines) that changed abruptly around noon on December 27.

During the deployment, we took advantage of the on-line logging feature of SWARMS to monitor and troubleshoot the health of the network. We found a variety of failures during this wide area deployment. Several of the nodes had hardware problems such as shorting out, freezing, and getting wet, which necesitated a restart of the node. Another problem with the sensor nodes is that the memory tends to get corrupted below $-15^{\circ C}$, which is different from the node crashing in that it is difficult to automatically detect a fault. Though disconnected mode was enabled to handle network link failures between the sensor node and the SWARMS head server, we did not experience this type of failure during this deployment. As reported, we have utilized SWARMS' caching recovery mechanism in past deployments.

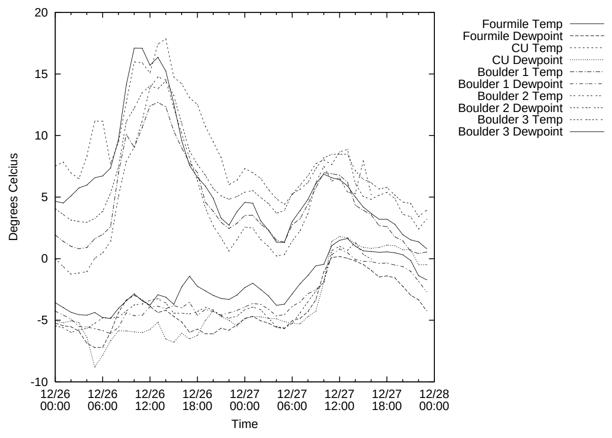


Fig. 8. SWARMS in wide area mode: temperature (upper lines) and dewpoint (lower lines) over time at 5 locations

In order to restart a node, i.e. restart the node mate process corresponding to a node and then reprogram that node, we needed remote access to the PC executing the local SWARMS software. In some cases, the PC was hidden behind a firewall, so the home user had to provide explicit secure login access for this port. This brings up an interesting issue with respect to security and ease of use. Our wide area implementation of SWARMS sought to achieve ease of use, e.g. after attaching a node to a PC, a user ran an installation script that only had to start one node mate process and one cluster controller, which then reached outbound through the home firewall to connect to the SWARMS server to begin streaming data. However, due to the failures noted above, we could not avoid involving the home user in some further configuration, e.g. opening up port 22 on their home firewall for ssh login to assist in fault recovery. While this simplified fault recovery and remote management, this complicated ease-of-use from the user's perspective and also required the home user to trust the SWARMS manager. At present, we are still considering where the right balance lies between ease of use for the sensor node publisher and ease of use for the remote sensor net manager.

VI. A Testbed Deployment and Performance Evaluation

A. A Local Area SWARMS Testbed

In addition to a wide area application, we have deployed SWARMS to control a local testbed of 50 TELOS-B sensor nodes confined to a single room in our lab. This is shown in Figure 9. The testbed nodes are split between two cluster controller PC's. The head server and database reside on a third PC, while the Web server operates on a fourth PC. A Webcam is also trained on the SWARMS testbed for visual feedback and monitoring.

We have effectively used this SWARMS testbed both as a research tool and a teaching tool. For example, SWARMS was used to support a programming project during fall 2006 in a graduate seminar on sensor networks. Just as in the wide area case, user clients, in this case students, were able to define

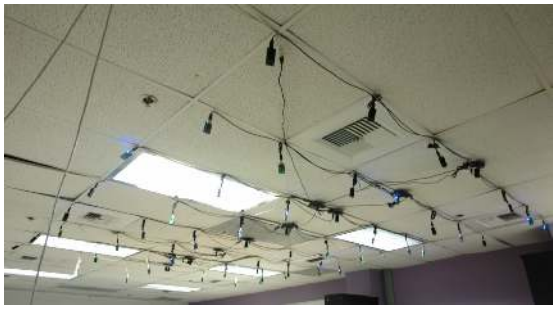


Fig. 9. SWARMS in local mode: controlling a large number of motes in one location

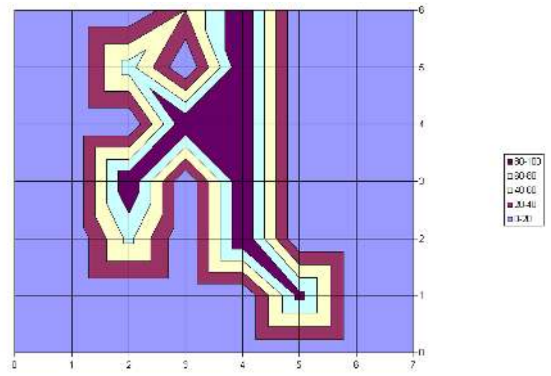


Fig. 10. Packet reception statistics from a single transmitter to multiple receivers on the SWARMS testbed. Students built their own application, programmed it into the testbed, and collected statistics via SWARMS.

jobs, upload binary program images onto the testbed, collect diagnostics, etc. SWARMS enabled the class to implement a programming assignment to collect and characterize real-world statistics of wireless radio behavior on sensor motes. An example of packet reception statistics collected when one of the students programmed the testbed with SWARMS is shown in Figure 10. The transmitter power setting is set to the lowest level on the TELOSB motes. The student programmed one node as a transmitter and the other nodes as receivers, and then streamed both packet loss and received signal strength measurements from each mote to the SWARMS head server. In the past, the SWARMS testbed has been used in our research to evaluate the efficacy of MAC and routing protocols.

B. Performance Evaluation

We evaluated the scaling properties of our SWARMS design on the testbed implementation, i.e. up to what rate of input data can be processed by the SWARMS system. Input data can refer to sensor data, debug info, or any other data sent from the nodes over serial/USB connections into the SWARMS logging system. Understanding the limits of our design is important if we expect this system to scale across the Internet, and support distributed sensor networks and testbeds. We must be sure that the overhead between abstraction layers is not too large. We measured three different metrics to understand system performance. These metrics are:

- **Output Packets Received**: The rate of packets processed and routed through the SWARMS system and stored by the database logger.

- **Percent Lost Packets**: The percentage of packets lost during test runs.
- **Latency Incurred**: The latency incurred while passing through the SWARMS system.

To test SWARMS, we wrote a performance client that attached to SWARMS just as any other software application client would attach, namely at the Head Server. The client implemented a simple ruby script that would send commands down to the sensor nodes, and then measure the throughput of reported sensor data routed through the SWARMS system as well as the roundtrip latency from the client to a sensor node and back to the client. The script was a performance testing suite that cycled through a series of variable settings, as shown in the table below:

Variables used	
Database:	On, Off
Packet Size (bytes):	1, 2, 4, 8, 16, 32, 64
Delay Between Packets (ms):	1, 3, 5, 7, 8, 9, 10, 25, 50, 100, 200
Samples per data point:	10 samples
Time per data point:	2 second

For each of the input variables, (database state, packet size, and delay between packets), the state is set, and a command is sent to each sensor node to generate packets of size and rate indicated. These data packets are routed from the node through the SWARMS infrastructure to the client, and may/may not include logging at the database. The performance suite consists of multiple nested loops, i.e. once the packet size is set, the script will cycle through each of the delay settings before incrementing the packet size and cycling again through the delay settings, etc. Since the data load from the nodes goes from light to heavy several times, the recovery of the system is also tested. The performance suite is an example of a script that sends commands to the testbed and the nodes, and then receives the output from the nodes.

On the nodes, we instrumented the software in a simple manner to generate different rates and sizes. The rate and size are configured by input over the serial line, so that they don't have to be reprogrammed for every different setting. Also, a special packet can be sent to the node that causes an instant "ACK" packet to be sent, for latency measurements.

The evaluation was run on ten of the Telos-B sensor nodes, each controlled by a separate node mate (i.e. each had an abstraction layer between them and the cluster controller). These ten nodes were controlled by a single cluster controller on one computer. The rest of the system follows the architecture of section III. In particular, the head server executes on a separate computer. The MySQL database is on the same computer as the head server. The web server is on a separate computer as well, but that isn't used during the execution of the performance testing. While we are testing different data rates, we assume that the data is constant for the duration of each test run. The computers are communicating over a wired link that experiences significant traffic at all times. We investigate the performance of the system with caching turned off to understand where SWARMS begins to overload and lose packets.

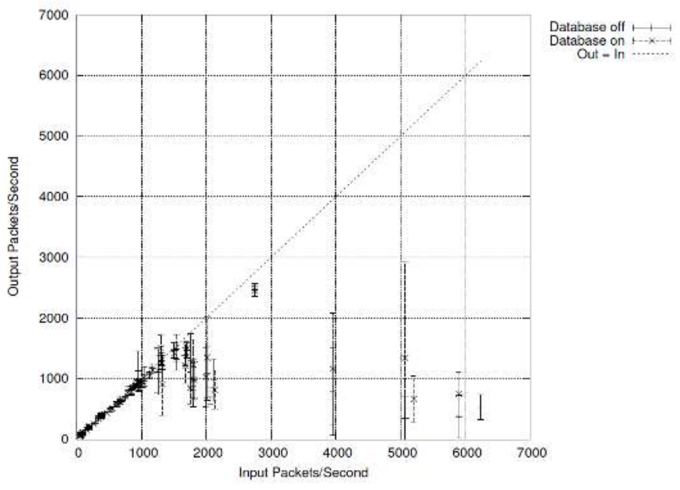


Fig. 11. Output packets/second received relative to input packets/second

1) Success Rate: Figure 11 shows the rate of output packets successfully received by the logger, relative to the rate of input packets sent into SWARMS over the serial interfaces. The output rate largely follows the input rate along the expected y=x equality until reaching a "saturation" point of about 1000 input packets/sec. This is true across the range of different packet sizes that we tested, which were chosen to resemble the small data packets typical of periodic sensor sampling. At least in this range, packet size did not affect the output rate, nor did the presence or absence of the database, which we had expected would be a time-intensive operation that would slow down SWARMS.

After the saturation point of 1000 input packets/sec, the average roughly plateaus, deviating substantially from the y=x line, while the variance starts to increase dramatically. These results imply that SWARMS begins to experience substantial performance degradation while trying to service more than 1000 packets per second. After degradation begins, SWARMS attempts a best-effort transport of all messages. This added feature extended the performance of the system, avoiding a complete meltdown up to a level of more than 3000 packets per second, or 3 times the saturation point. However, at this point, the figure shows that SWARMS begins to lose more than 50% of all input packets past 3000 packets/sec.

The results give us some reason to hope that SWARMS would be capable of handling fairly large sensor networks, perhaps up to hundreds of thousands of sensor nodes. This is because many sensor networks generate packet samples on the order of once per minute, or even once per hour. For example, a wide area sensor network consisting of 100K nodes, each generating a packet every 100 seconds (about 2 minutes),

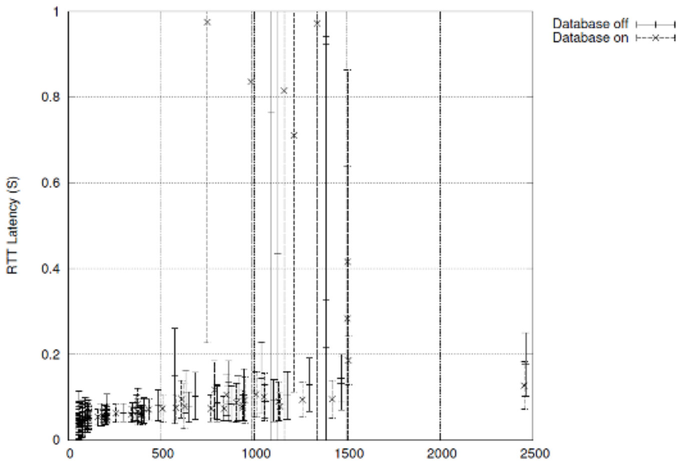


Fig. 12. Latency relative to input packets/second

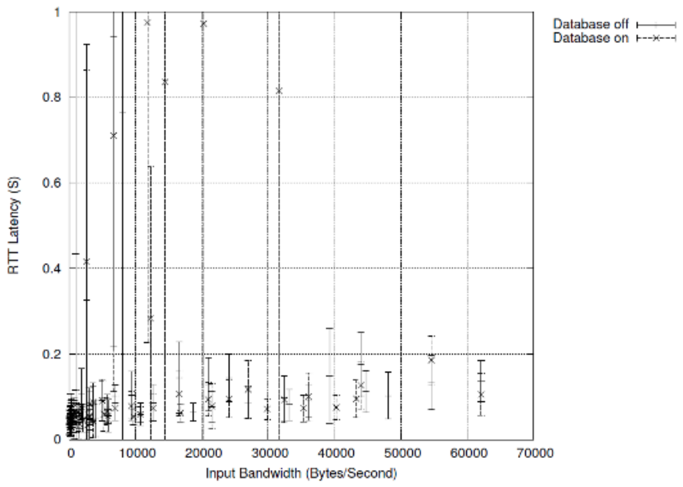


Fig. 13. Latency relative to insertion bandwidth

would in aggregate generate only 1000 packets/sec.

2) Performance Degradation: Latency: Figure 12 shows how packet latency is affected by congestion due to increasing message traffic. Assuming, message insertion rates are less than 100 packets/second, a roundtrip latency of 40 ms is incurred. However, above 200 packets/second, latency increases to 60 ms. Overall, latency rises gradually and is well-behaved below the 1000 packets/sec saturation point, not exceeding on average of about 100 ms. This is a fairly small amount of delay to be adding for most sensing applications. Recall also that this is the roundtrip delay, and SWARMS is actually adding only the one-way delay from the moment the sensor is sampled on the node to the moment it is delivered to the client. We measured the roundtrip, which only relied on the client's clock, because it was easier to capture than the one-way delay, which would require time synchronization with the sensor node's clock and time stamping mechanism.

Above 1000 input packets/sec, performance degrades noticeably at high rates, as SWARMS becomes bogged down with processing and starts to lose many packets. With insertion rates greater than 1000 messages/second, latency tends to vary greatly.

Finally, an understanding of what bandwidth is supported was also tested. This evaluation is critical to understanding whether a distributed design approach is plausible. Figure 13 shows latency measured relative to data insertion bandwidth, helping us to understand whether it is the number of packets or the overall bandwidth that is causing the congestion. It should be noted that this data is traveling over a single network, in our case it was a single 100BaseT link. While results varied somewhat, the contrast between the general steady trend of this graph and the plateauing behavior of figure 11 implies that pure bandwidth does not play as big of a role in system congestion as the number of packets does. This result suggests that the bottleneck is in software processing by the different components in SWARMS as opposed to the bandwidth of traffic on the Internet. Our future work is focused on understanding these software bottlenecks more clearly, though we have already made substantial progress in identifying and removing such bottlenecks thanks to the performance test suite, as explained below.

3) Implications: This evaluation has shown that the system can store packets with high success and good performance up until a certain threshold. We also have an understanding of what that threshold is and why it exists. Degradation begins slightly at a rate of 100 packets/second and becomes significant beyond 1000 packets/second. This performance degradation is primarily due to the number of packets and not the pure bandwidth. It is also comforting to know that degradation is relatively graceful, allowing for certain types of analysis e.g., data traces, statistical analysis to remain valid, even if data streams create flash events that overwhelm the network. A version of SWARMS is being developed that doesn't drop packets, which will significantly affect the performance under a significant load.

Ensuring that the performance client executed properly under varying workloads was a challenge, as quite a few parts of the original SWARMS system worked reasonably well under light loads, but crashed under heavy loads when we stressed the system. The commands to the nodes had to go through the system correctly, which exposed a few concurrency problems with reading and writing to the serial device that we fixed. Beyond that, quite a few changes were made to reduce the packet loss, and to aid in the recovery of the system after the nodes stop sending large number of packets. The resulting system is what was evaluated in the figures above. We were thus able validate the complete system for SWARMS in terms of its scaling performance and capabilities.

VII. Conclusions and Future Work

Remote wide area management of in situ sensor networks is becoming increasingly important. The SWARMS management system is designed and built to meet this need with scalability, flexibility, and extensibility. SWARMS has achieved these goals by adhering to a set of principles, namely: clean abstraction modularity, generalized communication between modules, and an easy to use and easy to access *flexible* interface. The following summarizes specific achievements of SWARMS.

- SWARMS achieves scalability through a modular design. An experimental evaluation demonstrated that SWARMS can currently scale up to 1000 input packets/sec with little packet loss and low latency
- SWARMS achieves flexibility through a modular design. We have shown through two separate implementations that the same SWARMS software can be configured to manage a wide area sensor net or a concentrated local area testbed of 50 sensor motes. The SWARMS architecture is general enough to accommodate any combination of sparse and dense sensor networks.
- SWARMS can program many nodes in parallel with relative ease. This is achieved through the use of distributed cluster controllers running on many computers.
- Programming of nodes is selective. The concept of "groups" allows for any combination of nodes being programmed in parallel to any other independent set of nodes. The concept of "job" flexibly ties groups and code images together, giving the user substantial freedom in time and space to choose how to program a set of nodes.
- The "node-mate" abstraction allows for customization and coexistence of heterogeneous data protocols.
- The high-level system interface(s) provide flexible remote access for developers. An easy-to-use web interface is complemented by the scripting DRb or TCP client interfaces, which can be used in parallel.
- The concept of "node" has been generalized to support any type of node connected to a node mate. This allows for heterogeneity in future systems. New node mates would have to be written and then added to SWARMS, just like loading a new driver.

A variety of objectives remain to be accomplished in SWARMS. First, our tests of scalability are incomplete. We would like to test the performance limits of SWARMS when nodes are more widely spread than they are presently situated. Second, SWARMS also should support a greater diversity of sensor hardware through the addition of new node-mates. Third, we would like to relax the assumption of our current two application deployments that there is a one-to-one correspondence between a node mate and a sensor node. Instead, we'd like to demonstrate that SWARMS easily supports a many-to-one mapping between a multi-hop network of sensor nodes masked behind a single gateway sensor node and a single node mate, with the sensor node that the nodemate is connected to acting as the relay, using the sensor node as a bridge from wired to radio communications. This would allow, for example, SWARMS to manage a wide area WSN consisting of many one-node WSNs (one-to-one mapping with a node mate) combined with a few other highly populated WSNs that have hundreds of nodes (many-to-one). This flexibility would be useful for example in FireWxNet to manage individual isolated sensor nodes across remote mountain tops, e.g. nodes far from wildfires, while simultaneously managing large groups of sensor nodes densely deployed near regions of interest, e.g. near active wildfires. This is precisely the scenario that we experienced in FireWxNet, where two mountains contained multi-node WSNs, while the third mountain consisted of a single-node WSN. Fourth, regarding security, we have incorporated only limited access control into SWARMS. We hope to provide improved isolation of user data so one user cannot view another user's diagnostic and sensor data without permission. Finally, we plan to release SWARMS in open source in the near future.

References

[1] C. Hartung, R. Han, C. Seielstad, and S. Holbrook, "Firewxnet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments," in *MobiSys 2006: Proceedings of the 4th international conference on Mobile systems, applications and services*. New York, NY, USA: ACM Press, 2006, pp. 28–41.

[2] G. Werner-Allen, P. Swieskowski, and M. Welsh, "Motelab: A wireless sensor network testbed," in *The Fourth International Conference on Information Processing in Sensor Networks (IPSN 2005), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, April 2005, pp. 73–78.

[3] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, "Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols," in *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, 2005.

[4] A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridharan, S. Kumar, N. Seddon, and e. a. Chris Anderson, "ExScal: Elements of an extreme scale wireless sensor network," in *The 11th IEEE International Conference on Embedded and REal-Time Computing Systems and Applications, (RTCSA'05)*, Hong Kong, Hong Kong SAR, August 2005.

[5] 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 *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*. New York, NY, USA: ACM Press, 2006, pp. 407–415.

[6] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms," vol. 10, no. 4, August 2005, pp. 563–579.

[7] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM Press, 2005, pp. 163–176.

[8] E. Welsh, W. Fish, and P. Frantz, "GNOMES: A testbed for low-power heterogeneous wireless sensor networks," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, Bangkok, Thailand, May 2003.

[9] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer, "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, 2004, pp. 201–213.

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

[11] R. Barr, Z. J. Haas, and R. van Renesse, *Scalable Wireless Ad hoc Network Simulation*. CRC Press, 2005, ch. 19, pp. 297–311.

[12] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris, "Link-level measurements from an 802.11b mesh network," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 121–132, 2004.