Wireless Sensor Network Simulation Using Tossim

by

Alireza Mortezaei

4B Software Engineering, University of Waterloo

Supervised by

Dr. Mahesh Tripunitara

April 26, 2012

Summary

The main focus of this literature is the simulation of wireless sensor networks. One the major purposes that this simulation serves is the performance analysis of certain security schemes specific to wireless sensor networks. These security measures were introduced in the paper *Query Privacy in Wireless Sensor Networks* by Carbunar et. al. Their paper is concerned with the problem of preserving the privacy of clients querying a wireless sensor network owned by untrusted organizations. It offers several strategies to deal with this issue for specific use case scenarios. However, it does not fully explore the performance boundaries of these proposed strategies. Using these strategies may result in some performance overhead as its traditional performance versus security tradeoff. We are interested to find the exact amount of the overhead imposed by these methodologies and thus find the exact performance boundaries of these methods. However, to justify our claims about these boundaries we need to collect relevant data to further support the results of our findings. This creates a need for a tool that can simulate the wireless sensor networks specific to our needs. Throughout this paper we will explain how this tool is developed and how it will simulate the network.

Contents

List of Figures

List of Blocks

1 Introduction

1.1 Background

1.1.1 Wireless Network Sensors

Wireless Sensor Networks (WSN) are a new breed of distributed systems. The special feature of WSNs is that they are an integrated part of the environment they inhabit and are designed to closely monitor their surroundings. They are a step closer to eliminating the gap between the physical and the virtual world. These networks are easily scalable, cost efficient and can obtain types of data that would be impossible to obtain using traditional approaches, for example, data obtained from hard-to-reach areas. WSNs have a wide range of applications in military, environmental science, health science, crisis management etc. WSNs consist of many sensor nodes called motes. These motes collect data and communicate with one another to provide a real-time data feed. These motes can often self-organize after being deployed in an ad hoc fashion[1]. WSN technology currently is at its infancy stage and there are many aspects of these systems such as security that still needs addressing. Since the architecture of these networks is drastically different from computer networks, it is difficult to directly employ the existing security approaches to the area of WSNs. The networks are fragile with respect to privacy measures. In the paper *Query privacy in wireless sensor networks*[2], several strategies that address the problem of preserving the privacy of clients querying a wireless sensor network owned by untrusted organizations are offered. However, the paper does not fully explore the performance analysis of these strategies and the amount of the

performance overhead imposed by these privacy measures is unknown. The goal of our project is to explore this "security versus performance" tradeoff and find exact performance boundaries of these strategies.
One of the main steps towards analyzing the performance of such strategies is to simulate them in a network using a simulation tool.

1.1.2 TinyOS

In order to build a WSN simulator we used TinyOS. TinyOS is a free and open source component-based operating system. It is a platform that targets WSNs. In essence, TinyOS is an embedded operating system which can be configured, built and deployed onto several wireless boards. Devices in WSNs have strict power and memory constraints and TinyOS is specifically designed with those restrictions in mind. The essential characteristic of TinyOS is its event-driven architecture. Its event-driven execution model enables fine-grained power management and yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interface.
TinyOS is written in the nesC programming language as a set of cooperating tasks and processes. nesC closely resembles C programming language with the difference that it is a component-based dialect of C. Similar to objects, components are discrete units of functionality and they encapsulate state and couple state with functionality[3].
Each component specifies what functions it provides and what functions it uses. The *provide* functions are implemented by the component where *use* functions are those called by the component. *Provide* functions are exposed to others components. Each *use* function, on the other hand, must

be implemented by some other component. This substitutes the API of the components. Many of these API can be factored into various interfaces as a collection of related functions. Individual components then can use or provide these interfaces. The main reason for such an arrangement is the event-driven nature of TinyOS. *Use* functions are events that signal task compilation. Consider the following scenario: component **A** uses the *Send* interface and component **B** provides this interface. A simplified version of the send interface is the following:

Block 1: Send Interface

```
interface Send {
        command error_t send(...);
        event void sendDone(...);
}
```

In fact, interfaces are contracts between components. The components that use the interface must implement the events defined by the interface and the components that provide the interface must implement the commands of the interface. Thus, **A** implements *sendDone* and **B** implements send function. Once **A** calls on **B**.*send*(), the control would be split and **A** can continue on another task until **B** signals task completion by calling on **A**.*sendDone*(). This scheme maximizes concurrency since function calls would be non-blocking.
There are two types pf components:

1. Module
2. Configuration

Simply put, modules are the atomic units of implementations and

configurations are the glue that put components together and create the system as a whole. Configurations connect the interfaces of various components. For example, consider the previous case of **A** and **B** component. **A** and **B** are in fact modules. Here is simplified implementation of **A** and **B** :

Block 2: Module A.nesc

```
module A{
        uses interface Send as sender;
}
implementation{
        event void sender.sendDone(...){...}

        task void start(){
                call sender.send();
        }
}
```

Block 3: Module B.nesc

```
module B{

        provides Send as sender;
}
implementation{
        command error_t sender.send(...){
                ...
                post sendComplete();
                ...
        }

        task void sendComplete(){
```

```
                signal sender.sendDone(...);
        }
}
```

How do we actually connect **A** and **B** together? We have to use a configuration component, let's say **abApp** , which first defines the required components and second mapps the used interface in one component to the corresponding provided interface in another component:

Block 4: Configuration asApp.nesc

```
configuration apApp{}
implementation{
        components A as a, B as b;

        a.sender -> b.sender;
}
```

Here, unlike C, '->' operator is used for interface connection rather than dereferencing. In, **A**'s module, *sender* is a placeholder for **A**'s used Send interface. Similarly, *sender* in **B** is also a placeholder for it's provided Send interface . '->' operator connects **A**'s used interface to **B**'s provided interface. In this section, we summarized the background knowledge we require to understand how TinyOS is used to build the simulation.

1.1.3 TOSSIM

Sensor networks are composed of a large number of motes. Arranging and managing these motes in the network is a crucial part of network simulation process. Luckily, TinyOS provides us with TOSSIM which is designed for this specific purpose.

TOSSIM is a TinyOS mote simulator which eases the development of WSN applications. It is a discrete event simulator for TinyOS sensor networks. Instead of compiling a TinyOS application of a mote and deploying it on a physical device, users can compile it into the TOSSIM framework which runs on a PC. TOSSIM scales to thousands of nodes and compiles directly from TinyOS code. Developers can test not only their algorithms, but also their implementations. TOSSIM simulates the TinyOS network stack at the bit level, allowing experimentation with low-level protocols in addition to top-level application systems. Users can connect to TOSSIM and interact with it using the same tools as one would for a real-world networking, making the transition between the two easy[4]. TOSSIM allows us to define motes in the network, connect motes to one another, add noise to the network and inject packets into the network.

1.2 Purpose

The final goal of our project is to find the performance boundaries of the security measures introduced in the paper *Query Privacy in Wireless Sensor Networks* by Carbunar et. al. The only reason that we are interested in creating a network simulation tool is that we want to run certain scenarios on the simulated network, collect some data and analyze them to support our theoretical claims. However, this paper barely touches the theoretical aspect of the analysis. This paper focuses only on the very first phase of the project which is the development of the simulation. On the very basic level, we want to create a simple network which is capable of source routing messages from an external source to a specific mote within the network. Once this simulation is functional, we can apply various

scenarios to it and collect the necessary data. Data collection and analysis would be the second phase of the project and will not be covered by this paper.

2 Network Simulation

The main purpose of this simulation is to run certain scenarios on the simulated network. These intended scenarios impose certain requirements on the network . The network is required to provide the following main functionalities:

1. Motes are capable of receiving messages.
2. Motes are capable of broadcasting messages.
3. Packets can be routed throughout the network.
4. Motes can be arranged in an arbitrary manner to create various network topologies.

In the following sections we will explain how these requirement are fulfilled by the simulation.

2.1 Architecture

2.1.1 Components

The first step to creating a WSN is creating its motes. The main component of each node is the *SourceRouter* module. In order to provide receiving functionality for this module, *Receive* interface is used by the module. This interface provides basic message reception. This interface

signals a *receive* event and requires the module to implement the corresponding handler. This will be the function that will be called on message arrival.
Next, in order to provide sending functionality for *SourceRouter* module, *AMSend* interface is used. Although TinyOS already has a simple *Send* interface, we decided to use *AMSend* since the latter suits the concurent nature of network communications better. It is very common to have multiple services using the same radio to communicate. Thus, TinyOS provides the Active Message (AM) layer to multiplex access to the radio. This interface provides the *send* command send that can be used by the module for message sending. This interface also signals a *sendDone* event to indicate whether a message was sent successfully or not and requires the module to implement the corresponding handler.
These two interfaces are the most important interfaces that the application uses. To provide full functionality of the mote, other interface are incorporated into the design. However, exploring them in detail would not contribute to the purpose of this literature. As such, we will briefly go over them:

- **Packet** : Provides the basic accessors for the message abstract data type.
- **Boot** : Notifies the component when TinyOS has booted
- **SplitControl** : Provides non-blocking behaviour for radio communications. In other words, it splits the radio's control from the rest of the hardware.

2.1.2 Compositions

On the application layer of the mote, *SourceRouterApp* is the configuration component that binds all the components of the system together.

2.2 Concurrency model

Communicative activities in WSNs are concurrent by nature. For instance, message arrival is an external event that can interrupt module's execution at any time. Message arrival will cause the receive function to be called preemptively. Message transmission on the other hand, is an internal activity that must be initiated by the module itself. Module does so by calling the *send* function. As it will be explained in section 2.3, we want to be able to send a new message right after we receive a message in order to route the message through the network. However, radio is a singleton resource and at any given time only one execution thread can access it. As such, we cannot call *send* function directly from the *receive* function since it would lead to a deadlock and the system crashes. Thus, we want to postpone the sending until we are done receiving. To do that, we define a helper task function which can be posted for later execution. This task will consequently call the *send* function whenever it is activated. Tasks of the same type are serialized by default and TinyOS guarantees the order of the posted tasks. Although the order is guaranteed, TinyOS does not make any assumption about the time that the tasks would run whether they are ordered or not. This can lead to certain problem due to the nature of the radio module and the way send function works. Once the *send* function is called, the message to be transmitted is handed to the radio and the control is split so that the radio can proceed to transmit the message while

the module can resume other tasks. Once the transmission is complete, the radio signals a *sendDon* event to indicate that the message has been sent. All period from when *send* is called until *sendDone* is signaled, the radio is busy and if another task tries to access it, it might cause a deadlock. In order to prevent this situation, each *send* task can use a global spin-lock to prevent other send tasks from accessing the radio. This lock is acquired by the task and is released by the *sendDone* event. Lock acquisition occurs in the body of the *send* task, however, it does not need to be in a critical section since (as mentioned earlier) *send* tasks are serialized and we would not have any race condition for accessing the lock. Figure 1 can help the reader better understand the sequence of actions.

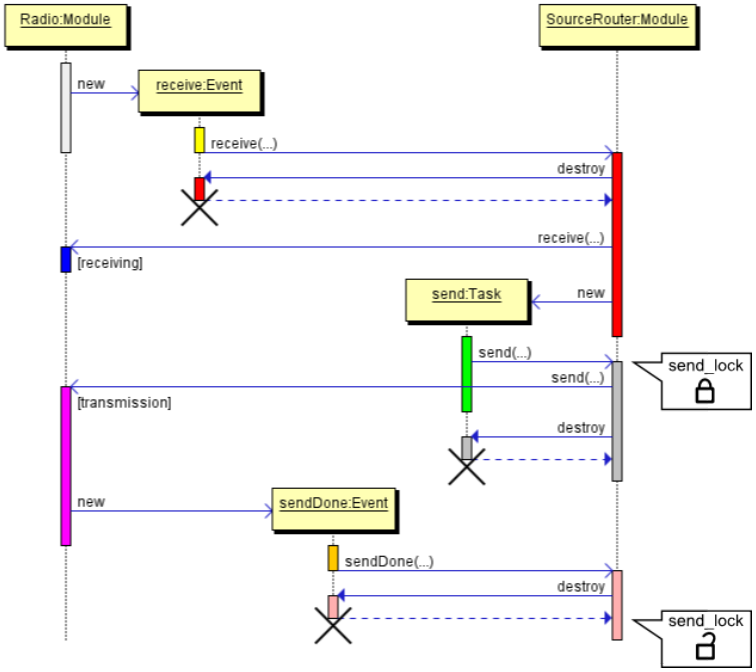

Figure 1: Sequence Diagram of the Mote

2.3 Network Protocol

At its minimum, the simulation must provide source routing mechanism. This means being able to determine the path that the message travels from the source to the destination before issuing the message. The simplest scenario includes one unique omniscient source external to the network which issues message in order to query information from specific mote within the network . Here omniscient refers to the fact that the issuer knows the entire topology of the network so it can encrypt the path in the message prior to its departure. The issuer can also choose different paths to query from the same destination mote.

In this scheme, at each step, an intermediate mote on the path inspects the message to determine the next mote that the message must be sent to. However, since we are working with wireless, devices there is no direct, one-to-one communication between the motes. The packet must be broadcasted at each step and will possibly arrive at multiple surrounding motes. Therefore, motes must have a way of determining whether they are the intended (intermediate or terminal) destination of the packet or not. This is made possible through the suggested message format and communication scheme.

2.3.1 Message Format

The message in this design has the following construct:

Block 5: Message Construct

```
typedef nx_struct source_route_msg{
        nx_uint16_t destid;
        nx_uint8_t data[256]
```

```
} source_route_msg_t;
```

destid indicates the intended destination of the packet, in other words "next hob". After each broadcast, receiving motes check this field to determine whether they are the intended destination or not. *Data* is a char array of the size 256 which contains the body of the message. However, the first portion of this array contains the remaining path that the message must travel to arrive at its final destination. Assume that the message arrives at a mote with id m_i along the path and the final mote's id is m_f. This would be the entire content of the maessage:

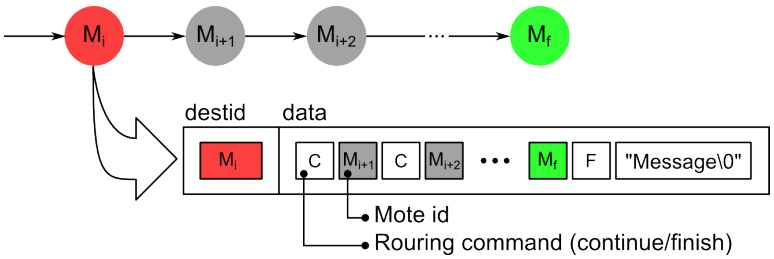

Figure 2: Message Format

As it is illustrated in Figure 2, the first portion of the data array contains a list of command/destination pairs. Next we will explain how the network motes use this message format to route the message through the network.

2.3.2 Communication Scheme

After m_i in Figure 2 receives the message and confirms that it is in fact the intended destination, it consults the data section of the message to figure out whether the message should be routed further or not. If the first character in the data array is 'F', mote realizes that it is in fact the final destination and the routing process terminates. Alternatively, the

character 'C' at the beginning of the array instructs the mote to continue the routing process. In this case, the mote proceeds by converting the second character in the array into a short integer and copying it into a new message's *destid* field. This second character in the array represents the next mote that the message must be sent to. Then, the mote copies the rest of the array (everything after the second index) into the new message's *data* field. Finally, the mote braodcasts this new message.
Once the message is broadcasted, it might arrive at several surrounding motes. However, each of these motes haveits own unique id. After each of these motes inspect the *destid* field of the message, only one of them will accept the message and the rest will drop the message.
In this design, there is only one active broadcaster at any given time. This active broadcaster is one of the intermediate motes along the route of the message. This communication scheme prevents message duplication. To elaborate on this point further consider the following scenario. Assume m_{i-1}, m_i and m_{i+1} are three consecutive motes on a route that a message takes from its source to its destination. m_{i-1}, m_i and m_{i+1} are initially listening for message arrival. Once the message arrives at m_{i-1}, it inspects the content of the packet to find out whether it is the intended destination of the message at this point or not. After realizing that is it in fact the intented receiver, it broadcasts the new message intended for m_i and starts listening once done broadcasting. At this point, all surrounding motes will receive the message but only m_i accepts it since it is the intended destination of the message. Similarly, as the active broadcaster, m_i broadcasts the message targeted for m_{i+1} while all other motes are listening. The message arrives at m_{i+1} and consequently m_{i+1} becomes the active broadcaster while all other motes are listening so on and so forth.

Figure 3 can help the reader better understand the concept.

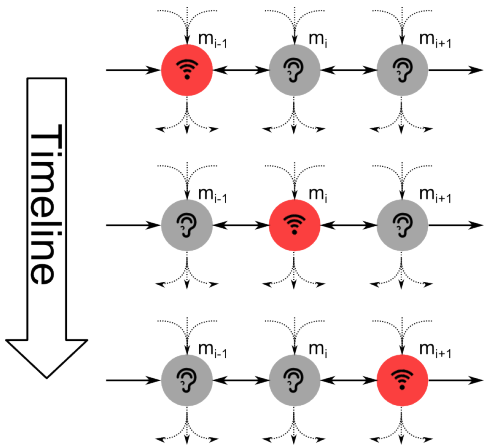

Figure 3: Singular Broadcaster

This property holds as long the server (original source) issues one message at a time which is true in our scenarios.

2.4 Simulation driver

The next step after completing mote's implementation is populating a large number of motes and assemble them together to create the network. To do this, TOSSIM is used. TOSSIM is capable of defining motes, connecting motes to one another, adding noise to the network and injecting messages into the network. However, a driver is needed to instruct TOSSIM to create the specific networks that we desire to use. This driver is written in python. It simply initializes the TOSSIM instance and instruct it to configure the topology of the network based on a provided `*.tpl` file. `*.tpl` files are topology descriptions. They can specify the motes and the connections between them. here is simple `*.tpl` file:

Block 6: Sample .tpl File

```
mote 1
mote 2
mote 3
gain 1 2 -30
gain 2 1 -30
gain 3 2 -60
```

This topology description maps to the network illustrated in the Figure 4.

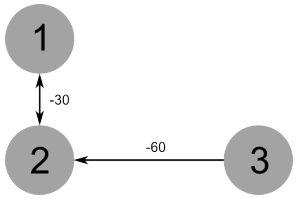

Figure 4: Simple Topology

The first portion of the `*.tpl` file describes the motes in the network and their ids. '`mote 1`' means define a new note with id 1. The second portion describes the connections between the motes . '`gain 1 2 -30`' means that when mote 1 transmits data, mote 2 hears it at -30 dBm. Once the topo file is parsed, the network is ready for message injection. This driver can act as the omniscient external source and issue queries to the network.

2.5 Topologies

As mentioned in section 2.4, in order to run the simulation, we must feed it with a `*.tpl` file. The format required by `*.tpl` files is very scriptive and is very prone to make mistakes. More so, some scenarios might require hundreds of motes connected in an arbitrary fashion. This makes it extremely difficult to edit the `*.tpl` file manually. The simulation driver is

agnostic of the actual physical structure of the network. It would parse any `*.tpl` file regardless of its semantics as long as it is syntactically correct. This can lead to many network anomalies (ie, topologies that cannot exist in reallity) which further separates the physical and the virtual domains. In order to avoid such situations, a tool is needed to enable the use to model topologies in a realistic manner. This tool must be intuitive and easy to use.

2.5.1 Requirements

The tool must have the following requirement:

- The user must be able to create topologies of arbitrary size
- The user must be able to determin the location of the motes with respect to one another
- The user must be able to determin the range of each motes signals
- The user must be able to connect or disconnect the motes

The requirements are very visual concepts. Thus a GUI tool is the most intuitive tool and would best fulfill our requirements. We created a Topology Generator tool based on these requirements.

2.5.2 UI Specification

Figure 5 illustrates the Topology Generator's GUI.

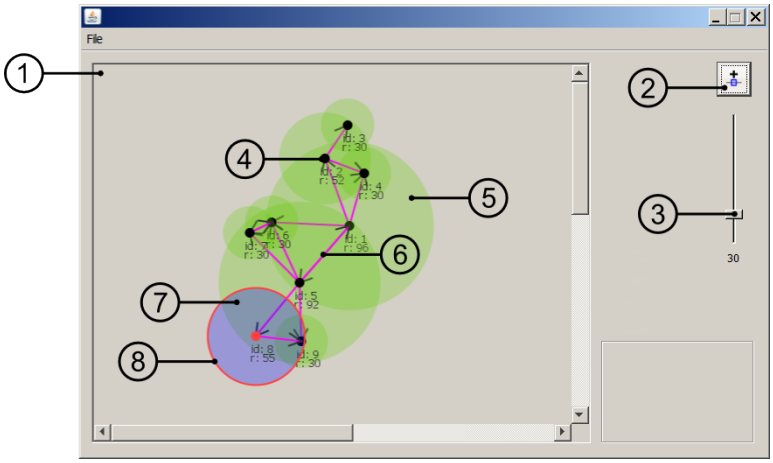

Figure 5: Topology Generator Tool

Here is a brief description of the UI:

1. **Map Panel**:

   This panel is the map of the topology. It represent a physical region in which motes can be positioned. It is an interactive panel which allows the user to manipulate the motes inside it.

2. **Add mote button**:

   This button allows the user to add new nodes to the map. Once the button is pressed, the user can add a new mote to the map by clicking on the Map panel. The new mote will appear where the user clicked on the map .

3. **Default range slider**:

   This slider determines the default initial range of the the motes when they are being added to the map.

4. **Mote center**:

This black dot indicates the location of the mote on the map. More so, the user can select the mote by clicking on this dot.

5. **Signal range**:
   This circle indicates the signal range of the mote.

6. **Connection edge**:
   This edge indicates a connection between two motes. Note that this edge is directed. If mote **A** lies within motes **B**'s range, this edge would be an arrow from **B** to **A**. If they both lie within each other's range this edge would be bi-directional.

7. **Selected mote**:
   The selected mote appears red with a blue range circle. There is only one selected mote at any given time. User can move the selected mote by dragging its center around the map

8. **Range circumference**:
   Circumference of the selected mote appears red whenever the pointer gets close to it. The user can drag this circumference toward and away from the mote center to change the range of the selected mote.

Note that mote connections are automatic in order to maintain the realistic structure of the topology. There is a connection from mote **A** to mote **B** if and only if **B** lies within the range of **A**. The user can enforce a connection by moving one mote into another's range or by expanding the range of one of them so that it would capture the other. In both cases the tool automatically detects and creates the connection. Once the user is satisfied with topology, she can export it as a `*.tpl` file.

3 Conclusion

Here, we simply review what has been accomplished and discuss whether our implementation meets the requirements.

3.1 Requirement revisited

To refresh our memory, here are proposed requirements:

1. Motes are capable of receiving packets
2. Motes are capable of broadcasting packets
3. Packets can be routed throughout the network
4. Motes can be arranged in an arbitrary manner to create various network topologies

In section 2.1, we explained the architecture of individual motes and in section 2.2, we discussed the execution model that the mote adhers to. This mote architecture combined by our concurrent execution model provide the basic send and receive functionalities. Then, we moved on and explained how our proposed message format and communication scheme will facilitate source routing. finally in section 2.5, by introducing the Topology Generator tool, we enabled the user to create aribtrary topologies on which different scenarios can be run.

3.2 Characteristics and Limitations

Before we finish, it is noteworthy to mention certain characteristics and limitations of the simulated networks.

3.2.1 characteristics

- Reliability: the network is not reliable in the general sense since the receiving motes never acknowledge the successfull message receival. There are several situations where a message can be lost. For example, The intended receiver might still be busy processing a message broadcasted from two stages before on the path. Since the radio is busy, the correct message is dropped and will never be revived. Also the noise introduced by TOSSIM might also corrupt the message.

- Scalability: the network is easily scalable since the routing mechanism of the network is fairly simple. There is only one omniscient source that issues the initial messages. The rest of the motes are agnostic of the structure and the topology of the network. All the information required to route a message is encrypted into the message by the external source. The burden of "valid" routing is on the external source and not the internal motes. So new motes can be added to the network without impacting the logic of routing.

3.2.2 limitation

- Bounded path length: The *data* field of the message is an array of length 256. Therefore, since each mote to mote transmision requires 2 entries of this array, the maximum path length that a message can travel is 128.

- Bounded number of motes: Since each mote id must be representable by a char in the *data* field of the message, up tp 128 motes can be

addressed.

Both of these limitations can be improved by increasing the size of *data* field of the message.

References

[1] John A. Stankovic: Wireless Sensor Networks, *University of Virginia*, June 2006

[2] B. Carbunar, Y. Yu, L. Shi, M. Pearce, V Vasudevan: Query privacy in wireless sensor networks, *Applications Research Center, Motorola Labs*

[3] P. Levis: TinyOS Programming , October 2006

[4] Levis, Phil. "Simulating TinyOS Networks." *Computer Science Division*. Web. 26 Apr. 2012. [http://www.cs.berkeley.edu/ pal/research/tossim.html]