# A File System Abstraction and Shell Interface for a Wireless Sensor Network Testbed

Andrew R. Dalton, Jason O. Hallstrom
School of Computing
Clemson University
{adalton, jasonoh}@cs.clemson.edu

*Abstract*— **Despite tremendous research interest and increased adoption, deeply embedded sensor networks are difficult to design, debug, and deploy; ultra-dependability remains an elusive goal. To address these difficulties, we have previously presented an *interactive*, *server-centric* testbed for wireless sensor networks that targets systems constructed using *nesC* and *TinyOS* — the emerging standard in sensor system development. The testbed infrastructure exposes an API suite that enables users to rapidly configure, instrument, compile, install, and profile their systems on one or more remote network deployments. The prototype deployment consists of 80 *Tmote Sky* devices arranged in a regular grid. The architecture is extensible in both the hardware and software dimensions to foster adoption and specialization.**

**In this paper, we demonstrate the extensibility of the testbed software design, and present a novel file system abstraction and shell interface developed using the original API suite. The design of the new interface is informed by user feedback from client institutions where the standard graphical interface is being used to support research and teaching activities. The new shell interface complements the traditional graphical interface, reducing interaction latency, and enabling programmatic experimentation through an interpreted scripting facility. We present the design and implementation of the new testbed interface, and present a small, but representative case-study that illustrates its utility.**

## I. Introduction

Wireless sensor networks are emerging as a linchpin in the foundation of the ubiquitous computing vision — networked computing devices integrated transparently with the world around us. The devices that form the lowest tiers of these networks are referred to as "*motes*" [1], and are responsible for sensing, processing, and communicating environmental phenomenon (*e.g.*, light, motion, sound). The *mote* moniker reflects their increasingly small form-factors, which have progressed from the size of a matchbox [2], to the size of a quarter [3], to the size of a ballpoint pen tip [4]. Their small size, low-cost, and wire-free operation make it possible to deploy mote networks in a range of contexts, both indoors and outdoors, at scales that have already exceeded the 1,000 node threshold. These "*smart dust*" [1] networks are enabling an exciting class of applications, including ecological studies [5], [6], active volcano monitoring [7], structural damage detection [8], [9], wildfire prediction and tracking [10], [11], disaster response [12], and intruder detection and classification [13]. Looking to the future, we expect an even richer class of applications to emerge as these networks become integrated into the international cyberinfrastructure.

Despite recent success and future promise, large-scale sensor systems remain difficult to design, debug, and deploy. These difficulties stem in large part from the inapplicability of existing analytical frameworks and simulation tools. The systems, when deployed at scale, are highly-distributed, concurrent, and reactive, resulting in a high degree of non-determinism in their execution behavior. They also tend to be embedded in environments that are rife with system hostility; network and node failure —both transient and persistent— are the norm rather than the exception. Existing analytical frameworks for reasoning about system correctness and performance offer limited suitability in this context. Simulation tools, while helpful, are also inadequate. Existing simulators fail to accurately model wireless signal propagation and interference [14], [15], nor do they capture the behavioral subtleties of underlying mote hardware platforms [16]. As a result, wireless sensor systems are constructed using cyclic development processes that rely on repeated *physical* experimentation. Hence, the community increasingly relies on shared *testbed* infrastructures. Indeed, numerous sensor network testbeds are in use at research institutions across the globe [17], [18], [19], [20], [21], [22], [23], [24].

While the design details vary from one testbed to another, these systems share a common architecture. Each is supported by a static collection of network nodes, and provides a software infrastructure for interacting with these nodes. The supporting infrastructure generally provides services that enable remote users to upload program images, map the images to physical devices, and collect message data for profiling purposes. The systems are typically batch-based, enabling multiple users to queue experiments for later execution. In [16], we present the design and implementation of the NESTbed system, an alternative testbed architecture for systems constructed using *nesC* [25] and *TinyOS* [26], [27] — the emerging platform standard in sensor system development. The testbed design differs from existing architectures in two important ways: First, the system is *real-time interactive*; it provides users with real-time access to network- and source-level symbol data without a priori consideration of the symbols to be profiled. These features serve not only to achieve the profiling goals of the system design, but also to enable the injection of network traffic and transient state faults (to, for example, assess the fault tolerance properties of a system under test). Second, the NESTbed architecture is *server-centric*; all phases of the experimentation lifecycle are exposed through a back-end server API, including pre-deployment activities, such as program image generation. Remote interfaces are developed as "*thin-clients*" using this API. Deferring *all* aspects of experimental control to the testbed server results in an architecture that supports features precluded by existing designs, including customized source-level analysis, instrumentation, and compilation. Programmer productivity is also improved since recurring tasks are automated by the server.

The prototype installation includes 80 *Tmote Sky* [28] motes arranged in a regular grid, and can be extended to support additional devices arranged in arbitrary topologies. Indeed, the NESTbed architecture is extensible in both the hardware *and* software dimensions. In this paper, we focus on the extensibility of the *software* design — in particular, on the extensibility of the *remote interface* design. The default software configuration includes a graphical remote interface for interacting with the testbed. The interface was designed with ease-of-use as a primary goal, and has been used to support both research and teaching activities at client institutions. A complete description of the graphical user interface is presented in [16], and summarized briefly in Section III. Based on our experiences working with this interface, and the experiences of our colleagues, two key limitations have been identified. First, for remote users outside of the server's hosting domain, the interaction latency introduced by the graphical interface reduces the timeliness of profiling results, hinders user interaction, and significantly reduces the overall usability of the tool. Second, and perhaps more important, the interface is not well-suited to performing a large number of tasks conveniently — especially repetitive tasks. Querying the value of a program symbol on each device at various points during a system's execution, for example, requires a high-degree of user interaction, and is both tedious and error-prone.

**Contributions.** We present two contributions. (**i**) First, we describe the design and implementation of a complementary remote interface for the NESTbed system that addresses the limitations of the existing graphical interface. The new *NEST-Shell* interface provides a file system abstraction for remote users, through which all of the NESTbed system features can be accessed. Further, to enable automation of complex and/or repetitive experimentation tasks, the interface provides an interpreted scripting facility. The scripting language provides constructs for interacting with external (client-side) tools, further enhancing the extensibility of the interface design. We demonstrate the utility of the new interface in the context of a small, but representative use-case scenario. (**ii**) Second, by virtue of developing an alternative interface without modifying (or compromising the operation of) the original system, we demonstrate the extensibility of the NESTbed architecture.

**Paper Organization.** In the next section, we survey key elements of related work, and highlight the novelty of the NESTbed system. In Section III, we summarize the original system architecture, with a focus on the server API used by the NESTShell interface. In Section IV, we present the design and implementation of the new interface. We present a representative use-case scenario in Section V. Finally, we conclude with a summary of contributions, and propose future points of NESTbed extension.

## II. Related Work

The difficulty of achieving predictable performance in wireless sensor networks is well-recognized. Numerous experimentation tools have been proposed to reduce this difficulty; we summarize some of the most relevant here. Our focus is on tools that support *physical* experimentation, as opposed to *simulation-based* experimentation. We note, however, that general purpose wireless simulators [29], [30], [31], and sensor network specific simulators [32], [33], [34] have proven invaluable to the research community in establishing first measures of system performance. These tools are not, however, sufficient by themselves; existing simulators are unable to faithfully model wireless signal propagation and interference [14], [15], nor do they accurately capture the subtleties of underlying hardware platforms [16]. Physical experimentation remains a necessity.

Closer to our work are *hybrid* frameworks that combine physical experimentation and simulation. This approach is often applied in the context of ethernet networks [35], [36], [37], and has more recently gained application in the context of wireless sensor networks. One example is the *EmStar* development platform [38]. Applications developed using EmStar can be executed using *EmCee*, a simulator capable of dispatching radio instructions to physical devices. Combined with *EmTOS* [39], an extension that enables mote-class applications to be executed within an *EmStar* application, designers can test their mote-class systems under a range of network realizations. Still, this tool suite is unable to faithfully simulate mote hardware. The tools cannot, for instance, account for hardware interrupts or load-induced violations of synchronization primitives. By contrast, our design relies solely on physical experimentation, thus yielding high precision.

The NESTbed design is not, however, the first to support pure physical experimentation. Testbeds of this type are commonly used in the context of 802.11 studies [40], [41], [42], [43], and are increasingly common in the sensor networks domain [17], [18], [19], [20], [21], [22], [23], [24]. We consider two representative testbeds from the latter category.

One of the first sensor network testbeds described in the literature is the *MoteLab* testbed [22] deployed at Harvard. The network includes 190 motes [44] attached to ethernet-based gateway devices, enabling network reprogramming through a centralized server. The system exposes a web interface that enables users to upload application images, and to map the images to physical devices. Users may additionally upload custom *Java* classes used to parse and store USB data (transmitted during system execution) for later retrieval. MoteLab is *batch-based* rather than *interactive*; it uses a queuing system for experiment scheduling. It does not support real-time source- or network-level profiling, nor the injection of transient state faults. MoteLab users are also required to generate the required application images, as well as the corresponding Java logging classes; the design is *client-centric* rather than *server-centric*. Consequently, MoteLab does not support automated source-level analysis or instrumentation. Finally, the MoteLab server does expose an API for programmatic control; the remote interface design is not extensible.

The *Kansei* testbed [17] deployed at Ohio State is a more recent example designed to support multi-tiered networks. The system includes over 400 motes arranged in stationary, portable, and mobile arrays. The overall architecture parallels the MoteLab design, but the software architecture includes several novel features, including job coordination facilities, system health monitoring, event injection, and sensor stream scaling. While Kansei is designed for *batch-style* experimentation, the NESTbed system is designed for *interactive* experimentation. Kansei does not support real-time profiling or fault injection, and provides limited traffic logging support [45]. It is also *client-centric*; automated source-level analysis and instrumentation are not supported. Developers are required, for example, to manually integrate specialized components as part of each application image. Finally, the granularity of control provided to external applications by the Kansei API is unclear.

MoteLab and Kansei are representative of testbed projects under development around the world [18], [19], [20], [21], [23], [24]. While the NESTbed design shares goals and architectural principles, existing systems are batch-based and client-centric. By contrast, the NESTbed design supports *interactive*, *server-centric* experimentation and evaluation.

Finally, it is important to note that the NESTShell scripting interface shares similarities with *Marionette* [46], a tool suite for querying and controlling wireless embedded devices. Marionette provides a Python interface for reading and modifying program state at runtime, as well as invoking nesC commands. Like NESTShell, Marionette enables developers to script debugging and profiling activities. It is not, however, tailored for *testbed* experimentation; it lacks services for managing projects and deployment configurations, reprogramming devices, constructing network gateways, and others. Marionette's integration with a popular object-based scripting language, however, is a point of advantage.

## III. NESTbed System Architecture

We now turn our attention to the NESTbed system architecture, and summarize key aspects of the design detailed in [16]. The focus is on the server API, since this is the API used to construct the NESTShell interface.

An overview of the architecture is illustrated in Figure 1. As shown in the figure, the testbed supports multiple network deployments, each connected to an application server using a series of USB hubs. Our prototype installation, shown in Figure 2, includes 80 *Tmote Sky* [28] devices arranged in a grid measuring 4'x8'[1]. The Tmote Sky platform is a popular research device; it includes an *MSP430* microcontroller operating at 8Mhz, 48K of ROM, 10K of RAM, and a 2.4GHz wireless transceiver. The transceiver is used for all in-network communication; the USB connection is used as an *out-of-band* link by the NESTbed server to manage and power the attached device. The management services are realized as a suite of six APIs exposed to remote applications using *Java RMI* [47]. We briefly summarize these APIs in the paragraphs that follow.

[1]The density of our prototype deployment is an artifact of spatial constraints. The deployment can be configured to support connections in excess of 150' with the addition of wireless USB extenders. But as we will see, even in this confined space, the server API provides radio power management features to ensure the construction of representative routing topologies.

**Configuration API.** The *Configuration API* provides services for managing *projects* and *deployment configurations* (maintained by the server in persistent storage). A project consists of source materials uploaded by an end-user, and an associated collection of meta-data (*e.g.*, program symbol and message structure information). A deployment configuration specifies the application image to install on each device, runtime profiling options, and radio power settings. Multiple deployment configurations may be specified for each project.

**Instrumentation and Compilation API.** The *Instrumentation and Compilation API* provides services to instrument and compile project source materials. The API automates the integration of NESTbed management components, and provides options to replace default system components (*e.g.*, radio stack, sensor drivers) with alternative implementations chosen from a library or uploaded by a user. Compilation services automate program compilation activities, and provide detailed result reporting to client applications. The API additionally provides analysis services to identify program symbols and message structures used to populate project meta-data.

**Deployment API.** The *Deployment API* provides services to *activate* a deployment configuration. This process involves programming and configuring the network based on the settings specified by a given configuration. The API provides per-mote and whole-network programming functions, as well as error detection, error reporting, and error recovery support.

**Profiling API.** The *Profiling API* provides source- and network-level profiling functions. The source-level functions enable client applications to read and write program variables during system execution, supporting both profiling and fault injection objectives. The network-level functions enable clients to subscribe to *message streams* corresponding to message traffic at one or more nodes. As we will see, this is implemented using a radio-to-USB forwarding mechanism.

**Power Control API.** The *Power Control API* provides services for toggling the power of network nodes. These services are implemented using USB power control functions included as part of the *USB 2.0* standard. The API services support the injection of transient and persistent node failures, as well as recovery from unresponsive device states[2].

**Gateway Control API.** The *Gateway Control API* provides services for managing a set of TinyOS `SerialForwarder` instances [27]. Each instance serves as a mote-to-TCP bridge for a particular device. Messages transmitted by the device are relayed to an advertised TCP port; messages transmitted to the port are relayed to the device. The API provides functions

[2]This API was not included as part of the implementation discussed in [16].

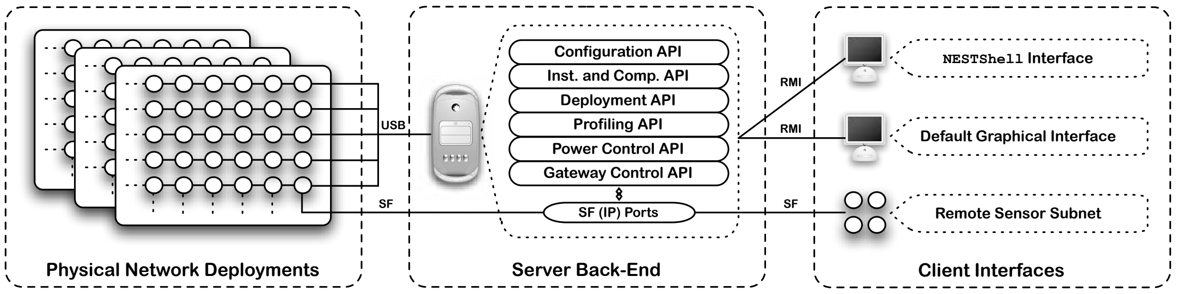


Fig. 1. NESTbed System Architecture

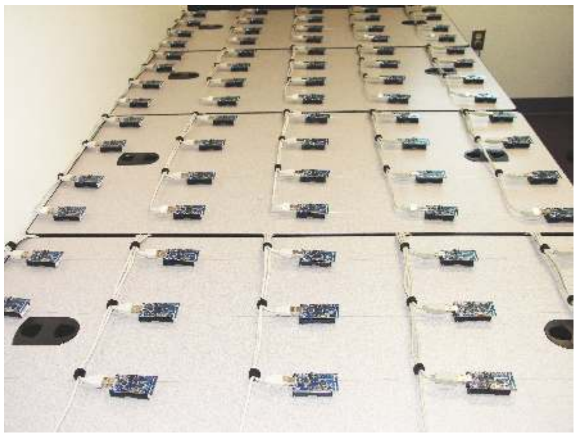


Fig. 2. Prototype NESTbed Deployment (80 Motes)

to associate and disassociate `SerialForwarder` instances with individual nodes. This *gateway* support enables remote clients to extend the testbed infrastructure with custom sensor subnets, and upper-tier control and analysis tools.

As noted earlier, the default testbed interface was designed with ease-of-use as a primary goal. It consequently provides "*point-and-click*" access to the features exposed by the NESTbed server. For instance, creating a new deployment configuration involves mapping each program image to its corresponding device using a standard "*drag-and-drop*" interface. During execution, retrieving the value of a program symbol involves selecting the relevant device and choosing the symbol of interest. Similar interfaces are provided for managing power, injecting faults, accessing network traffic, etc. More important than the latency issues associated with using the interface from outside of its hosting domain, the point-and-click access format is ill-suited to tasks that are repetitive, complex, or involve a large network subset. It is inconvenient, for instance, to query the value of a particular program symbol on *every* device within the network at various points during system execution. It is similarly inconvenient to bring nodes on- or off-line in a particular order, or to inject repetitive state or fail-stop faults. This is especially significant in scenarios involving iterative experimentation, where a series of steps must be followed repeatedly. The NESTShell interface was designed to complement the graphical interface by addressing experimentation scenarios of this type.

## IV. THE NESTSHELL INTERFACE

The NESTShell interface is designed to enable remote users to interact with the NESTbed system in a manner that parallels the way in which users interact with a typical operating system shell. The goal is to provide convenient manual and script-based access to the NESTbed system features, while reducing interaction latency (by avoiding network-intensive graphics) — this, of course, without reducing the level of information detail available to end-users. At the core of the NESTShell implementation is a file system abstraction that models the hierarchical structure of (*i*) physical network deployments, (*ii*) NESTbed projects, (*iii*) deployment configurations, (*iv*) programs, and (*v*) profiling data. Users traverse the file system and interact with the elements that it contains using familiar *UNIX-style* concepts and command primitives.

Each directory within the file system defines a *command context*. A user's active directory defines the active context, and dictates the set of available commands. For example, when the active directory is the *project management* directory, the shell provides commands for managing projects. Similarly, when the active directory is the *symbol profiling* directory for a particular device, the shell provides commands for reading and writing program variables defined by the application executing on the device. A directory may also contain files used to convey information about the active context. A *mote* directory, for instance, includes a file that specifies information about the corresponding network node (*e.g.*, deployment coordinates, executing program image, hardware characteristics). The contents of this file (and others) are read using standard UNIX-style commands (*e.g.*, `cat`).

The NESTShell file system structure is shown in Figure 3. Boldface labels correspond to literals; italicized labels are placeholders for names that vary. The commands applicable in each directory appear in Table I. In the paragraphs that follow, we describe the purpose of each directory, and the usage of the associated commands.

### A. Experiment Configuration

The root directory of the file system contains subdirectories corresponding to the physical deployments available for use. These subdirectories are "created" automatically based on static configuration data exposed through the server API. In our current installation, only one physical deployment is

| *Command* | *Directory* | *Description* |
|---|---|---|
| cat | *all* | Display the contents of a file |
| cd | | Change the working directory |
| ls | | List the contents of the current directory |
| man (help) | | Get help on the commands in the current directory |
| quit (exit) | | Exit the application |
| set | | Set the value of a variable |
| unset | | Unset the value of a variable |
| echo | | Display a line of text |
| env | | Display the name and value of all variables |
| shell | | Execute a system-level command |
| foreach | | Loop over a list of items and execute a set of commands |
| iferror | | Conditionally execute a set of commands if the last command failed |
| [mk/rm]proj | *Physical Deployment* | Create a new project or remove an existing project by name |
| [mk/rm]conf | *Project* | Create a new deployment configuration or remove an existing deployment configuration by name |
| upload | Programs | Upload a new program |
| rm | | Remove an existing program |
| profile | Messages | Select a message type to be profiled |
| | Symbols/*Module* | Select a program symbol to be profiled |
| rm | SymbolProfiling | Deselect a program symbol to be profiled |
| | MessageProfiling | Deselect a message type to be profiled |
| configure | Motes | Configure a mote to run the specified program at the specified radio power level |
| unconfigure | | Unconfigure the specified mote |
| ls | Motes | Directory-specific ls; displays network information |
| | NetworkMonitor | Directory-specific ls; displays network information and mote state |
| install | Network Monitor | Installs a program on the specified mote |
| wait | | Wait for current installations to complete |
| reset | | Perform a soft reset on the specified mote |
| power[On/Off] | | Power-on or power-off the specified mote |
| [mk/rm]gw | | Create or destroy a network gateway for the specified mote |
| query | *Mote*/SymbolProfiling | Query the mote for the value of the specified symbol |
| write | | Write the specified value to the specified symbol |

TABLE I
NESTSHELL COMMAND SUMMARY

installed; hence, users have access to only one deployment directory. Within a deployment directory, as in all directories, users have access to the standard commands. In addition, they have access to commands used to create and remove project directories. The project creation command requires project name and description arguments. The description is stored within a file located in the corresponding directory.

Deployment configurations are represented as subdirectories beneath each project, and are managed in a similar manner. As shown in the figure, a configuration directory contains five subdirectories: (*i*) Programs, (*ii*) SymbolProfiling, (*iii*) MessageProfiling, (*iv*) Motes, and (*v*) NetworkMonitor. We describe each of these directories and the subdirectories they contain (in a depth-first fashion) in the paragraphs that follow.

As its name suggests, Programs contains subdirectories corresponding to the applications uploaded by an end-user[3]. The associated command context includes commands for uploading and removing applications. The upload command requires an application name as argument, an associated description, and a path to the application source materials on the user's local machine. When the command completes (and the application data has been uploaded to the server), the new program directory is created (beneath Programs), and two subdirectories are created beneath it, Symbols and Messages. The first subdirectory corresponds to *program symbols*, and contains subdirectories that match the nesC modules defined within the uploaded application. Within each of these subdirectories are files corresponding to the program symbols declared by the respective module. The associated command context enables users to select a program symbol for profiling, making it available for runtime access. The Messages subdirectory contains files corresponding to the *message structures* declared by the uploaded application. These structures are not associated with particular modules — hence the omission of the module directories. The associated command context is analogous to that associated with the Symbols directory.

The next subdirectories beneath a deployment configuration are SymbolProfiling and MessageProfiling. These contain files corresponding to the program symbols and message structures, respectively, that have been selected for runtime profiling. The files are populated based on the selection commands discussed in the preceding paragraph. The associated command contexts include commands to deselect symbols and messages to cancel profiling of previously selected elements.

Next is the Motes directory, which provides a context for configuring the images to install on the network when the current deployment configuration is *activated*. The directory contains a file for each mote, which specifies information about the hardware characteristics of the device (*e.g.*, network address, deployment coordinates, memory capacity), and its current *configuration status*. The configuration status includes the application to install on the device, and the radio power

[3]Applications uploaded by a user are shared across deployment configurations within a project. Hence, although each configuration directory includes a Programs subdirectory, this is only a syntactic convenience; the Programs subdirectory is conceptually stored beneath the containing *project* directory.

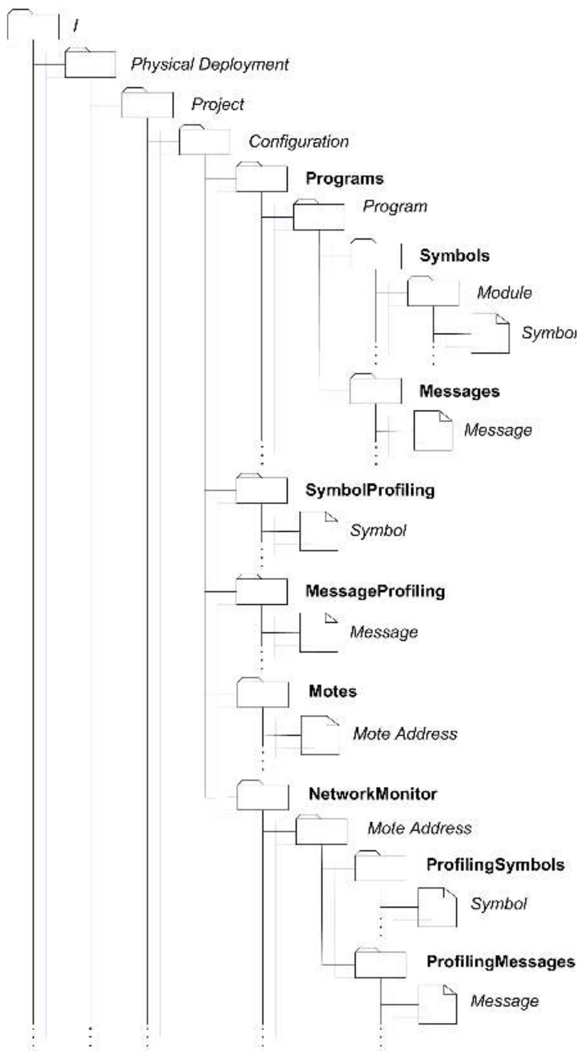
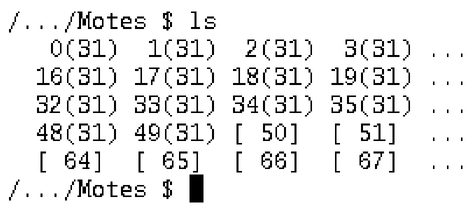


Fig. 3. NESTShell File System Structure



```
/.../Motes $ ls
   0(31)  1(31)  2(31)  3(31) ...
  16(31) 17(31) 18(31) 19(31) ...
  32(31) 33(31) 34(31) 35(31) ...
  48(31) 49(31) [ 50]  [ 51]  ...
  [ 64]  [ 65]  [ 66]  [ 67]  ...
/.../Motes $ █
```

Fig. 4. ls Command Results (Motes)

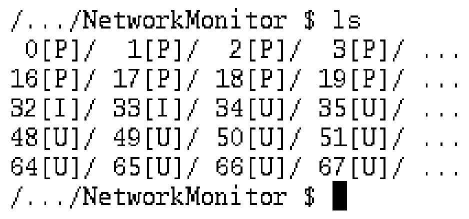
level to be set when the application is executed. The command context for the Motes directory overrides the standard ls command to provide a formatted display that includes the configuration status of each mote. A sample of the output produced by this command is shown in Figure 4. The numbers in parentheses indicate the radio power level of *configured* motes; *unconfigured* motes are shown in square brackets. Additional detail (*e.g.*, program image information, hardware characteristics) can be retrieved by invoking cat on the individual mote files. The context additionally provides commands for configuring and unconfiguring a device. The configuration command requires the address of a device, the name of an application contained in the Programs directory, and the desired radio power level. The command used to unconfigure a device clears the configuration status of the mote specified as argument.

## B. Experiment Execution

Each deployment configuration directory includes a NetworkMonitor subdirectory that defines a context for controlling the current network deployment. The most important commands provided in this context are install and wait. The install command is used to *activate* the current deployment configuration on a specified mote. This involves programming the device using the application image mapped to it in the Motes directory, and setting the requested power level upon installation. The command executes asynchronously to allow users to initiate concurrent installation requests — to, for example, activate the current deployment configuration in a *whole-network* fashion. After initiating a sequence of install commands, the user can issue a wait command to block until the programming step completes; the user will be notified of the aggregate installation results upon completion. The context includes additional commands to perform a soft reset on a specified mote, and to toggle the power supply to a specified mote. Finally, the context provides commands to create and destroy SerialForwarder *gateways*. The gateway creation command requires the address of the device that will serve as the gateway, and prints the resulting server-assigned IP port. The command used to destroy a gateway accepts a mote address as argument, and frees the resources associated with the corresponding gateway.

NetworkMonitor includes subdirectories corresponding to the network nodes. As before, each subdirectory contains a file that specifies information about the hardware characteristics of the corresponding device. In addition, each file specifies information about the device *activity state*. Initially, each device is in an *unknown* state since the runtime state of the network is not maintained in persistent storage. When an install command is issued on a device, the device state changes to *installing*. Depending on whether the program installation succeeds, the mote enters either the *programmed* state or the *failed* state. When a device is in the *programmed* state, it can be used as a gateway, at which point it enters the *gateway* state. (When the gateway is destroyed, the device returns to the *programmed* state.) To simplify the collection of aggregate status information, the command context of the NetworkMonitor directory overrides the standard ls command to include information about the activity state of the network. A sample of the output produced by this command is shown in Figure 5. The symbol shown in brackets indicates the respective node's activity state (*i.e.*, P=*programmed*, U=*unknown*, etc.).

Finally, each mote directory beneath NetworkMonitor includes two subdirectories used for profiling purposes, ProfilingSymbols and ProfilingMessages. ProfilingSymbols contains files corresponding to the program symbols previously selected for profiling; the file names match those contained



```
/.../NetworkMonitor $ ls
 0[P]/  1[P]/  2[P]/  3[P]/ ...
16[P]/ 17[P]/ 18[P]/ 19[P]/ ...
32[I]/ 33[I]/ 34[U]/ 35[U]/ ...
48[U]/ 49[U]/ 50[U]/ 51[U]/ ...
64[U]/ 65[U]/ 66[U]/ 67[U]/ ...
/.../NetworkMonitor $ █
```

Fig. 5. ls Command Results (NetworkMonitor)

in SymbolProfiling. Each file contains the most recent value recorded for the corresponding symbol. The command context includes a query command to update this value based on the symbol's *current* runtime value. It additionally includes a write command to overwrite the existing value, which in turn causes the state of the executing device to be modified. The ProfilingMessages directory is defined analogously; its contents mirror those of MessageProfiling. The command context for ProfilingMessages includes commands to subscribe and unsubscribe to *message streams*. When a user subscribes to a message stream associated with a particular message structure, the content of the corresponding file is initially cleared. Messages received over the USB port of the active device that are of the appropriate message type are appended to this file[4]. Each log entry includes a line-separated list of the values contained within the record fields using a simple field=value format. The logging process continues until the user unsubscribes from the message stream.

### C. Environment Variables and Control Flow

The NESTShell interface includes a global *environment map* used to store variables that can be referenced in NESTShell commands. The commands used to interact with the environment are similar to those found in UNIX-style operating system shells. The commands set, unset, and env, for example, allow a user to set the value of an environment variable, remove a variable, and display the contents of the environment, respectively. The familiar ${variable} notation is used to access the value of a variable. As in an operating system shell, the environment map simplifies the interface by enabling users to define aliases for complex and/or recurring strings. We will see in Section V that this is especially useful in the case of NESTShell scripts, where environment variables serve as a convenient parameterization mechanism.

In addition to user-defined variables, the environment contains the status *system* variable. This variable stores the *exit status* of the last executed command. It might, for example, be used to determine whether a write against a particular program symbol was successful, and to trigger the execution of associated recovery logic if it was not.

We note that the interface also includes *iteration* and *conditional evaluation* constructs to enable users to express more complex experimentation and evaluation scenarios. We will see examples of these constructs in Section V.

[4]A number of radio-to-USB forwarding components are freely available for TinyOS. Installing one of these components as part of an application image enables users to easily record network traffic through the mote's USB port.

### D. Interface Interoperability

We conclude this section by emphasizing that the NESTShell interface is intended to complement, rather than replace, the default graphical interface. In some scenarios, the graphical interface is appropriate; in others, the NESTShell interface is a better choice. A novice user might, for example, prefer using the testbed through a "*point-and-click*" interface for simple debugging tasks. An expert user performing a series of complex experiments is likely to prefer the shell-based interface. The point is that the user is free to choose the interface that best addresses the task at hand.

Finally, We note that there are some features provided by the graphical user interface that are not provided by the NESTShell interface. In particular, the latter interface does not provide commands for source-level instrumentation. The manner in which these features should be integrated with the shell abstraction is unclear. As a stop-gap measure, users can access the instrumentation features through the graphical interface as part of configuring a NESTbed project. This same project may then be accessed using the NESTShell interface.

## V. Use-Case Scenario

We now turn our attention to a use-case scenario designed to demonstrate the utility of the NESTShell design when the interface is applied in the context of a typical evaluation task. The scenario involves runtime profiling of an 80 node network running a slightly modified version of the SurgeTelos application included as part of the TinyOS distribution [27]. For the sake of presentation, we demonstrate the interface using an *experimentation script* that can be executed by the shell interpreter. Alternatively, the contents of the script can be entered interactively.

SurgeTelos implements a distributed sensing infrastructure. Participating nodes execute a spanning tree protocol, with a pre-selected mote serving as the *root node*. Each device periodically polls its attached photo sensor, and conveys the readings to the root node using the spanning tree as a routing structure. We modified the basic application to record the *RSSI* (*received signal strength*) and *LQI* (*link quality indicator*) readings associated with the last packet received from each node's *parent* in the spanning tree. RSSI and LQI readings are commonly used as link quality metrics, and inform the parent selection process in the SurgeTelos implementation.

The profiling task involves installing the SurgeTelos application under three different deployment configurations, each corresponding to a different radio power setting. In each configuration, the goal is to allow the routing tree to stabilize for a period of time before collecting five elements of profiling data from each node: (*i*) the RSSI and LQI readings mentioned previously, (*ii*) the address of the node's parent, (*iii*) the node's *hop count* from the root, and (*iv*) the internal link quality metric used to inform parent selection.

The experimentation script used to perform the evaluation task appears in Listing 1. (Due to space constraints, portions of the script have been simplified or elided.) Key elements are described in the paragraphs that follow.

```
1  set MOTES="[0-79]"
2  set LEVELS="[1-3]"
3  cd "Ultra-Dense Network"; cd "Surge Evaluation"
4  iferror; then
5      mkproj "Surge Evaluation" \
6             "Evaluation of RSSI/LQI"
7      cd "Surge Evaluation"
8  endif
9  foreach powerLevel in ${LEVELS} do
10     mkconf "Power Level ${powerLevel}" \
11     "SurgeTelos at Power Level ${powerLevel}"
12 done
13 cd "Power Level 1"; cd Programs
14 upload SurgeTelos \
15        "SurgeTelos Application" \
16        /opt/tinyos-1.x/apps/SurgeTelos
17 ... iferror, exit ...
18 foreach powerLevel in ${LEVELS} do
19     echo "-- Power Level ${powerLevel}"
20     cd /; cd "Ultra-Dense Network"
21     cd "Surge Evaluation"
22     cd "Power Level ${powerLevel}"
23     cd Programs; cd SurgeTelos; cd Symbols
24     cd MultiHopLQI
25     foreach i in
26         rawRSSI rawLQI gbCurrentParent \
27         gbCurrentHopCount gbCurrentLinkEst do
28         profile ${i}
29     done
30     ... cd .. back to configuration directory ...
31     cd Motes
32     foreach i in ${MOTES} do
33         configure ${i} SurgeTelos ${powerLevel}
34     done
35     cd ..; cd NetworkMonitor
36     foreach i in ${MOTES} do
37         install ${i}
38     done
39     echo "Waiting for installation to complete"
40     wait
41     ... iferror, exit ...
42     echo "Waiting for experiment to complete"
43     shell sleep 1m
44     foreach mote in ${MOTES} do
45         echo "Querying ${mote} symbols"
46         cd ${mote}; cd ProfilingSymbols
47         foreach sym in
48             rawRSSI rawLQI gbCurrentParent \
49             gbCurrentHopCount gbCurrentLinkEst do
50             query MultiHopLQI.${sym}
51         done;  ... cd ..; cd .. ... ; done; done
```

Listing 1. SurgeTelos Experimentation Script

**Lines 1–2.** The script first declares the variables MOTES and LEVELS, used to parameterize the subset of devices to be programmed, and the power levels to be considered, respectively. This enables users to easily modify the script to execute on the network subset and power levels of interest.

**Lines 3–8.** Next, the "*Surge Evaluation*" project is selected within the "*Ultra-Dense Network*" deployment. The iferror condition checks the value of the status variable (set by each NESTShell command), to determine whether the selection was successful. Hence, if the project does not exist, it will be created. The second parameter to mkproj provides a description for the new project. At the termination of the block, the current directory is set to the new project directory.

**Lines 9–12.** Within the project directory, each of the three deployment configurations are created. This is achieved using a foreach construct that iterates through each of the power levels defined in LEVELS. The mkconf command is analogous to the mkproj command; note, however, the use of the powerLevel variable in defining the name of the deployment configuration directory and the associated description.

**Lines 13–17.** Next, the current directory is changed to the Programs subdirectory beneath the first configuration. The SurgeTelos application is then uploaded from the user's local machine, using the specified name and description. (Recall that uploaded applications are shared across the deployment configurations within a project.)

**Line 18.** The remainder of the script is contained within the body of the loop initiated on this line. It iterates through the selected power levels to (*i*) complete the process of configuring each deployment configuration, (*ii*) activate each configuration, and (*iii*) collect the required profiling data.

**Lines 19–29.** The first step within the loop body is to select the program symbols to be profiled. This is achieved by changing the current directory to the MultiHopLQI module directory. The nesC module of the same name defines the symbols of interest. These symbols, selected in the body of the foreach loop, correspond to the five data elements enumerated in the discussion of our profiling goals.

**Lines 30–34.** In the Motes directory, the foreach block configures each device in the selected subset. Given the value of the MOTES variable, all 80 motes are configured with the SurgeTelos application at the current power level.

**Lines 35–41.** Next, the install command is used to activate the current deployment configuration on each device in the network. Recall that this command executes asynchronously; the network is programmed in parallel. The wait command blocks until the pending installs are complete, and sets the status variable (used by iferror) appropriately.

**Lines 42–43.** When the installation process completes, the experimentation script remains idle for one minute to allow the network routing structure to stabilize. Note that the shell command enables a NESTShell script to invoke commands in the hosting *operating system* shell. In this case, the *UNIX* sleep command is used to implement the idle period.

**Lines 44–51.** Finally, after the one minute idle period, the script iterates through each device, and queries the runtime value of each of the five program symbols being profiled. Note that in addition to updating the content of the relevant symbol file, the query command displays the retrieved value to the console. (If desired, the script output can be redirected to a file using standard redirection primitives.)

## A. *Result Summary*

While the focus of the presentation is on the enabling features of the NESTShell interface, it is useful to briefly summarize the results of executing the experimentation script. The purpose is to emphasize the types of studies that can be performed using the interface.

| *ID* | *Par* | *RSSI* | *LQI* | *ID* | *Par* | *RSSI* | *LQI* |
|---|---|---|---|---|---|---|---|
| **1** | 0 | 215 | 105 | **35** | 32 | 211 | 83 |
| **2** | 0 | 217 | 106 | **36** | 20 | 219 | 107 |
| **3** | 0 | 215 | 103 | **48** | 32 | 219 | 108 |
| **4** | 20 | 213 | 103 | **49** | 32 | 215 | 99 |
| **16** | 16 | 216 | 103 | **50** | 48 | 210 | 105 |
| **17** | 32 | 211 | 95 | **51** | 20 | 215 | 87 |
| **18** | 33 | 211 | 104 | **52** | 20 | 211 | 102 |
| **19** | 33 | 210 | 95 | **64** | 32 | 213 | 104 |
| **20** | 48 | 206 | 69 | **65** | 32 | 212 | 103 |
| **32** | 0 | 218 | 103 | **66** | 67 | 219 | 105 |
| **33** | 32 | 215 | 106 | **67** | 48 | 207 | 81 |
| **34** | 0 | 207 | 91 | **68** | 67 | 218 | 106 |

TABLE II
SurgeTelos EXPERIMENT RESULTS (PARTIAL)

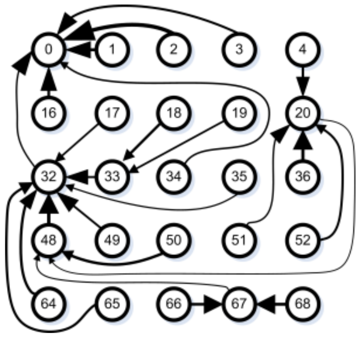


Fig. 6. SurgeTelos Experiment Results (partial)

Profiling data collected for a subset of the nodes appears in Table II; the table includes RSSI, LQI, and parent address information for 24 devices. This information is represented graphically in Figure 6. Each circle corresponds to a device; arrows represent parent links. The size of each arrow head is proportional to the RSSI measured on the link, and the corresponding line width is proportional to the LQI measurement. Not surprisingly, the two metrics are strongly correlated.

Again, the point is to emphasize the expressivity of the scripting language, and the *controllability* and *observability* offered by the file system abstraction. Even with little experience, users can quickly specify and perform complex configuration, deployment, and profiling scenarios.

## VI. Conclusion

In [16], we presented the NESTbed system, an *interactive*, *server-centric* testbed for wireless sensor networks. The testbed is designed to support rapid experimentation, debugging, and profiling of network applications constructed using *nesC* [25] and *TinyOS* [26], [27] — the emerging platform standard in the sensor networks domain. The system is deployed on the Clemson University campus, and includes 80 *Tmote Sky* devices [28] exposed for shared, interactive use. The system includes a graphical user interface constructed using the NESTbed server API; it enables remote clients to easily configure, deploy, and profile their systems on one or more fixed network deployments. The system is used to support a range of research and teaching activities, and is accessed by researchers and students from both Clemson University and Cleveland State University. Our experiences working with the system over the past eight months provide the point of departure for the work presented here.

While the graphical user interface has proven useful — especially for students and novice users— we have faced two significant challenges. First, when accessed from outside of its hosting domain (*i.e.*, the Clemson campus), the graphical interface introduces interaction delays that compromise the freshness of profiling data, and reduce the overall usability of the tool. Second, and more important, the tool is ill-suited to performing tasks that are complex, repetitive, and/or involve a large number of devices. An interpreted scripting interface is preferable in such cases.

**Contributions.** To address the limitations of the graphical interface, we presented the design and implementation of *NESTShell*, an alternative *shell-based* interface for the NESTbed system. The interface exposes the testbed infrastructure using a file system abstraction that enables clients to interact with the testbed in a manner similar to that used when interacting with a file system through an operating system shell. The interface includes a full set of shell commands, and an interpreted scripting facility that enables users to control the testbed programmatically. *Experimentation scripts* conveniently capture complex experimentation and evaluation scenarios, and can be stored for repeated use. The NESTShell commands and scripting features were demonstrated in the context of a small, but representative case-study. We emphasize that the NESTShell interface uses the original NESTbed server API, and co-exists with the graphical interface, demonstrating the extensibility of the NESTbed software design.

The NESTbed system and its associated user interfaces are continually refined based on user feedback. Refinement will continue as part of future work. Looking further out, we plan to pursue two additional activities. First, we plan to integrate the NESTShell file system abstraction as part of an existing operating system shell. Thus, for example, a user could interact with the NESTbed system as though it were an *actual UNIX* device, dramatically extending the suite of tools available for working with the testbed. This will involve adaptation of the existing abstraction, and the construction of suitable virtual device drivers to support the realization. Second, we plan to integrate ethernet-based gateway devices in the NESTbed architecture to enable physical deployments that are more geographically distributed. Our long-term goal is to deploy a *campus-wide* testbed consisting of nodes placed both indoors and outdoors. This would enable a rich class of experimentation scenarios, beyond what is supported by any existing sensor network testbed.

All of the software tools described in this manuscript, including source code, documentation, and script examples, are available by contacting the authors. (If accepted for presentation at TridentCom'07, these artifacts will be made publicly available through our web site.)

## Acknowledgements

This work is funded in part by a grant from the National Science Foundation (CNS-0520222), and a grant from the

South Carolina Space Grant Consortium. The authors gratefully acknowledge these agencies for their support.

## References

[1] B. Warneke *et al.*, "Smart dust: Communicating with a cubic-millimeter computer," *Computer*, vol. 34, no. 1, pp. 44–51, 2001.

[2] Crossbow Technology Incorporated, "Mica2 datasheet," http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0042-06_B_MICA2.pdf, 2003.

[3] ——, "Mica2Dot datasheet," http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0043-04_C_MICA2DOT.pdf, 2003.

[4] J. Hill, "Integrated $\mu$-wireless communication platform," http://webs.cs.berkeley.edu/retreat-1-03/slides/Mote_Chip_Jhill_Nest_jan2003.ppt, 2003.

[5] W. Hu *et al.*, "The design and evaluation of a hybrid sensor network for cane-toad monitoring," in *The 4th International Symposium on Information Processing in Sensor Networks*. IEEE, April 2005, pp. 503–508.

[6] A. Mainwaring *et al.*, "Wireless sensor networks for habitat monitoring," in *The 1st ACM International Workshop on Wireless Sensor Networks and Applications*. ACM, September 2002, pp. 88–97.

[7] G. Werner-Allen *et al.*, "Deploying a wireless sensor network on an active volcano," *IEEE Internet Computing*, vol. 10, no. 2, pp. 18–25, 2006.

[8] K. Chintalapudi *et al.*, "Monitoring civil structures with a wireless sensor network," *IEEE Internet Computing*, vol. 10, no. 2, pp. 26–34, 2006.

[9] S. Glaser, "Some real-world applications of wireless sensor nodes," in *SPIE Symposium on Smart Structures & Materials / NDE 2004*. SPIE Press, March 2004, pp. 344–355.

[10] C. Hartung *et al.*, "FireWxNet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments," in *The 4th International Conference on Mobile Systems, Applications, and Services*. ACM, June 2006, (to appear).

[11] D. Doolin and N. Sitar, "Wireless sensors for wildfire monitoring," in *SPIE Symposium on Smart Structures and Materials / NDE 2005*. SPIE Press, March 2005, pp. 477–484.

[12] K. Lorincz *et al.*, "Sensor networks for emergency response: Challenges and opportunities," *IEEE Pervasive Computing*, vol. 3, no. 4, pp. 16–23, 2004.

[13] A. Arora *et al.*, "Exscal: Elements of an extreme scale wireless sensor network," in *The 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, August 2005, pp. 102–108.

[14] G. Zhou *et al.*, "Impact of radio irregularity on wireless sensor networks," in *The 2nd International Conference on Mobile Systems, Applications, and Services*. ACM, June 2004, pp. 125–138.

[15] M. Takai *et al.*, "Effects of wireless physical layer modeling in mobile ad hoc networks," in *The 2nd ACM International Symposium on Mobile Ad Hoc Networking & Computing*. ACM, October 2001, pp. 87–94.

[16] A. Dalton and J. Hallstrom, "An interactive, server-centric testbed for wireless sensor systems," Clemson University (Dependable Systems Research Group), Tech. Rep. CU-DSRG-08-06-01, 2006.

[17] E. Ertin *et al.*, "Kansei: A testbed for sensing at scale," in *The 5th International Conference on Information Processing in Sensor Networks*. ACM, April 2006, pp. 399–406.

[18] University of Southern California, "Tutornet: A tiered wireless sensor network testbed," http://enl.usc.edu/projects/testbed/, 2006.

[19] V. Handziski *et al.*, "TWIST: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks," in *Proceedings of the 2nd International Workshop on Multi-hop Ad Hoc Networks: From Theory to Reality*. ACM, January 2006, pp. 63–70.

[20] D. Johnson *et al.*, "TrueMobile: A mobile robotic wireless and sensor network testbed," in *The 25th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, April 2006.

[21] UC Berkeley, "Soda hall wireless sensor network testbeds," http://www.millennium.berkeley.edu/sensornets/, 2006.

[22] G. Werner-Allen *et al.*, "MoteLab: a wireless sensor network testbed," in *The 4th International Conference on Information Processing in Sensor Networks*. IEEE, April 2005, pp. 483–488.

[23] B. Chun *et al.*, "Mirage: A microeconomic resource allocation system for sensornet testbeds," in *The 2nd IEEE Workshop on Embedded Networked Sensors*. IEEE, May 2005, p. 10pp.

[24] E. Welsh *et al.*, "GNOMES: a testbed for low-power heterogeneous wireless sensor networks," in *IEEE International Symposium on Circuits and Systems*. IEEE, May 2003, pp. 836–839.

[25] D. Gay *et al.*, "The nesC language: A holistic approach to networked embedded systems," in *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM, June 2003, pp. 1–11.

[26] J. Hill *et al.*, "System architecture directions for networked sensors," in *The 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 34, no. 5. ACM, November 2000, pp. 93–104.

[27] UC Berkeley, "TinyOS community forum —— an open-source OS for the networked sensor regime," http://www.tinyos.net/, 2004.

[28] Moteiv Corporation, "Tmote Sky datasheet," http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf, 2006.

[29] S. McCanne and S. Floyd, "Network simulator ns-2," http://www.isi.edu/nsnam/ns/, 1997.

[30] X. Zeng *et al.*, "GloMoSim: A library for parallel simulation of large-scale wireless networks," in *The 12th Workshop on Parallel and Distributed Simulation*. IEEE, May 1998, pp. 154–161.

[31] R. Barr *et al.*, "Scalable wireless ad hoc network simulation," in *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*, J. Wu, Ed. CRC Press, 2005, pp. 297–311.

[32] P. Levis *et al.*, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *The 1st ACM Conference on Embedded Networked Sensor Systems*. ACM, November 2003, pp. 126–137.

[33] B. Titzer *et al.*, "Avrora: Scalable sensor network simulation with precise timing," in *The 4th International Symposium on Information Processing in Sensor Networks*. IEEE, April 2005, pp. 477–482.

[34] G. Simon *et al.*, "Simulation-based optimization of communication protocols for large-scale wireless sensor networks," in *The 2003 IEEE Aerospace Conference*, vol. 3. IEEE, March 2003, pp. 1339–1346.

[35] J. Zhou *et al.*, "TWINE: a hybrid emulation testbed for wireless networks," in *The 25th IEEE Conference on Computer Communications*. IEEE, April 2006.

[36] P. De *et al.*, "MiNT: A miniaturized network testbed for mobile wireless research," in *The 24th Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE, March 2005, pp. 2731–2742.

[37] B. White *et al.*, "An integrated experimental environment for distributed systems and networks," in *The 5th Symposium on Operating Systems Design and Implementation*. ACM, December 2002, pp. 255–270.

[38] L. Girod *et al.*, "EmStar: a software environment for developing and deploying wireless sensor networks," in *The 2004 USENIX Technical Conference*. USENIX Association, June–July 2004.

[39] ——, "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *The 2nd International Conference on Embedded Networked Sensor Systems*. ACM, November 2004, pp. 201–213.

[40] A. Karygiannis and E. Antonakakis, "mLab: a mobile ad hoc network testbed," in *The 1st Workshop on Security, Privacy, and Trust in Pervasive and Ubiquitous Computing*. Diavlos S.A., July 2005, pp. 88–97.

[41] D. Raychaudhuri *et al.*, "Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols," in *Wireless Communications and Networking Conference*, vol. 3. IEEE, March 2005, pp. 1664–1669.

[42] E. Nordström *et al.*, "A testbed and methodology for experimental evaluation of wireless mobile ad hoc networks," in *Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*. IEEE, February 2005, pp. 100–109.

[43] B. Chambers, "The grid roofnet: a rooftop ad hoc wireless network," Master's thesis, Massachusetts Institute of Technology, May 2002.

[44] Harvard University, "MoteLab: harvard network sensor testbed," http://motelab.eecs.harvard.edu/, 2006.

[45] Ohio State University, "Using kansei — basics," http://exscal.nullcode.org/kansei/help.php, 2006.

[46] K. Whitehouse *et al.*, "Marionette: Using RPC for interactive development and debugging of wireless embedded networks," in *The 5th International Conference on Information Processing in Sensor Networks*. New York, NY, USA: ACM, 2006, pp. 416–423.

[47] Sun Microsystems, "Java(TM) remote method invocation (Java RMI)," http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html, 2004.