Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

A SCALABLE SIMULATOR FOR TINYOS APPLICATIONS

Luiz Felipe Perrone
David M. Nicol

Institute for Security Technology Studies
45 Lyme Rd. Suite 200
Dartmouth College
Hanover, NH 03755-1221, U.S.A.

ABSTRACT

Large clouds of tiny devices capable of computation, communication and sensing, goal of the *Smart Dust* project, will soon become a reality. Hardware miniaturization is shrinking devices and research in software is producing applications that allow devices to communicate and cooperate toward a common goal. Success on the software front hinges on the design of algorithms that can scale up with system size. Given that the number of individual cooperating devices will reach high orders of magnitude (hundreds of thousands or even millions), debugging and evaluating the software in such a large system can reap much benefit from simulation. This paper describes the design of a scalable and flexible simulator which allows for the direct execution, at source code level, of applications written for TinyOS, the operating system that executes on Smart Dust. This simulator also provides detailed models for radio signal propagation and node mobility.

1 INTRODUCTION

The idea of *Smart Dust*, which seemed seemed to be almost science fiction not long ago, is fast becoming a reality. This project born to a group at UC Berkeley, has aimed at producing tiny devices called *motes*, which are capable of some form of communication, using either laser or radio frequency, and are embedded with a microprocessor, memory and sensing circuits (Warneke et al. 2001). These devices, the motes, can be distributed over some portion of physical space where they self-organize into a communication network that is tightly integrated with the environment and, perhaps, even invisible. In large numbers, the devices can come close to being ubiquitous and, one small portion at a time, the physical world can be imbued with the ability to take inputs, process them and react back on the physical world (Estrin et al. 2002). One cannot overstate the impact this new technology will have on the way computing devices and the world interact.

In many ways, the diminutive size of the devices will be one of the key factors behind this revolution in the way computing is done. For this reason, it is only natural that the word *tiny* has been associated with the Smart Dust project from its inception. Their main goal at this time is to achieve a level of integration, using micro-electrical mechanical systems (MEMS) technology, that brings the form factor of the *mote* down to one cubic millimeter (Warneke, Atwood and Pister 2001). While the fruits of these efforts in miniaturization take time to ripen, devices with much larger form factor are becoming increasingly popular in the research community.

The *motes* currently in industrial production (CrossBow 2002) are perhaps unworthy of being called a speck of dust since their form factor is 1 inch by 1.5 inches. Time and experience are demonstrating, however, that even at this size, these wireless computing platforms have a wide range of applicability. Motes are enabling several projects in remote sensing as well as in robotics (Sibley et al. 2002). The success and the popularity of these devices is not only the result of advances in hardware, but also of ingenuity in the design of TinyOS, the system software that empowers them.

TinyOS can be considered as much of a technology enabler as motes themselves are. It is the operating system that allows Smart Dust to happen creating an environment of high flexibility for application design and execution over a severely constrained computing platform. Some may question what TinyOS really is, in this day and age when some operating systems have been more super-sized than meals at fast-food restaurants. What this operating system offers lies not so much in the range of services it provides, but rather in the principles that guide the *design* of these applications.

At the core of TinyOS are lean mechanisms for task scheduling and interrupt handling. The application design framework based on this foundation is one which allows the high flexibility desirable in problem-specific optimizations and ease of expression of the extensive level of concurrency inherent to the system. The programming model used in TinyOS imposes a hierarchical, component-based design resembling the organization of hardware, as we see later in this paper.

Although this hierarchical design methodology can lead to application programs that are easier to understand, construct and maintain, it does not eliminate the need for verification, debugging and, very likely, optimization before the code is run in production stages. Even if one takes great pains to ascertain the correctness and the performance of the application by trial runs on a few of motes, little can be said about what happens when conditions that affect communication change or when the number of motes participating in the network is substantially increased. In circumstances such as these, the value of conducting trial runs over a simulator is obvious and unquestionable.

For quite a while, the distribution of TinyOS has shipped with a simple simulator TOSSIM (Levis 2002) that allows the verification of basic properties of applications before they are loaded into motes for operation in the field. Although helpful for preliminary debugging and verification of applications, this simulator has proved to restrictive in a number of ways.

First, TOSSIM doesn't allow one to mix different applications in the same simulation run: all motes in the simulation must run exactly the same code. One can easily conceive of scenarios where different motes in a Smart Dust cloud would have specialized functions, thus running different applications, and still cooperate toward a common objective. With a bit of programming trickery, one can get around this limitation and make TOSSIM run different branches of the same code for different types of motes. For instance: if the identification for a mote is an even number, take a certain branch in the code and behave as application A, if the identification is an odd number, take another branch and behave as application B. While this may suffice for simulation purposes, it would require that the code executed on the simulator be a modification of the actual code that is loaded onto real motes for operation. This process of source code modification for simulation introduces the possibility of changing the behavior the application would exhibit in the real system, thus invalidating the results of the simulation. It would be much preferable if the simulation could run the code that goes in the mote *as-is*.

Second, this simulator is rather simplistic in modeling the environment where motes are placed. There is no provision for simulating the processes that stimulate the sensors on the motes. For instance, one could design a cloud of motes that can take measurements of temperature and the presence of a certain gas, then somehow process and exchange the collected data. Without models that accurately represent the conditions of temperature and diffusion of gases in the environment, simulation runs will not exhibit the same reactions that motes experience in the field and the verification of correctness or the estimation of performance of the application software are severely impaired. Another important environmental model which can be improved in TOSSIM regards the propagation of radio signals. Radio connectivity is described by an all-or-nothing approach: the radio channel is either perfect (effective bit error rate of zero), or totally broken, meaning that the motes cannot communicate. A common criticism to simulators of wireless communicating devices is that they hardly ever accurately portray what happens in the real world. Rather than give up on studying a wireless network by simulation for this reason, we propose the development of a more detailed simulator, one which accounts for how radio signals propagate on a given terrain with features specified by the modeler.

Finally, and most importantly, TOSSIM was not built with performance scalability in mind: the current documentation reports that simulations scale well up to one thoursand motes. One cannot use TOSSIM to efficiently run experiments where the number of motes has the same order of magnitude as that which is expected of large-scale Smart Dust systems. We may design applications that are intended to bring together hundreds of thousands of motes, or perhaps even more, but if the number of motes we can simulate is actually orders of magnitude smaller, simulation will teach us little about the dynamics of the larger system. Moreover, it is a well-known fact that simulation provides the experimentalist with not only a virtual environment with nearly endless possibilities, but also with powers of controllability and repeatability over the conditions where experiments are run. The importance of these powers is magnified when we consider the difficulty of controlling field experiments with large-scale wireless networks and the near impossibility of repeating all the conditions for each test trial. Even if one could amass the incredible resources to perform experiments with a large-scale Smart Dust system, deploying it and exercising any measure of control over an experiment with it would be a herculean task, not to mention what it would take to observe it in action.

These gaps in what current simulators for TinyOS devices have to offer and the long wish list of what would be desirable in a simulator have motivated us to start the project which we call TOSSF, a TinyOS Scalable Simulation Framework. Fortunately, our efforts in constructing TOSSF don't start from ground zero, but rather build up on two other projects in our group: DaSSF, the Dartmouth Scalable Simulation

Framework, and SWAN, our Simulator for Wireless Ad-Hoc Networks. We present more details on DaSSF and SWAN in later sections of this paper; for now, suffice it to say that DaSSF is a streamlined and optimized simulation kernel with a proved record for high-performance and scalability, and that SWAN is collection of C++ classes that builds up on the DaSSF kernel to offer a range of models for the simulation of wireless ad hoc networks.

The remainder of this paper is organized as follows. Section 2 presents concepts on TinyOS, expounding its philosophy of application design, its programming model and implementation. Section 3 introduces the reader to the most important concepts in DaSSF and SWAN, explaining the foundation upon which TOSSF is built, while Section 4 describes the architecture and the implementation of TOSSF. Finally, Section 5 presents the challenges in the evolution of TOSSF and the work that lies ahead in its development.

2 TINYOS CONCEPTS

One of the greatest contributions TinyOS makes to the development of applications for massively distributed, heavily constrained computing platforms is arguably its programming model. Applications are made of small, self-contained units called *components* which are interconnected by a directed graph to compose the greater picture. A TinyOS component is made of collections of *command* handlers, *event* handlers, and *tasks* plus its *frame*, a fixed-size portion of memory allocated at compilation-time to store the local state.

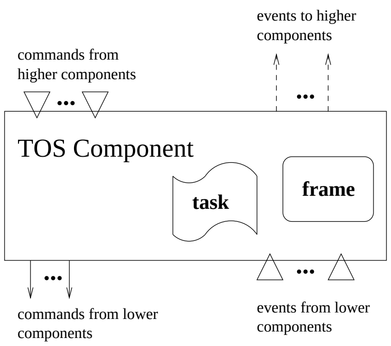

Figure 1: Structure of a TinyOS Component

Figure 1 illustrates the standard component structure defined in the TinyOS literature (Hill et al. 2000, TinyOS Programming Bootcamp 2001). Triangles pointing up represent event handlers. Events are software signals analogous to hardware interrupts. The orientation of the triangles is not an arbitrary choice, but rather indicates that the events a component handles can only be signaled by components below in the hierarchy. An event can signal higher events or issue commands to lower components. Similarly, triangles pointing down represent command handlers and indicate that only components placed above can issue commands to the component. The arrows indicate that a component can only issue commands to other components below and events to other components above. It is important to point out that commands are non-blocking function calls to a lower component (which can in its turn trigger commands to even lower components) and also that both event and commands are expected to execute in very short time.

Commands or events can post tasks into a FIFO queue handled by the scheduler for the mote. We can say that task scheduling is the only "service" implemented by TinyOS, or put in another way, that the task scheduler is basically the kernel of this operating system. Once a task starts to execute, it can only be preempted by the arrival of an event, neither by commands nor by other tasks. Tasks can post other tasks and operate only on data that is placed within the frame of the component to which they belong. When the queue empties out, in the absence of any events, the mote can enters a dormant, power-saving state.

Activity in a TinyOS application starts down at the hardware, the lowest level of components in the hierarchy. Events such as the arrival of a sensor reading from the analogue-to-digital converter or an incoming message from the radio receiver cause a hardware interrupt, which is handled by some component, which may, in its turn, initiate an upward flow of events. At a certain point, this upward flow changes direction and the components at the high-end of the chain issue a downward flow of commands with processing activities eventually ceasing until the arrival of another hardware interrupt. This architecture supports multiple flows, or threads, allowing for the representation of the extensive concurrency inherent to applications in this regime.

The programmer uses the C programming language to describe the component in two distinct steps. First, the interface is defined in a `.comp` file using keywords that label blocks containing function prototypes. The keywords `HANDLES` and `SIGNALS`, respectively, indicate the events the component handles and the events the component generates on higher components. The keywords `ACCEPTS` and `USES`, respectively, indicate the commands that the component exports to higher components and the commands from lower components that it calls. The `.comp` interface files describe the connection points for software *wires* that represent inputs and outputs of a component in the same way that one could describe a piece of hardware. Second, the "guts" of the component are defined in a dialect of the standard C syntax that introduces TinyOS keywords to specify structures such as the component's frame, the command handlers it implements, the command calls it makes, the event handlers it implements, the events it signals, etc.

In what TinyOS programming style is concerned, most components can be described by finite state machines (FSM). The first state in a component's FSM is an initialization state. This state is entered whenever the component receives a command to initialize is issued by another component in a higher position in the hierarchy. In practice, the `MAIN` component in the application, implemented by the `main()` function in C, spawns a wave of initialization commands that propagates out from its hooks and descends along the application graph causing all components to be initialized. After initialization, components enter a state where they wait for another wave of commands, this time a wave of `START` commands also spawned by the `MAIN` component, which take the components' FSMs into their main processing loop.

Each component can be defined in isolation of others or it can be defined as a collection of other components hiding the internal details and offering the programmer, at a higher-level of abstraction, just the functionality the collective implements. This design philosophy allows for the construction of databases of components from which the programmer can draw pieces to quickly assemble highly tailored applications.

Applications and components constructed from other components need to define the interconnections between its pieces. For this purpose, TinyOS uses description (`.desc`) files, which contain two different sections: first, an enumeration of the components used; second, how these components are interconnected. These interconnections are mappings between the endpoints of the wires defined in each component's `.comp` file and define a directed graph such as the one in Figure 2. For instance, if a component `CLOCK` signals the occurrence of an event `CLOCK_FIRE_EVENT` to another component `COUNTER`, which expects to receive this kind of events at a "wire" called `COUNTER_CLOCK_EVENT`, one would find the following line in the application's `.desc` file: `COUNTER:COUNTER_CLOCK_EVENT CLOCK:CLOCK_FIRE_EVENT`. Figure 2 also indicates that there is a dichotomy in the kinds of components in TinyOS: some are just software, while others are pieces of hardware which interact with software components according to the same component interface.

The TinyOS augmentations for the C language must be translated to standard syntax before the source code is passed to a C compiler. The translation process happens in several steps handled by compile time tools invoked from the application's `make` file. The most important of these steps are the translation of TinyOS

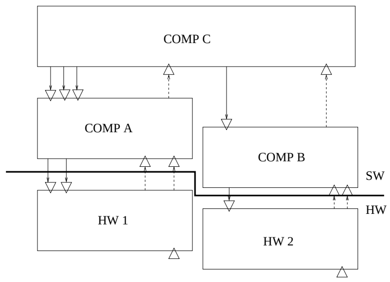

Figure 2: An Example of an Application Graph

keywords to standard C syntax and the wiring together of components. This latter step consists of the creation of a linkage header file which invokes the C preprocessor to do symbol substitutions mapping the endpoints of component's wires to one another. Clearly, in order to allow for the direct execution of TinyOS source code, a simulator will need to perform similar translations preserving the semantics from the original application while adapting translated commands to fit in the new framework. Providing the tools to accomplish the translations for a simulator entails a simple, although, unexciting task.

We have identified features of TinyOS, however, that are encouraging for the development of a scalable simulator for the execution of mote applications. First and foremost, it is interesting to note that the memory footprint of the simulated mote is very small. Clearly, the memory space occupied by *source code* of the mote application is the same whether one simulates a model with one mote or with hundreds of thousands of motes. Only one copy of the source code for each component needs to be kept in memory. This fact is exploited in TOSSIM (Levis 2002). On the other hand, the overall memory space occupied by component *frames* is proportional to the number of motes multiplied by the number of components per mote. Since motes have very small RAM space, however, one can expect that even in today's humble platforms such as a workstation with 512M bytes of memory, it would be possible to store large numbers of component frames for a simulation.

Second, the fact that the memory for each mote application is static and has its size defined at compile-time makes memory management in the simulator nearly trivial. Exchanges between components happen exclusively via commands or events, which deposit data in the frame of the callee. Component frames can be encapsulated by wrapper-classes in object-oriented programming and allocated prior to the start of the simulation.

Third and finally, the event-driven nature of the TinyOS programming model facilitates the construction of a simulator based on the discrete-event world view. If a kernel for discrete-event simulation, which can efficiently deal with very large numbers of events, is available, one of the hardest parts of the construction of the simulator is made easy. We have been working with such a kernel (DaSSF) and discuss it in more detail in the next section, where we also introduce a general purpose simulator for wireless ad-hoc networks (SWAN) which has been used to create a substrate for the implementation of a TinyOS simulator.

3 THE SIMULATION SUBSTRATE

The Dartmouth Scalable Simulation Framework (DaSSF) is an implementation of the SSF standard application programming interface for discrete-event simulation of large and complex systems (SSF API 1999). This concise API allows for high portability between compliant simulators and, furthermore, allows for the automatic parallelization of simulation models. DaSSF is one particular implementation of this API and employs conservative synchronization in the construction of a simulation kernel optimized for high performance when dealing with large models (Liu and Nicol 2001).

Since the SSF API was developed within the context of models for telecommunication systems, it lends itself naturally to the description of computer networks. The five classes defined in SSF, *Entity*, *Process*, *Event*, *inChannel* and *outChannel* have proved useful and powerful in the simulation of models for network protocols. Starting from these few classes, SSFNET, a comprehensive library of models for Internet protocols, was built and extensively used in large-scale simulations (Cowie, Nicol and Ogielski 1999a, 1999b). The SSFNET project provides a high-performance alternative to established simulators, such as *ns-2* and OPNET, which scales well with the number of nodes in the network model.

Our interest in investigating the use of self-configurable wireless networks in emergency response scenarios coupled with the expertise we accumulated in the simulation of Internet protocols lead us to develop a project similar to SSFNET, but focused on models of wireless protocols instead. To this end, in cooperation with BBN Technologies, we have constructed a framework in which we can execute simulation models of wireless networks in conjunction with detailed models that describe the environment where the network operates. We named this framework SWAN: Simulator for Wireless Ad-Hoc Networks (Liu et al. 2001a). Version 1.0, the first public release of the SWAN is available on the WWW (Liu et al. 2002).

The architecture of SWAN, illustrated in Figure 3, is composed of five classes of submodels. Each of these

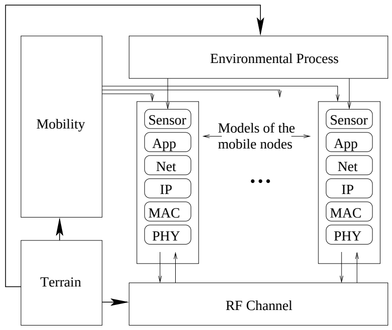

Figure 3: Component Architecture of the SWAN Framework

submodels is encapsulated so that instances can be removed or substituted by other instances from a library. The specification of all the submodels that compose a simulation resides in a configuration script which is fed to the simulator engine at execution time. This means that the code base does not have to be recompiled and submodels from the libraries provided can be easily plugged in or out of the overall configuration.

The first submodel worth of mention is that of the terrain, which affects how radio signals propagate, how sensed processes behave and how the nodes in the network move. As of now, only two simple terrain models are available in SWAN: one shaped as a rectangle and another shaped as a torus. The terrain models can be simply bi-dimensional or have an extra axis to measure elevation. We have on-going efforts to allow the modeler to enter arbitrarily complex descriptions of terrain features using file formats similar to TIGER, used in electronic maps by the USGS.

The RF propagation model is also of great importance in this framework. Among other goals, the simulation should be able to indicate the behavior and the performance of the network in realistic conditions. Radio connectivity needs to be modeled at a level of detail that is able to stress the salient features in the network design and at the same time be simple enough so as not to overburden the simulation with unnecessary computation (Takai et al. 2001). We have provided several variants of RF propagation model, so that different levels of detail are available to the simulationist, who can tailor the specifics according to the requirements of the experiments.

The environmental process, which interacts with sensors in mobile nodes, may or may not be affected by the terrain submodel. At this time, we have only implemented a simple model for the diffusion of gases on the simulated space, which is subject to a field specifying the direction and strength of air currents. The nature of the sensors which equip mobile nodes ultimately determines the nature of environmental processes and, as is the case with any other SWAN submodels, new specific environmental process can be easily added to the framework.

Finally, the last model which depends on terrain data describes the mobility of nodes. We have implemented a number of different kinds of mobility models such as those described in (Camp et al. 2002). It is currently not possible to mix nodes with different mobility models in the same simulation, although we intend to relax this restriction in future releases of SWAN. Building a simulator for TinyOS sensor networks on top of SWAN should perhaps require only one kind of mobility model: one where nodes are stationary (either with random or directed placement). It would seem that all the other mobility models built into SWAN would be wasted on TinyOS simulations because sensor nodes don't move. However, that is not the case. A research group at USC has proposed Robomote, a robot design based on Berkeley motes running TinyOS (Sibley et al. 2002). Simulations of Robomotes would benefit from the various mobility models that SWAN has to offer, especially one that allows the application to determine dynamically the direction and the speed of movement.

The description of the application code which runs inside each mobile node in SWAN is also highly modular. We have provided for a number of options commonly used to compose the protocol stack in wireless networks. To simulate Berkeley motes running TinyOS, however, none of these are of any help with the exception of the physical layer model. Since Berkeley motes use RF signals to communicate, we can reuse that component from SWAN as the foundation for the communication hardware model in a TinyOS simulator. In the next section, we discuss in detail how we constructed this simulator from SWAN's existing code base.

4 RUNNING TINYOS APPLICATIONS IN A SIMULATOR

Adapting SWAN to produce a simulator for TinyOS applications requires modifications that lie mostly within the model of a mobile node, which is shown as a stack of protocols in Figure 3. Part of the attraction for using SWAN as the foundation for this new simulator comes from the fact that SWAN offers the flexibility of model configuration at run time. In TinyOS, when an application is changed, part of the code has to be recompiled and a different executable is generated by the linker.

The first difficulty that arises when we try to implement a TinyOS application graph on a simulator comes from the fact that to achieve the same flexibility of model configuration found in SWAN, a lot more work needs to be done. Not only the list of components that makes up the application graph may change, but also the way they are wired together.

The solution we chose to give TOSSF the same flexibility of model configuration at run time found in SWAN breaks the applications as TinyOS would have built them, so that we can put them together again our own way. Rather than allow TinyOS' programming tools to wire components together by mapping symbols exported to symbols imported at compile time, we compile applications one component at a time. The commands and events exported by a component are *registered* with the simulator. This registration is achieved by identifying the exported functions in the source code and instrumenting them so that when constructors for the objects in the program are called, a string with the name of the function and a pointer to the function are passes to the simulator. TOSSF organizes this information in a lookup table which is used later on to perform the dynamic linking of components. We further instrument the component to fetch from TOSSF (during its initialization stage) the pointers to all the functions from other components that it will call.

More than just add flexibility to model configuration, with a little extra trickery, this scheme of dynamic linking allows one single copy of the object code for each component to reside in memory. Say that two different application types A and B need to be used in the same simulation model. If the intersection of the component types used in the two applications is non-empty, dynamic linking will guarantee that the memory footprint of the simulation model is optimal. If we were to blindly bundle up all components that constitute application type A, build an object-wrapper around this bundle and then repeate the same process for application type B, the components common to both types would end up in memory twice.

It turns out that what we do for the sake of optimizing the memory footprint of the simulation model and for the sake of enhanced flexibility in model configuration is a must when we deal with the allocation of storage for each component. Every different instance of the same component must have its own *frame*. Therefore, we need to ensure that when the code of a component executes, it has access to the correct instance of component frame. Again, we instrumented the TinyOS source code. This time we translated frame definitions to force components to register its frame type with TOSSF. During the initialization of the component, when the wave of `INIT` calls percolates through the application graph carrying along in the function call a unique identification for the mote to which it belongs, its own instance of the frame is allocated. This instance of the frame is also registered in a lookup table withing the simulator. All references to data in a component's frame via the `VAR` keyword are modified so that access to the data is made indirectly: first a pointer to that specific instance of the component is fetched from TOSSF, then the data can be accessed. Surely this adds overhead to the execution of the model by adding one level of indirection to every data access. We have yet to quantify this effect and analyze it in the light of its benefits.

A positive consequence of this dynamic linking scheme is that TOSSF can create application types on the fly and reuse them throughout the model initialization. When the configuration script for a model is first read in and a new application name is encountered, a corresponding application type is created in the form of a data structure containing a list of components and a wiring map that describes the application graph. From that point on, if the same application type is found again in the configuration file, all the simulator has to do is to allocate space for its component frames. Instances of the same application type share the same wiring map, so no memory is occupied with multiple replicas of the same information.

The configuration scripts used by TOSSF are very similar to those in SWAN. They are described in DML (Domain Modeling Language), which has a simple, but powerful grammar whose worth has been proved time and again in SSFNET models. The DML model script can be described as the composition of three distinct parts. First, the `ARENA` section defines models for mobility, node deployment and radio propagation for the length of the simulation run. Next, the `MOTE`s in the model are instantiated one at a time, defining for each one a unique identification and an application type. Last, at the end of the script, comes a dictionary of application types. Each application type in this dictionary is derived mechanically from the original TinyOS `.desc` files.

In the development of TOSSF, a substantial portion of time was spent in the construction of programs or scripts to instrument source code or convert configuration file formats. While the latter of these two tasks can be performed by simple Perl scripts, the transformations of component source codes proved a bit problematic. One crash at a time, we've discovered that this translation process can get complicated. The Perl script transforming the source code from TinyOS syntax to C++ had to be corrected a number of times and still does not cover all possible situations correctly. The right approach in this circumstance is to leave this work for a specialized tool such as a source-to-source compiler. As TOSSF matures, we expect to abandon our much hacked Perl script in favour of a better solution.

It is important to note that two facts drive us to automate these transformations on source code. First, from the perspective of the TinyOS programmer the learning curve in using TOSSF has a very sharp rise. The same set of files and programming tools used to produce the application for a real Berkeley mote is used to create the application for TOSSF. The programmer does not need to aquire any skill other than learning to write a very basic DML configuration script to define the scenario for the simulation and the motes which populate the simulation space. Second, the automation of this translation process will likely be much less error prone than if done by hand.

In addition to developing techniques to convert software components and applications for execution on TOSSF, we also had to develop additional SWAN models to represent the hardware components in the Berkeley mote platform, namely CLOCK, ADC, LEDS, RFM and UART. Actually, since we were not currently interested in simulating the interaction of motes with PCs via serial port, we have left the development of a UART model for later. On the other hand, we have developed software components that mimic the behavior of the other pieces of hardware.

Our model for the radio transceiver, the RFM component, is nearly identical to the physical layer in standard SWAN. The CLOCK component was also very simple to construct: it consists of a SWAN timer programmed with the data for resolution and time scale passed during the initialization of the CLOCK's FSM. The ADC, used to convert sensed data from analogue to digital format, was also simple to implement. The command to request data from the ADC spawns a SWAN timer that expires after a fixed delay and pushes the converted data out through the port specified in the request. Finally, the LEDS component which contains three LEDs in red, green and yellow, only has to receive commands to control individual diodes which are represented by boolean variables in the frame of the component. These components are made available for the construction of models of applications and the resulting architecture, illustrated in Figure 4 still bears much resemblance to the original SWAN from Figure 3.

When we consider the current design of TOSSF, we see that its main limitation regards the accuracy in estimates of the processing time for the applications executed. As of yet, TOSSF cannot deal with interrupts that would preempt an executing task. If a the simulation model of a hardware component flags an interrupt during the execution of a task, the corresponding event is queued to be handled when the task completes. Of course, in the real hardware, the event would take precedence over the task execution. In the simulation, the implementation of task preemption would require too much added complexity at a stage when we sought

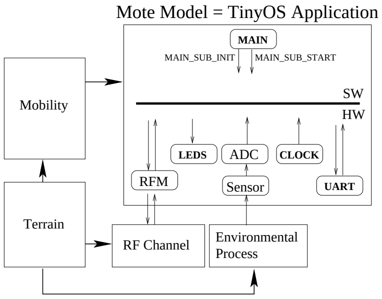

Figure 4: Architecture of the TinyOS Scalable Simulation Framework

a proof of concept and, therefore, has been postponed to a later time. In any case, one can argue that the timing error incurred in our scheme is very small since it is a fraction of the execution time of a task, which by construction should be very small. The precise time when a hardware interrupt is handled in the simulation is delayed by a very small (perhaps even negligible) factor.

This first version of TOSSF does take liberties with respect to the timing of certain operations for the sake of simplifying the simulation model. In addition to the absence of task preemption, TOSSF assumes that tasks execute instantaneously, that is, in zero simulation time. The time taken in the execution of commands and events is also neglected. The simulation clock is incremented only when events generated at the level of TinyOS hardware components are processed. Since communication latencies should dominate the execution time of the applications, we expect that the relative errors in timing should be small. However, we intend to improve on the current design in an attempt to increase the accuracy of timing in our simulations.

5 CONCLUSIONS

This paper presented concepts related to TinyOS, the operating system behind Berkeley motes, a computing platform capable of sensing and radio communication that can be seen as the precursor of Smart Dust. Our contribution with this work was to take SWAN, a high-performance simulator of wireless ad-hoc networks, and build upon it to develop an architecture for a flexible and scalable simulator for TinyOS applications. The TinyOS Scalable Simulation Framework, or TOSSF, resulted in a collection of additional models for SWAN that represent hardware components from the Berkeley motes. Also, TOSSF offers the programmer of TinyOS applications with a set of scripts that transparently adapt the source code for execution in the simulator. Through the use of dynamic linking of TinyOS components into working applications, we have managed to minimize the memory footprint of simulation models at the cost of a degradation of memory access times that still needs to be quantified. Finally, we indicate that there is still work to be done to enable TOSSF to produce faithful estimates of timing of execution of applications.

TOSSF still needs to mature before becoming ready for a public release. We expect that the demand for a simulator with its features is already great and that it keeps growing as more and more researchers look forward to performing scalability studies in the applications under development.

6 ACKNOWLEDGMENTS

The authors would like to thank Jason Liu and Michael Liljentstam, from the Institute of Security Technology Studies at Dartmouth College for the many insightful discussions which have aided the development of this project.

This research is supported in part by DARPA Contract N66001-96-C-8530, NSF Grant ANI-98 08964, NSF Grant EIA-98-02068, and Dept. of Justice contract 2000-CX-K001.

REFERENCES

Camp, T., J. Boleng, and V. Davies. 2002. Mobility Models for Ad Hoc Network Simulations. In *Wiley Wireless Communication & Mobile Computing (WCMC): Special Issue on Mobile Ad Hoc Networking: Research, Trends and Applications.*

Cowie, James, David M. Nicol and Andy T. Ogielski. 1999a. Modeling 100,000 Nodes and Beyond: Self-Validating Design. In *DARPA/NIST Workshop on Validation of Large Scale Network Simulation Models.*

Cowie, James, David M. Nicol and Andy T. Ogielski. 1999b. Modeling The Global Internet. In *Computing in Science & Engineering*, 1(1):42–50.

CrossBow. `http://www.xbow.com` [Accessed March 10, 2002]

Estrin, Deborah, David Culler, Kris Pister, and Gaurav Sukhatme. 2002. Connecting the Physical World with Pervasive Networks. In *IEEE Pervasive Computing*, 1(1): 59–69.

Hill, Jason, Robert Szewcyk, Alec Woo, David Culler, Seth Hollar, and Kristofer Pister. 2000. System Architecture Directions for Networked Sensors. In *ACM Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, USA.

Levis, Philip. 2002. TOSSIM System Description. `http://webs.cs.berkeley.edu/tos/` [Accessed March 15, 2002]

Liu, Jason, David M. Nicol, L. Felipe Perrone, and Michael Liljenstam. 2001. Towards High Performance Modeling of the 802.11 Wireless Protocol. In *Proceedings of the 2001 Winter Simulation Conference*, ed. B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, 1315–1320. Piscataway, New Jersey: Institute of Electrical and Electronic Engineers.

Liu, Jason, and David M. Nicol. 2001. The DaSSF 3.1 User's Manual. `http://www.cs.dartmouth.edu/research/DaSSF/` [Accessed March 3, 2002]

Liu, Jason, L. Felipe Perrone, David M. Nicol, Chip Elliott, and David Pearson. 2001. Simulation Modeling of Large-Scale Ad-hoc Sensor Networks. In *Proceedings of the 2001 European Simulation Interoperability Workshop*, London, England.

Liu, Jason, Yougy Yuan, Michael Liljenstam and L. Felipe Perrone. 2002. SWAN: Simulator for Wireless Ad-Hoc Networks. `http://www.cs.dartmouth.edu/research/SWAN/`

Sibley, Gabriel T., Mohammad H. Rahimi and Gaurav S. Sukhatme. 2002. Robomote: A Tiny Mobile Robot Platform for Large-Scale Sensor Networks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA2002).*

Scalable Simulation Framework (SSF) API Reference Manual `http://ssfnet.org` [Accessed March 3, 2002]

Takai, Mineo, Jay Martin, and Rajive Bagrodia. 2001. Effects of Physical Layer Modeling in Wireless Ad Hoc Networks. In *Proceedings of The ACM Symposium on Mobile Ad Hoc Networking & Computing 2001 (MobiHoc).*

TinyOS Programming Bootcamp. Available online via `http://webs.cs.berkeley.edu/tos/presentations/Boot_Camp/` [Accessed February 25, 2002]

Warneke, Brett, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. 2001. Smart Dust: communicating with a cubic-millimeter computer. In *Computer* 34(1): 44–51

Woo, Alec, and David Culler. 2001. A Transmission Control Scheme for Media Access in Sensor Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (Mobicom)*, Rome, Italy.

Warneke, Brett, Bryan Atwood, and Kristofer S. J. Pister. 2001. Smart dust mote forerunners. In *The 14th IEEE International Conference on Micro Electro Mechanical Systems (MEMS 2001)*, 357–360.

Ye, Wei, John Heidemann and Deborah Estrin. 2002. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, New York, NY, USA.

AUTHOR BIOGRAPHIES

LUIZ FELIPE PERRONE is a Research Associate with the Institute for Security Technology Studies and a Visiting Lecturer at the Department of Computer Science at Dartmouth College. He holds the degrees of Electrical Engineer and M.Sc. in Systems Engineering and Computer Science from the Federal University of Rio de Janeiro, in Brazil, and Ph.D. from the College of William & Mary. His research interests include modeling and simulation of wireless systems, network security and parallel discrete-event simulation. He has served as vice-program chair of *The 9th International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2001).* He is a member of the IEEE Computer Society. His e-mail address is `<perrone@ists.dartmouth.edu>`.

DAVID M. NICOL is Professor of Computer Science at Dartmouth College, and is presently serving as Associate Director of the Institute for Security Technology Studies, responsible for Resesarch and Development. His research interests include network security, parallel processing, and simulation/modeling; he is Editor-in-Chief of the ACM Transactions on Modeling and Computer Simulation, and is a Senior Member of IEEE. He holds a B.A. degree in mathematics from Carleton College, M.S. and Ph.D. degrees in Computer Science from the University of Virginia. His e-mail address is `<nicol@ists.dartmouth.edu>`.