TYTHON: A DYNAMIC SIMULATION ENVIRONMENT FOR SENSOR NETWORKS

Michael Demmer, Philip Levis, August Joki, Eric Brewer, David Culler

University of California, Berkeley
Computer Science Division
Berkeley, CA 94720

ABSTRACT

We present Tython, a Python-based scripting extension to TinyOS's TOSSIM simulator. Tython includes a rich library of scripting primitives that enable users to describe dynamic but reproducible simulation scenarios. We take advantage of TinyOS' event driven execution to allow users to attach script callbacks to particular simulation scenarios. Scripts can also use interfaces at both a whole-network and a per-mote level to analyze and affect behavior in response to changes in the environment. We use the development of a Pursuer-Evader application as a running example to demonstrate the value of this approach.

1. INTRODUCTION

Programming sensor network systems requires developers to design distributed and robust algorithms, implement them for resource-constrained motes, and test or debug them with a limited ability (e.g., LEDs) to inspect program execution. These challenges have led to the development of several sensor network simulators, such as TOSSIM[1], EmStar[2], and atemu[3]. Simulation can speed up the development cycle; the embedment and large scale of mote networks, as well as their limited network bandwidth, lead to a long and limiting install-test-debug cycle (sometimes on the order of hours). Simulation provides repeatable experimental conditions; in deployment, a combination of high concurrency, reactive behavior, and uncontrolled environmental input leads to unforeseen system conditions and interactions. Finally, simulation allows users to inspect and control execution more easily and at a much higher fidelity than possible with a collection of hundreds of tiny computers dispersed in space.

At the same time, sensor networks introduce novel requirements for a simulator. A sensor network's behavior depends on how nodes perceive and interact with the dynamic and uncontrollable real world. Applications and protocols can be sensitive to minor variations in environmental effects, such as radio interference or sensor input. Although existing simulators can model program execution at a wide range of fidelity and scale, they mostly ignore the stimuli that drive that execution. Systems such as EmStar address the environment by introducing a hybrid mode, where simulation code interacts with real-world motes; although this approach allows testing simulated applications based on real data, it does not support repeatable, controlled experiments.

Experience in our research group developing a pursuer-evader game (PEG) TinyOS [4] application demonstrates some of the issues that developers face. In this application, a field of motes senses moving vehicles by periodically sampling magnetometers and smoothing the readings. If the resulting reading is over a threshold, a node broadcasts the observed value to nearby neighbors. The node with the highest reading (the implicit leader) aggregates all other received readings into a single packet and sends it to a pursuer robot.

After an initial implementation on real motes, we ported the application to the TOSSIM simulator. Simulating the TinyOS application code allows developers to test code and algorithms with static routing points. However, TOSSIM alone proved to be insufficient, as fully testing and analyzing this application requires modeling how sensing, aggregation, and routing behave as pursuer and evader robots dynamically move through the sensor field.

To effectively simulate this and many other applications, the TOSSIM framework requires two additional capabilities:

Reproducible Interactivity: Environmental dynamics and noise define a large space of possible system effects. Programs need to be exposed to as much of this space as possible to find error cases, and any given scenario should be reproducible. Finding a race condition by introducing interesting sensor readings is useful; being able to verify that the problem is fixed by reproducing the situation is more so.

Flexible Environmental Models: Extreme deployment environments such as building structures, volcanos, and bird habitats are a common application requirement for mote networks. Correspondingly, although common environments such as an open field or an office may be suitable for some applications, developers must be able to introduce their own environmental models, either by building on or replacing those already available, and the domain of expressible models should be as large as possible.

This paper presents Tython, an extension to the TOSSIM simulator for testing and analyzing TinyOS applications. Tython addresses these key requirements with two main contributions. First, Tython integrates an implementation of the Python[5] scripting language with a set of primitives and structures used to control and interact with a TOSSIM simulation. This full-featured programming environment enables repeatable experimentation with fine-grained execution control.

Second, Tython implements a rich set of internal abstractions and library routines to model and control the environmental effects on a simulation. These include a programmable distance-based radio interference model as well as a flexible mechanism for representing the inputs to real-world sensors. Through the combination of reproducible interactivity and environmental modeling, Tython is a powerful tool for the sensor network application developer.

In the next section, we review existing sensor network simulators and their capabilities. In Section 3, we outline the TOSSIM simulator framework and how Tython uses and extends that framework to provide reproducible interaction and flexible environmen-

tal models. We present several use cases in Section 4 to demonstrate the kinds of problems Tython seeks to solve and how it solves them. Finally, we discuss several areas for future work and conclude in Section 5.

2. RELATED WORK

Given that Tython is based on TOSSIM, much of the related work analysis in [1] applies directly to this work as well. We present the aspects of TOSSIM's design that are pertinent to Tython in Section 3.1. Here we present how several other sensor network simulators relate to Tython and motivate its particular capabilities.

`ns-2` [6] is a widely used event simulator for network systems, and many experiments have leveraged it to examine protocols and distributed systems. `ns-2` integrates the Tcl scripting language into the tool, and therefore offers similar flexibility and determinism as in Tython. The main limitation with `ns-2` is that it is exclusively a simulation framework; a protocol authored for `ns-2` must be reimplemented to test in deployment. As the Tython/TOSSIM environment runs identical application code in simulation and in deployment, a particular implementation can be examined in simulation as well on actual hardware, and multiple implementations can be compared in terms of code complexity, size, and execution time.

EmStar [2] is a flexible environment for developing sensor network applications, originally intended for iPAQ-class devices, and recently extended to include support for TinyOS based applications [7]. The EmStar framework enables distributed application executions on physical nodes, simulated executions within a single application on a node, or a hybrid environment in which "real" nodes can communicate with simulated nodes. Although the framework includes many utilities for managing this process execution, the distributed nature of the execution precludes certain control capabilities that are possible in Tython due to the tight synchronization between the execution of the program and the simulation of the environment. An interesting future avenue of research would be to integrate the Tython execution control framework to interact with nodes running under the EmStar framework.

TOSSF [8] is a simulation system that compiles a TinyOS application into the SWAN [9] simulation framework. The primary focus of TOSSF is on scalability, whereas Tython is targeted more for flexibility and ease of use. As such, TOSSF does not provide a scripting framework for experimentation, hence suffers from a potentially long test-debug cycle. Also, although TOSSF also enables development of custom environmental models, the lack of scripting support requires those models to be compiled into the simulation platform.

3. THE TYTHON FRAMEWORK

3.1. TOSSIM Background

Tython uses TOSSIM as its simulation core. TOSSIM compiles whole TinyOS programs, written in nesC [10], into its discrete event simulation framework. This allows a developer to use a desktop and existing development tools, such as debuggers, to test and analyze the same code that will run on motes. Internally, the TOSSIM event queue models the execution of a number of nodes with very precise timing. This precision, combined with controllable random seeds, provides a deterministic execution environment and repeatable experiments.

TOSSIM does not provide any direct interface for interacting with a running simulation. Instead, it implements a network protocol by which interactive applications can send commands and receive events from the simulation. This communication channel is synchronous: when TOSSIM sends an event to a registered application, it blocks until it receives an acknowledgment packet. This behavior is critical to allow an interacting application to deterministically affect execution: the application can execute commands synchronously in response to an event.

TOSSIM includes a Java-based graphical tool, TinyViz, which interacts with TOSSIM through its network protocol. TinyViz allows users to select and manipulate simulated nodes through a GUI, and load pre-compiled "plug-ins" for additional functionality. Although TinyViz allows a user to interact with a simulation, it has two major limitations. First, a GUI is an inherently unrepeatable interface: dragging a GUI-based robot in exactly the same way and speed twice is not an effective experimental methodology. Second, although Java code in plugins can create repeatable effects, the range of control is defined when TinyViz boots: a user cannot, for example, easily change how a plugin-based robot moves without starting a new simulation.

3.2. Tython Scripting Support

Tython extends TinyViz by incorporating a Java based implementation of the Python scripting language called Jython [11]. In addition to TinyViz's standard GUI interaction, users can interact with a simulation through a console or pre-loaded scripts. Jython empowers the developer with a full featured programming language with a rich library of control primitives, library objects and built-in data structures. Furthermore, Jython includes a powerful reflection capability to expose Java objects to the scripting environment and vice versa. The use of Jython also enables both an interactive console and file-based scripting capabilities.

Tython leverages this reflection to provide a set of abstractions for the various entities within the simulation framework. These abstractions are implemented as Java classes and provide the interface for controlling the simulated motes, other "objects" within the system, environmental effects, and the simulation engine itself. For example, `motes[3].turnOff()` signals the simulator to turn off mote 3, and `sim.pause()` pauses simulation.

Additionally, because Tython provides a full fledged scripting language, functions can use these low-level interfaces to encode higher-level abstractions. For example, at the lowest level, Tython allows a script to set the ADC value of a particular ADC port on a mote. Some scripts used in the PEG simulations model a vehicle source by setting the magnetometer ADC values for nearby motes based on their distance. Our experiences with situations such as these led us to develop additional, higher level sensor abstractions, which we discuss in Section 3.6.

Tython also extends the TinyViz event dispatching mechanism to the scripting environment. This enables a script to be synchronously notified when a particular operation occurs in one of the simulated motes, such as a radio transmission or LED blinking. For example, a script can trigger when a particular mote has been elected leader, or even when two motes very close to one another are both elected leader. This latter example represents a case in which packet loss has caused the leader election to operate sub-optimally, or it could be due to a bug in the leader election algorithm; determining which is the case is useful to a developer.

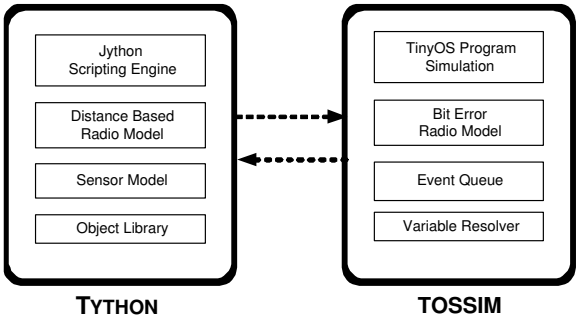

Fig. 1. Tython / TOSSIM Design

3.3. Tython / TOSSIM Boundary

Figure 1 depicts the functionality division between Tython and TOSSIM. The large enclosing boxes represent the process boundary, and as mentioned previously, the two processes communicate over a network protocol. One way to characterize this functionality division is as the boundary between the simulation of environmental effects on a system and the execution of the system code itself.

The process separation in this architecture offers several benefits: First of all, the command/event protocol codifies the full set of effects that the environment can impart on the system execution. Secondly, keeping potentially complex environmental modeling out of TOSSIM helps to maintain the simplicity of the TOSSIM code base, improving both scalability and reliability. Finally, this separation allows for more implementation flexibility, as evidenced by the use of Java to implement Tython, while TOSSIM is implemented in nesC.

TOSSIM's synchronous event dispatch model enables this process separation without losing repeatability. As such, even though Tython and TOSSIM run in separate execution threads, the network protocol provides the necessary synchronization and ties the two executions together without any race conditions.

3.4. Tython Radio Model

As described elsewhere [1] TOSSIM implements a radio channel model that assigns a bit error probability to each "link" between a pair of nodes. Although this model is flexible, it does not naturally express environmental effects such as distance on the radio channel. To address this shortfall, TinyViz implements a distance based radio model, where bit error probabilities are derived from the motes' locations in a virtual coordinate space. The actual loss rates come from a small set of predefined models, such as a unit disc or distance based distributions derived from empirical data.

This approach allows users to quickly lay out a physical topology for experiments, but is also very limiting. First, adding new radio models requires either modifying TinyViz or writing new Java classes that TinyViz loads when it boots: either case requires quitting and restarting TinyViz. Second, direct user interaction is neither precise nor repeatable, for example, users click and drag motes to move them to new positions. Enabling users to control these variables dynamically and repeatably as a simulation runs leads to a more powerful development environment.

Tython extends the TinyViz model, providing scripting interfaces to "move" motes around in the virtual space and manipulate the current propagation model. The scripting interface also exposes facilities that disable the distance-based calculations and directly set the loss rate probability between any pair of nodes, via the`radio.setLossRate()` function.

As one example in which the direct interface may be more appropriate, several studies have gathered comprehensive trace data of sensor network deployments in complex environments [12, 13, 14]. Within the Tython framework, the connectivity characteristics of a particular deployment could be extrapolated from a trace and then used as input to the radio model. This, in essence, would program the simulated environment to mirror the effects observed in the real deployment, and allows developers to develop protocols that address the exact observed phenomena from a particular application deployment.

As we gathered network traces from the initial deployment of the PEG application, another avenue for exploration would be to use these traces to drive the radio loss model within Tython. This could allow us to examine the causes of some observed misbehaviors in deployment, as well as to examine the behaviors that result from the particular radio characteristics observed in the field.

3.5. Objects in the Environment

While some sensor network applications are intended to monitor ambient environmental effects such as temperature or humidity, other applications are intended to interact with or respond to other physical objects in the environment. In many cases, the position of this object in space relative to the nodes may influence the sensor readings on the nodes themselves. For example, a photo sensor will respond to a nearby flashlight that is shining on it, and a radio transmission may be obstructed by a barrier.

To accommodate these effects, Tython exposes an interface through which non-mote objects can be incorporated into a simulation experiment. This mechanism enables a natural framework for evaluating the effects that these objects can have on the system execution. For example, a wall object could be added to the simulation through the scripting interface, and a radio propagation model could learn about the existence of the wall object and take it into account when calculating a loss rate, e.g. by setting the loss rate to 95% for any pair of nodes that lie on opposite sides of the wall. These objects are invisible to TOSSIM, which only observes their effects in terms of mote interactions and sensing (e.g., radio bit error rates, ADC settings).

3.6. Sensor Model

TOSSIM exposes a simple API hook to directly set the value of a particular analog to digital converter (ADC) port and thereby simulate a sensor input. As in the case of the radio model, Tython both exposes this direct interface and also implements a higher level model for sensor readings. To that end, we implemented object interfaces for the following abstractions:

Sensor Field: a real world phenomenon that is to be sensed, such as, light, magnetism, or temperature; each field is mapped to a specific ADC port within the simulation.

Sensor Source Object: a mote or another simulated object with some intrinsic intensity for a particular sensor field; for example, a light source will have some luminescence, and a metal object has a certain magnetic attraction.

Sensor Propagation: the model for signal attenuation as a function of the distance between a sensing mote and the source, e.g. a fixed diameter "disc" model, a "linear" model in which the

value declines as a proportion of the distance, or more complicated custom models.

Sensor Combination: the interface used to combine the values from individual sensor values to obtain an aggregate reading that is set as the ADC port value, either additive, multiplicative, or again, a custom combination module.

In general, we designed this sensor model framework to be a simple framework for representing common scenarios based on our experiences using Tython for applications such as PEG. At the same time, the model is flexible and extensible, allowing it to evolve over time for new hardware platforms or environments.

3.7. Library Routines

In addition to exposing the internal state of the simulation environment through Java reflection, the Tython framework also includes a rich library of utility objects. For example, one object exposes the random number generator used within Tython/TOSSIM for use within scripts, another manages periodic callbacks, and another performs step-wise movement of an object with various mobility patterns (e.g. linear, random walk) in the virtual space.

For Tython to accurately model phenomena over time in a simulation, it must have a way of knowing when the simulation reaches a certain time. We extended TOSSIM to include a *time reached* command, which enqueues an event on the TOSSIM queue at a specified simulation time. As with all other events, when TOSSIM handles the event, it signals Tython, so Tython can manipulate the environment as it needs to. Using this functionality, scripts can implement "future actions" and periodic callbacks that are synchronized with the simulation execution, an essential characteristic for repeatable simulations.

3.8. Simulation Variable Resolution

Another feature enabled by the Tython environment is to expose the ability to read variables directly from the simulated executable. To enable this feature, we made some simple modifications to the nesC compiler to generate a variable resolver function, much like a limited form of debugging symbols. Through this mechanism, Tython can resolve address and length of a variable from TOSSIM, and subsequently retrieve the variable's value from memory.

This facility is most useful for debugging, as it allows interactive access to the simulated program state of all nodes. For example, a script in the PEG application can print out which mote(s) are "leaders" as follows [1]:

```
for m in motes:
  if m.getByte("MutationRoutingM$leader"):
    print m, "is a leader"
  else:
    print m, "is not a leader"
```

One potential extension to this system would to add variable writing capabilities as well. This would allow, for example, a test script to force multiple nodes to be leaders and determine how the PEG routing protocols handle that situation. Clearly, this functionality must be used carefully, as it could easily corrupt mote and simulator state.

[1]Some basic Tython statements and syntax have been omitted from this and future examples in the interest of brevity and clarity.

4. EXAMPLES / USE CASES

To illustrate the advantages of the Tython system, this section outlines some use cases for which we have designed the Tython system and some examples of its utility.

4.1. Basic Scenario Validation

A basic use case for Tython is as a development framework for debugging and evaluating an implementation that depends on environmental dynamics. For example, Tython enables easy testing of PEG's magnetometer filtering and local aggregation. A developer can introduce simulated magnetometer sources, noise, or spurious readings, and analyze how the application performs. Unfortunately, as our development and testing of this part of the PEG demo motivated much of Tython, it also preceded it: by the time Tython was ready for use, this part of PEG was well tested and stable, after many hours of work and deployment.

To demonstrate the benefits of Tython, we step through how the PEG developers could have used it to simplify testing and development.

To start off, the script would distribute 100 motes in a grid covering a one hundred foot square region. This will also have the effect of updating the distance based radio model to reflect the connectivity of this distribution:

```
for x in range(0, 10):
  for y in range(0, 10):
    m = motes[x + (y * 10)]
    m.setCoord(x * sim.worldWidth / 10,
               y * sim.worldHeight / 10)
```

Next, the script creates a sensor propagation model that attenuates linearly from 5 feet to 20 feet and adds it to the system:

```
model = LinearSensorModel(5.0, 20.0)
sensor.addModel("magmodel", model)
sensor.addField("mag", port, "magmodel")
```

The script then creates an evader object (the sensor source object) and attaches a magnetism attribute with value 100:

```
evader = sim.newSimObject()
attr = SensorAttribute("mag", value=100)
evader.addAttribute("mag", attr)
```

Finally, using the `ObjMover` utility class, the script moves the object to the other side of the field of motes in one foot increments:

```
evader.setCoord(0, 20)
objmover.moveTo(evader, 1, 100, 40)
```

To illustrate the effects of this example, we tracked the ADC input value on a row of the simulated motes in the grid whenever a periodic sampling timer fires.

As expected, the plot of sensor values in Figure 2 confirms the expected results. For the motes that detect the object, their sensor readings climb linearly as the evader approaches the mote, plateau at full strength if the evader comes within 10 feet, then fall as the evader moves away.

This simple example illustrates the flexibility of the Tython scripting API as well as its ability to represent real-world situations in a natural and easy to use manner.

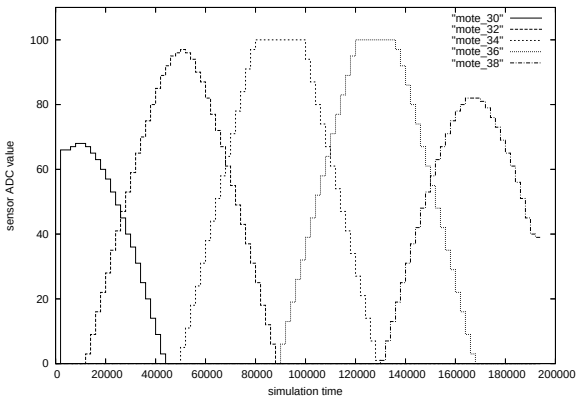

Fig. 2. Sensor readings from the example script

4.2. Network Protocol Development

The TinyOS toolchain includes Java-based utilities for generating classes representing TinyOS packets. Jython's Java reflection allows Tython to seamlessly integrate all of these tools. Instead of compiling and writing Java classes to inject packets and monitor network traffic in TOSSIM, users can do so more easily with scripts.

One early adopter of the Tython framework uses it to develop and evaluate an any-to-any mote routing protocol based on logical coordinates [15]. A simple experiment first initializes a topology, and then evaluates the overall connectivity achieved by the protocol by iterating over the N nodes in the system, and for each node, injecting a command packet with an instruction to try to send a message to every other node. The script also registers an event handler to receive all packets that are sent by the system, and thereby can compare the observed behavior versus the expected behavior.

To evaluate the protocol's recovery from transient network problems, the developer has run the same experiments while introducing node failures. This tests how temporary disconnections or failures affect end-to-end delivery and how well the protocol can react to temporary disconnections that are contrary to longer term link estimators. In general, the greatly enhanced debugging and introspection support provided by the Tython system has been extremely valuable tool for this development.

In the case of the PEG application, injecting and monitoring network traffic allows a developer to script how a user might interact with a deployment. For example, normally a deployed network is in a deep sleep mode, and a user wakes up the network by sending a few special packets through a base station. Tython scripts can recreate this interaction in a repeatable way.

4.3. Edge Condition Coverage

A widely accepted principle in systems design is that code to handle exception cases is the hardest code to get right [16]. This follows from the fact that exception handling code is hard to test because the situations which trigger that code are rare and hard to create. Furthermore, it may be very difficult or even impossible to test such code in an actual deployment since the cause of the exception may be something irreversible like a hardware component failure.

An example of one such situation in the PEG applications is the existence of a faulty sensor. This failure could be due to low batteries, a loose connection, or faulty hardware. Because nodes automatically send sensor readings if they are above a threshold, a single faulty sensor can continuously claim it detects a moving vehicle when none exists. Theoretically, the magnetometer filtering code should take care of this case (it automatically adjusts magnetometer gain to find a baseline noise floor), but this was never tested in the real world. For a short term deployment, the general technique is usually to just replace the bad sensor. Long term deployments, however, do not have this luxury.

Another example is is the case of one-way radio "connectivity". It has been shown in empirical studies [17] that deployments of a field of wireless sensor nodes will in general have pairs of nodes in which communication is one-way. Networking protocols should be designed to handle these situations, yet it is challenging to reliably set up scenarios of one-way connectivity in a lab testing environment.

Testing both of these scenarios in Tython is simple. For the former, Tython controls the input to the magnetometer and can easily simulate faulty readings. For the latter, using the direct radio interface, a script can set a 99% loss rate on packets transmitted from mote 1 to mote 2, and a 0% loss rate on packets in the reverse direction as follows:

```
radio.setLossRate(1, 2, 0.99)
radio.setLossRate(2, 1, 0.00)
```

These examples both show the benefits of a full application simulator. Testing filtering techniques and routing protocols in controlled experiments, not simply concluding "it seems to work," can vastly help improve application performance and reliability.

4.4. Parameter Exploration

Many algorithms and protocols depend on a set of parameters that affect how the system operates in a given situation. Experimentation can help determine the best settings for a given input scenario. Iterating through a large parameter space in deployment or on a test bed can quickly become infeasible due to the burdens of reprogramming motes. Additionally, the input to a real world network is unrepeatable And can change between experiments. In contrast, controlled input in simulation allows a developer to separate and understand the causes of behavior, rather than conflate parameters with the environment.

Since Tython implements the ability to run, pause, resume, and stop a TOSSIM simulation, a script can easily iterate through a parameter space. For each setting, it is trivial to run an experiment, gather results, then reset the simulator and try the next value. This process is easily amenable to unattended batch simulations

Furthermore, since the scripting framework is itself a programming language, this exploration need not be static. For example, the above technique can run independent experiments for each parameter value with different random seeds. The number of experiments can be determined dynamically, to achieve a desired confidence interval in the results.

4.5. Protocol Comparisons

Many research papers use simulation frameworks to compare a set of protocols. For example, Broch et al. [18] present a comparison

of several mobile ad-hoc routing protocols based on simulations in `ns-2` [6]. One of the significant points that their paper makes is that the different protocols are highly sensitive to the particulars of a mobility scenario. To provide a fair comparison, they pre-calculated a set of candidate mobility scenarios and then ran each protocol on the same set of scenarios, thus reducing the biases of an individual mobility pattern.

The Tython framework easily enables these types of experiments. The built-in Python libraries include primitives for reading and parsing files, allowing a set of scenarios to be pre-computed and stored for subsequent execution. For this type of study, it is critical to ensure that all aspects of the system remain constant across the comparisons, except of course the program code being evaluated. Because the Tython framework is intimately tied to the TOSSIM event loop, Tython ensures that the experimental setup is identical among protocol comparisons.

4.6. Simulating Sensors and Actuators

Tython has proven to be an invaluable tool in the experimentation and evaluation of the PEG application. Some of these uses are discussed above, such as triggering a route discovery flood by injecting a control packet to the intended root of the routing tree, monitoring the location of the evader to effect magnetometer readings, and using the scripting capabilities to iterate repeatedly through a set of experiments.

In addition, a key use of Tython in the PEG application was to emulate sensors and actuators that are not provided by the core TOSSIM environment. Since TOSSIM does not provide a model of radio-signal strength, the PEG scripts monitored radio traffic and converted the packets into radio strength measurement values based on the sender's location. These values are then set as ADC port readings on the simulated motes, thereby simulating a hardware feature based on packet transmissions in the simulation.

Similarly, as TOSSIM also does not have a model for simulating robotic vehicles, Tython provided that functionality. In this case, Tython acts as the implementation of actuators for velocity and direction, responding to instructions (in the form of debugging messages) by moving the simulated object in the virtual space.

As these two examples show, a useful feature of Tython is to emulate the behavior of sensor and actuator systems that are not provided by TOSSIM itself.

5. CONCLUSIONS

Programming sensor network systems is and will likely remain a challenging task. A primary goal of simulation platforms is to help to alleviate these burdens. For wireless sensor network systems, two features of simulators are extremely valuable: reproducible experimentation and dynamic environment modeling.

Tython adds these two key features to the TOSSIM simulation engine for TinyOS programs. Tython implements a Python based scripting framework that enables flexible and repeatable interactions with the underlying simulation. Through extensible models for sensor propagation and radio channel losses, Tython exposes a framework for representing the effects of a dynamic environment on a system's execution. Some early experiments with the Tython environment in a pursuer-evader application have verified the core benefits of the framework as well as informing the design of certain key aspects, such as the sensor abstraction model.

Tython is a functional framework with many users worldwide; however, there are several ways in which it can be improved, some of which are being actively working on. The most important is the command/event interface. Although TOSSIM's synchronous interface allows reproducible dynamics, it can impose a significant overhead as it generally leads to four context switches for each event. We are working to relax this aspect of the system: while critical events and commands (decided by the scripting side) will remain synchronous, others can be delivered asynchronously. An initial implementation of this type of functionality allows a script to batch a set of commands into a single synchronous exchange; this turns out to have a major effect when Tython modifies state such as the radio connectivity, which can require up to n^2 commands. We expect these developments to be part of the next major TinyOS release.

Tython is an integral part of TinyOS distributions 1.1.4 and later, and is available at `http://www.tinyos.net/`

6. REFERENCES

[1] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proceedings of SenSys*, June 2003.

[2] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin, "Emstar: a software environment for developing and deploying wireless sensor networks," in *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004.

[3] "atemu - Sensor Network Emulator / Simulator / Debugger," `http://www.cshcn.umd.edu/research/atemu/`.

[4] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister, "System architecture directions for networked sensors," in *Proceeedings of ASPLOS 2000*, Boston, MA, USA, Nov. 2000, pp. 93–104.

[5] "The Python Programming Language," `http://www.python.org/`.

[6] "The Network Simulator," `http://www.isi.edu/nsnam/ns/`.

[7] Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson, Deborah Estrin, Eric Osterweil, and Tom Schoellhammer, "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004.

[8] Luiz Felipe Perrone and David Nicol, "A Scalable Simulator for TinyOS Applications," in *Proceedings of the 2002 Winter Simulation Conference*, 2002.

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

[10] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[11] "Jython," `http://www.jython.org/`.

[12] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler, "Lessons from a sensor network expedition," in *Proceedings of EWSN*, January 2004.

[13] Alberto Cerpa, Naim Busek, and Deborah Estrin, "SCALE: A tool for simple connectivity assessment in lossy environments," Tech. Rep. CENS-21, UCLA, 2003.

[14] Jerry Zhao and Ramesh Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in *Proceedings of SenSys*, 2003.

[15] Rodrigo Fonseca, Sylvia Ratnasamy, David Culler, Scott Shenker, and Ion Stoica, "Beacon vector routing: Scalable point-to-point in wireless sensornets," Tech. Rep. IRB-TR-04-012, Intel Research Berkeley, May 2004.

[16] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *Proceedings of SOSP*, 2001, pp. 57–72.

[17] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker, "An empirical study of epidemic algorithms in large scale multihop wireless networks," 2002, Submitted for publication, February 2002.

[18] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," in *Proceedings of Mobicom*, 1998.