

# Scalable Emulation of TinyOS Applications in Heterogeneous Network Scenarios

Yi-Tao Wang

*Department of Computer Science  
University of California, Los Angeles  
yitao@cs.ucla.edu*

Rajive Bagrodia

*Department of Computer Science  
University of California, Los Angeles  
rajive@cs.ucla.edu*

## Abstract

*Simulating the behavior of sensor applications in a heterogeneous network or under diverse environmental conditions is particularly challenging. In this paper, we present the design and implementation of TiQ<sup>†</sup>, a scalable framework that allows unmodified TinyOS applications to be evaluated in a diverse set of operating conditions, including heterogeneous networks. We validate TiQ against MoteLab, a physical sensor network testbed, and show that TiQ can predict the behavior of the real network with less than 4% error. Through several case studies, we demonstrate the key benefits of TiQ: (1) it supports a diverse set of scenarios involving heterogeneous networks, mobile data mules, and multiple operating systems, (2) it scales to over a thousand nodes and can simulate such large networks up to 6X faster than comparable simulators, (3) it provides a system to easily validate TinyOS applications, evaluate network designs, and optimize design parameters (e.g., beacon rate) based on individual criteria, and (4) it leverages existing physical layer models in network simulators to provide more accurate simulations.*

## 1. Introduction

Over the years, TinyOS has become a popular operating system for wireless sensor networks. Its compact size, numerous features, and modular framework offer portability, robustness, and scalability. Complex TinyOS applications can be created where motes work collaboratively to monitor the environment and communicate with base stations when interesting events occur. With large-scale deployments of motes running such applications, it is beneficial to first evaluate and validate their design and performance through simulation prior to deployment.

Furthermore, sensor network developers prefer to use emulations so that the software used for evaluation can be easily ported with little or no modification to the physical mote. This approach eliminates the possibility of introducing errors while porting the code from the testing and evaluation to the deployment phase.

<sup>†</sup>. This work was supported by the Army Research Office MURI grant W911NF-05-1-0246.

Although numerous tools have been developed for the study of sensor networks over the past decade, many of these are pure simulators, such as SensorSim [1], SWAN [2], and SENS [3], that use models to simulate the behavior of applications. While these simulations are scalable and sufficient to gauge performance metrics like network delays, packet collisions, and node localization errors, they lack the fidelity of software emulation. Among commonly used sensor simulators, TOSSIM [4], Avrora [5], and EmTOS [6] provide software emulation for the latest release of TinyOS and thus allow the actual application code to be used directly for testing & evaluation purposes. Although these tools provide high software emulation fidelity, they lack a diverse set of detailed simulation models, such as the sensing channel, battery, and clock skew, which restricts their utility in accurate performance prediction of sensor applications.

In contrast to simulators, physical testbeds provide high fidelity, but lack temporal and spatial scalability. Moreover, repeating experiments in physical testbeds is difficult because of the limited control they offer.

In this paper, we present TiQ, a simulator for sensor networks that enhances the current state of TinyOS emulation by meeting all of the following criteria:

- **Heterogeneity:** The simulator provides a diverse set of accurate models so that TinyOS applications can be evaluated in complex scenarios involving heterogeneous networks, operating systems, and conditions such as mobility. It offers an extensible framework that can enhance the simulation with new or emerging technologies and scenarios, such as different models of mobility or new MAC layer protocols.
- **Fidelity:** The simulator accurately captures the behavior of sensor motes. This requirement involves both software emulation and environment modeling. While using models for sensor motes offers better performance and allows the evaluation of high-level network interactions, only software emulation will allow us to evaluate the behavior of actual TinyOS code. Moreover, the simulator offers accurate models of sensing phenomenon that ensure the simulation results reflect those of real-

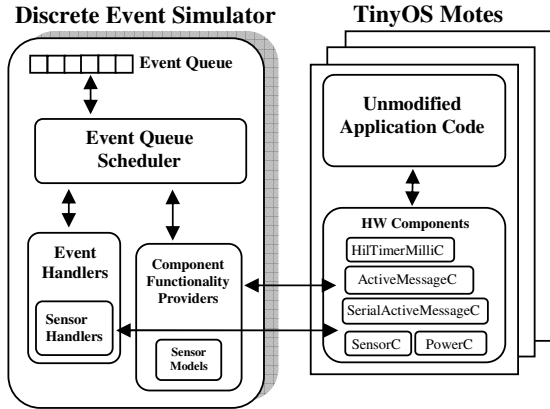


Figure 1. TiQ's architecture

world deployments.

- **Scalability:** As use of sensor motes continues to expand, the size of deployed sensor networks will increase. A framework that is able to efficiently simulate large networks for long simulation horizons is necessary to meet future simulation requirements.

Thus, our primary goal in designing and implementing the TiQ framework is to enhance the current state of simulators by providing accurate and scalable simulations of complex scenarios. TiQ leverages existing models in network simulators to provide accurate simulations without the need to re-implement physical layer models for stand-alone simulators. The TiQ framework and our implementation approach can be extended to any discrete event network simulator to provide emulation of TinyOS applications. As we will demonstrate later in our case studies, TiQ not only supports common sensor networks used for environment monitoring [7], [8], but also multi-tiered networks with mobile data mules [9] and multiple operating systems.

TiQ allows network designers to validate their TinyOS applications without modification before deployment. Our experiments show that TiQ can predict the behavior of MoteLab [10], a physical sensor network testbed, with less than 4% error. Thus, TiQ can provide evaluations of network designs comparable to a physical testbed, without the high costs and limited control of physical testbeds. TiQ can support networks of over a thousand nodes and, based on our comparison against other TinyOS emulators, perform simulations up to 6X faster for such large networks. TiQ provides a platform to easily evaluate and optimize a variety of parameters, such as packet size, based on individual criteria.

## 2. TiQ

TiQ emulates TinyOS applications in an accurate and scalable simulated environment. We consider TiQ to be complementary to other TinyOS emulators and physical testbeds that provide higher fidelity. Designers can verify

the correctness of their applications on a few motes and may then use TiQ for large scale evaluations and/or detailed performance prediction studies.

In the remainder of this section, we will discuss the details of TiQ. First provide an overview of TiQ's design. Then we will discuss our implementation of TiQ, its execution, and sensor models.

### 2.1. Design Overview

Our design of the TiQ framework takes advantage of TinyOS's structure. TinyOS is a programming framework that enables an application-specific OS to be built for each application through the use of components. Components are abstractions of objects (i.e., timer, radio, sensor, etc.), each providing functionality related to its abstraction, similar to classes in Object-Oriented Programming. All hardware resources are abstracted as components. Each component uses and provides the functionality of interfaces, abstractions of actions (i.e., send, set, print, etc). When a component uses an interface, it must define the component that provides that interface. The design and programming model of TinyOS is event-driven, which lends itself to emulation in a discrete event simulator.

Figure 1 provides an abstract overview of TiQ's design and our integration approach. TiQ is composed of a discrete event simulator (DES) and emulated TinyOS motes. The DES is responsible for simulating the environment (i.e., radio, mobility, etc) and other parts of the network that are not TinyOS motes. The emulated motes execute unmodified TinyOS application code. The DES initializes the motes and processes events from the emulated TinyOS motes.

In order to emulate TinyOS, we replace a selected set of hardware-abstracted components with custom components that provided the same interfaces but that functionally interact with the DES instead of physical hardware. The TinyOS application code used in TiQ is identical to the code deployed on physical motes. From TinyOS's perspective, the discrete event simulator is just another hardware platform because TinyOS's component-based framework hides the implementation of the components from each other. When the application uses the interface of a replaced component, the interface calls a Component Functionality Provider (CFP) (Figure 1) in the simulator. The CFPs use underlying models to simulate the desired behavior of the hardware in the DES's simulated environment, queueing events if necessary.

In order to provide more accurate simulation of sensor networks, we replace several components of interest to sensor motes: the battery, sensor, radio, clock, and UART serial port. Each of these components uses a model (e.g., sensing channel) in the DES to simulate the behavior of the corresponding hardware. The clock is

also replaced to offer the DES control of the simulation time, to model clock skew and drift, and to synchronize the executions of the simulated motes. The specific TinyOS components for the clock, radio, UART, sensor, and battery are shown in Figure 1. It should be noted that there are several TinyOS components for the sensor and battery, so we use generic TinyOS components (SensorC and PowerC) that can perform the functionality of all the components. Section 2.5 provides details on the specific models that correspond to the replaced components.

For portability to various discrete event simulators, our design does not specify the exact implementation of the DES. However, our implementation, discussed in the next section, can be extended to other DESs.

## 2.2. Implementation

Rather than build a simulator from scratch, an early architectural decision was to leverage the capabilities of existing network simulators. Although we chose to use QualNet [11] due to our familiarity with it from our previous work, SenQ [12], the TiQ framework and our implementation approach can be similarly used with another DES. Using QualNet in TiQ allows us to leverage its modular framework, which simplifies the implementation of the TiQ framework and several sensor-specific models (i.e., clock skew, clock drift, sensing channel, and power consumption) for TinyOS applications.

In order to provide a high fidelity for their behavior, each TinyOS mote is executed in its own thread without preemption. Since TinyOS is known for its small memory footprint (usually less than a few KBs), each thread requires trivial memory overhead. As our later tests show (section 4.2), TiQ can scale up to a thousand TinyOS motes. Moreover, multi-threading TiQ allows it be easily extended in future releases for improved performance through parallel execution on multi-core/multi-processor systems. The specifics of the execution process is covered in Section 2.3.

In the DES, the TinyOS motes are abstracted as simulation objects and communicate through interfaces with the rest of the simulator, hiding the implementation details of each node. A mote is represented as a simulation object with custom event handlers and CFPs. From the DES's perspective, the mote is just an abstract node that generates events. This implementation allows the emulation of the TinyOS application to be independent of the models used by the simulator.

## 2.3. Execution

Figure 2 shows the control flow diagram of a TinyOS thread. As a part of the DES's initialization, the TinyOS mote is booted up in a thread. At this point, the DES blocks itself and transfers control to the thread. The TinyOS thread boots, initializing all other hardware and

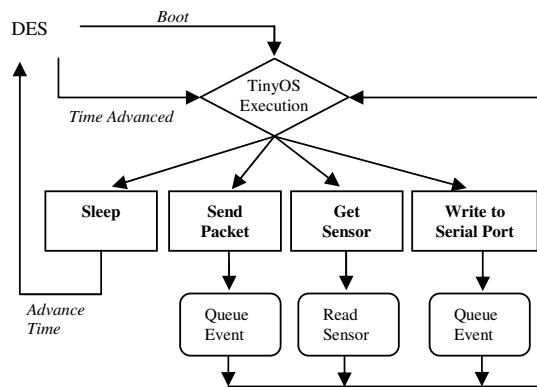


Figure 2. The control flow of a TinyOS thread: squares are interfaces of a replaced component, rounded squares are CFPs, and italics are interactions involving context switches

software components that it needs, and executes its tasks. What happens when the thread uses an interface of a replaced component is discussed in section 2.4 but it suffices to mention here that a context switch only occurs when the mote tries to sleep, either because the mote has run out of tasks to execute or because a component explicitly puts the mote to sleep. For all other functions, the TinyOS thread retains control and continues to execute until it reaches the condition above. When that happens, the thread will store its global data, unblock the DES, and block itself. Then, the DES will boot the next node and the process repeats until all motes are booted.

After all motes are booted, the DES advances the simulation time, loads the global data of the first available TinyOS thread that has not executed at this simulation time, blocks itself, and switches control to that thread. Once again, the mote will continue to execute until it tries to go to sleep, at which point, control is switched back to the DES. The process repeats until the end of the simulation.

The synchronization between TinyOS threads and the DES is equivalent to a global synchronization at every time step, preventing the simulation time from advancing until all motes have executed their tasks at that time step. This ensures that no mote can move so far ahead in simulation time that there can be some messages from other motes which it should have received at an earlier simulation time.

## 2.4. Interactions

In previous sections, we outlined how the TinyOS motes and the DES can communicate. Now, we discuss the specific interactions between them (Figure 2). Note that all interactions that start on the TinyOS side are triggered when the application uses an interface of a replaced component. There are three interactions that are required for execution management of the TinyOS

notes:

- *Boot*: During the simulator’s initialization, it will spawn a thread for each TinyOS mote. The DES will tell the mote its ID and wait for it to finish booting.
- *Advance Time*: When a mote tries to go to sleep, either because it has done all the processing that it can at the current time step or one of its tasks puts it to sleep, it will tell the simulator to proceed to the next time step by queuing an *advance time* event and blocking itself. The advance time event will be processed by the DES at the next time step. Each mote can adjust the size of the time step that it wants. For example, users can have temperature sensing motes sleep for 200 ms and light sensing motes sleep for 400 ms in the same simulation.
- *Time Advanced*: When the DES gets an *advance time* event, it will unblock the mote that queued the event and give it the current time. Since all processing happens in zero time, a behavior common to all discrete event simulators, the current time will not change and the mote can simply cache this value instead of asking the DES for it multiple times.

These three interactions are the only times that a context switch will occur. In particular, the *Advance Time* and *Time Advanced* interactions synchronize the TinyOS threads at each time step (section 2.3). The rest of the interactions are for hardware resource usage:

- *Send Packet*: Motes that want to transmit a packet will queue events in the DES with the packets that they want to send. The DES will then simulate the transmission of the packet to other motes. When the DES determines that the mote should receive a packet from the MAC layer, it will pass the packet to the corresponding TinyOS mote.
- *Get Sensor Value*: When a mote wants a sensor reading, the DES will produce one using its sensing channel model [12].
- *Send Message on Serial Port*: Communication over the serial port works similarly to sending packets if the other end is a simulated mote. However, users can also set the other end to standard output or files, in which case, the data message will be written there.

## 2.5. Sensor Models

There are three main physical phenomenon that are specific to sensor motes which are not modeled in our DES, QualNet: the clock, the sensing channel, and the battery. We added models for these based on our previous work on SenQ [12]. One of the key functions of sensor networks is to aggregate data (such as the network in Section 4.1). In order to accurately match up sensor readings for analysis, the various motes must

agree on a global time. Most simulators assume that nodes have perfect time. However, real motes are never perfectly synchronized with each other. Thus, we must model clock drift to accurately simulate the behavior of TinyOS motes.

For clock drift, we have used the model presented in [13]:

$$f = f_{nom} + \Delta f_0 + a \times (t - t_0) + \Delta f_n(t) + \Delta f_e(t)$$

where  $f$  is the frequency of an inaccurate clock,  $f_{nom}$  is the nominal frequency,  $t_0$  is the starting time,  $a$  is the aging rate,  $\Delta f_n$  is the noise effect, and  $\Delta f_e$  is the environmental effect. The DES keeps an accurate view of the global time, but each mote has a different clock skew and drift. The time is converted when it passes between the DES and the mote according to the clock model.

Despite the importance of the sensing channel to sensor networks, many simulators and emulators use a simplistic model that reports random or specific values to the sensors. For TiQ, we have provided two extra sensing channels: mobility and diffusive. The mobility sensing model uses trigonometric functions to determine when a mote is going to enter or leave the vicinity of a sensor. The diffusive sensing model allows sensor readings to be specified at any space-time coordinates. For the same location, the sensor readings between two different time coordinates will change linearly. At any time, bilinear spine interpolation of the four corners and specified points is used to calculate the sensor readings of unspecified points.

For the battery, we have used a model that we previously published in [14] but we give a short description here for completeness. The model takes into account both non-linear discharge and recovery effects, which are present in real life batteries:

$$C(T) = \int_0^T I(t) dt + 2 \sum_{m=0}^{\infty} \int_0^T e^{-\beta^2 m^2 (T-t)} dt$$

where  $C(t)$  is the capacity used after time  $T$  and  $I(t)$  is the current drawn from the battery at time  $t$ . For the special case of loads applied for milliseconds, which is typically the granularity for sensor motes, we found that the function can be approximated by a polynomial function. Thus, we can use the model at a low computational cost.

## 3. Accuracy of TiQ

Since TiQ is an evaluation tool for TinyOS applications, it is important to validate it against physical motes to ensure that the conclusions from simulations are accurate and useful for real deployments. To validate the accuracy of TiQ, we ran a sensor network using

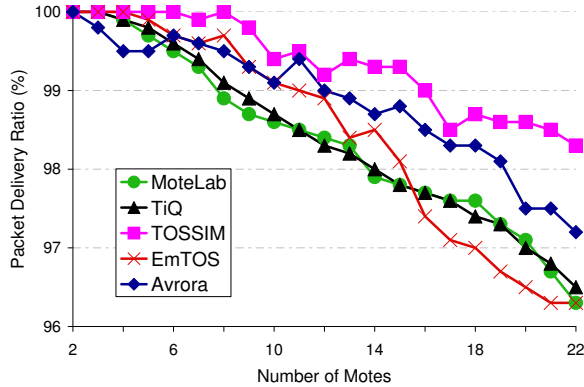


Figure 3. Reproducing the MoteLab scenario in TiQ and various TinyOS emulators

Harvard’s MoteLab [10] and TiQ. We selected 22 random TMote Sky motes (indoor range of approximately 100 meters) from the second floor of the facility such that all motes were able to communicate with at least one neighbor. We used a simple network design where one source would periodically send a packet to the sink. Flooding was used to route messages and CSMA was used as the MAC protocol. Each test was repeated 20 times with random source and sink selections. After running the tests using MoteLab, the experiment was replicated in TiQ. The motes locations were reproduced in TiQ. The same TinyOS applications code was used with CSMA for the MAC protocol. The radio channel was simulated using two-ray pathloss, radio interference, and Ricean fading.

Figure 3 shows the packet delivery ratio (PDR) of the experiment reported by MoteLab, TiQ, TOSSIM, Avrora, and EmTOS. We observe that, as more nodes are added to the network and they start to interfere with each other, the PDR decreases. Interestingly, the results predicted by TiQ are almost identical to those obtained via measurements from the physical testbed. The error between TiQ’s predictions and the real behavior from MoteLab is less than 4%. TOSSIM has an error that is 15X that of TiQ’s while both Avrora and EmTOS have errors that are 5X that of TiQ’s, which resulted from abstract models used for the physical layer.

We recognize that TOSSIM, Avrora, and EmTOS can be extended with more models to provide similar fidelity to that of TiQ. However, it’s more efficient to leverage network simulators to provide simulation fidelity without re-implementing models for stand-alone sensor simulators. For example, the more accurate models that were only in TiQ (e.g., two-ray pathloss and Ricean fading) requires a layered communications framework and support for non-uniform dynamic environments [15], [16]. These requirements are lacking from most stand-alone simulators but are present in most network simulators.

## 4. Case Studies

Simulation fidelity and flexibility are necessary to evaluate TinyOS applications. The following case studies highlight the diverse set of sensor networks that TiQ supports. The first case study uses TiQ to evaluate an environmental monitoring sensor network, in the presence of clock drift. The next case study uses TiQ to evaluate the effects of network parameters on power consumption in a multi-tiered network with mobile data mules. The final case study uses TiQ to model complex scenarios involving heterogeneous nodes, networks, and operating systems. Together, these case studies demonstrate the ease with which TiQ can validate TinyOS applications and evaluate a variety of networks.

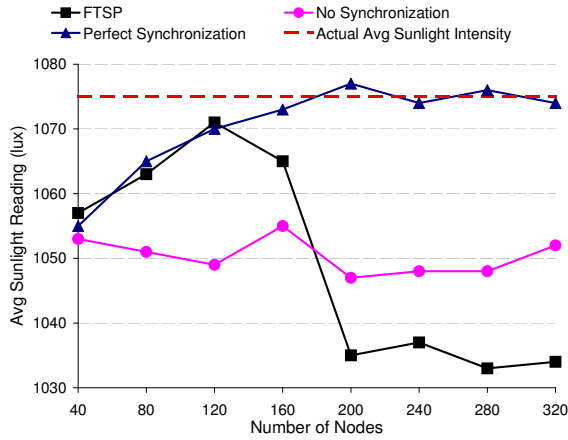
### 4.1. Evaluating an Environment Monitoring Network

Wireless sensor networks provide real-time data of the environment that enable scientists to measure and study properties that have previously been difficult to observe. Some research groups and companies have found interest in it and began to work on the deployment and application in environmental and habitat monitoring [7], [8]. For this case study, we consider a common environment monitoring scenario in which we want to deploy a network of Mica2 motes to determine the average sunlight a 1000m x 1000m area receives. Multiple photo sensors route their readings to a base station that aggregates the data. We want to determine the optimal network size to provide the most accurate data.

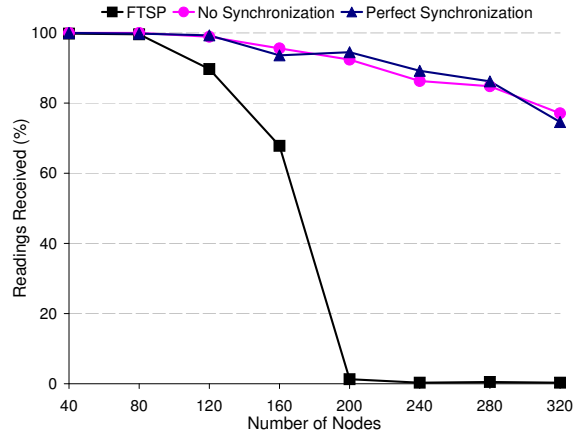
Clock synchronization is essential for sensor networks that aggregate sensor readings for analysis. Without a global agreement on time, the data from different sensors cannot be accurately matched up. TiQ provides a clock model, which allows us to evaluate our application with Flooding Time Synchronization Protocol (FTSP) [17], a popular synchronization protocol, in the same setup as that of a real deployment.

In FTSP, a root node is selected in the sensor network. The root’s clock becomes the global clock, which is periodically flooded throughout the network. Since a node can receive the same synchronization message multiple times from each of its one-hop neighbors, it can estimate its clock offset and rate difference from the root node. Linear regression over past messages can be used to determine a node’s current clock error even when it does not receive any synchronization messages.

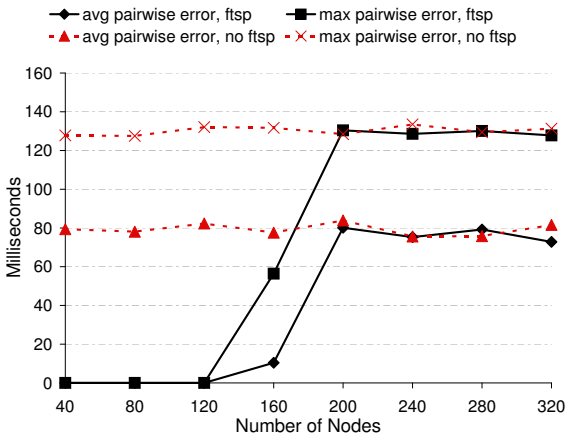
In our simulation, the motes start with no initial clock skew and a clock drift of 0 to 40  $\mu$ s, the reported drift range for Mica2 motes [17]. The radio range was about 100m and we configured a terrain of 1000m x 1000m, thus the network diameter is about ten hops, with nodes deployed in grids of varying density. A base station was placed at the center of the grid to receive and



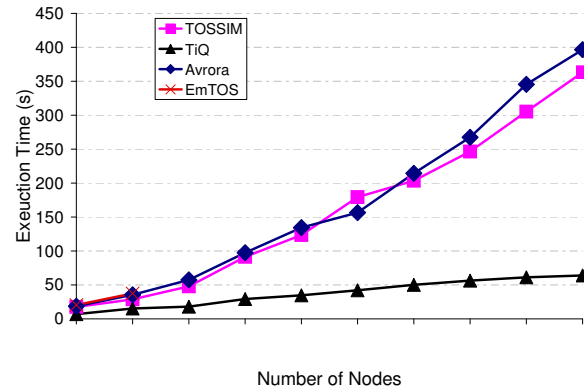
(a) The average sensor readings received by the base station at the end of the experiment



(b) Percentage of sensor readings successfully received by the base station



(c) The average and maximum pairwise clock errors as the number of nodes increases



(d) Time required to execute 10 simulated seconds of the scenario in various TinyOS emulators

Figure 4. Results of the environment monitoring network simulation

aggregate sensor readings. We used the FTSP and *MultihopOscilloscope* application code directly supported by the TinyOS 2.x distribution. CSMA was used as the MAC protocol. FTSP was configured to use a resynchronization interval of 30s and sensor readings are taken every 120s. The simulation ran for 1 simulated hour. A non-uniform sunlight intensity (average of 1075 lux) was applied to the field throughout the experiment. We recorded the sensor readings at the base station under three conditions: (1) no synchronization protocol is used by the network, (2) FTSP is used by the network to synchronize the clocks, and (3) the network has perfect synchronization, which is simulated by turning off the clock drift.

Figure 4 shows the results of our experiment. Figure 4(a) shows the average photo sensor readings when the network has perfect synchronization (no clock drift) and when it uses FTSP to synchronize in the presence of clock drift. For brevity, we only show the average of all of the photo sensor readings at the end of the simulation,

when clock errors have had the largest impact on sensor reading fusion. The dark dotted line is the average sunlight intensity (1075 lux) that we applied to the terrain. From the results, we can conclude that 120 nodes is the optimal number to deploy. However, the results also provide unexpected insight into FTSP. Specifically, there are three ranges of interest:

- **40-120 nodes:** We observe that the average readings from FTSP are almost identical to those of the model that assumes perfect clock synchronization, so the difference from the true intensity reading is not due to any limitation of FTSP. Upon closer inspection, we determined that the inaccuracy in the readings and the increase in reading accuracy over this range is due to the size of the network. That is, as the sunlight distribution over the terrain is non-uniform, more nodes cover more space and at a finer granularity, which, in turn, allow us to gather a more accurate reading of the average sunlight over the terrain.
- **120-200 nodes:** In this range, the accuracy of the

readings as predicted by the model with FTSP drops dramatically. We observe that the accuracy of the readings from the model with perfect clock synchronization continues to increase. As the only difference between the two network models is the presence of FTSP, we expect that the operation of FTSP in this range of nodes is responsible for the increased inaccuracy. Furthermore, the accuracy of the readings from the model with FTSP drops below that of the model with no clock synchronization protocol, so the root cause is unlikely to be simply the failure of FTSP to synchronize the clocks.

A closer inspection of the percentage of sensor readings successfully received by the base station (Figure 4(b)) reveals that as the network size grows beyond 120 nodes, FTSP floods so many synchronization packets in the network that they interfere with the transmission of the sensor reading data packets. As a result, the base station is unable to receive as many of the sensor reading data packets.

- *200+ nodes:* Interestingly, the accuracy of the readings from the model with FTSP level out in this range. From Figure 4(b), we observe that the results of the readings from this model correspond to when the base station is almost completely unable to receive any sensor readings for the reason previously discussed. Closer inspection reveals that, in this range, the base station only managed to receive packets from its immediate neighbors and we verified that 1035 lux is the sunlight intensity of the area around the base station.

We also plot the average and maximum pairwise clock errors (Figure 4(c)) in the network as a function of increasing network size (and hence density since the terrain area is kept constant). As expected, the pairwise clock errors remains relatively constant around 17  $\mu$ s up to 120 nodes, where the flooded synchronization packets do not have a large impact on network traffic. However, the protocol starts failing at 160+ nodes. Although FTSP is flooding the network with synchronization messages, the network is clearly unable to handle the larger traffic volume. As the network density increases, nodes are not synchronized for longer periods of time and the clock errors approach those of the model where no clock synchronization protocol is used.

From this case study, we determine that 120 nodes is the optimal number to deploy over the terrain using FTSP. However, TiQ also revealed that FTSP may not be the optimal synchronization protocol for our environment monitoring network due to its scalability problems as previously discussed. A protocol without scalability problems, closer to the perfect synchronization case, could provide more accurate readings for the network. In that case, the optimal number of nodes to deploy

becomes 160.

The purpose of this case study is to highlight the benefits of TiQ's evaluation framework. Although, in retrospect, it may not be surprising that FTSP would suffer from scalability problems because of its flooding, TiQ allows a quantitative evaluation of its limitations, as well as any other protocol implementations. It allows us to not only determine the optimal number of nodes to deploy for sensor networks, but also quantitatively evaluate clock errors, packet reception, and other metrics for network designs.

## 4.2. Scalability

Figure 4(d) compares the performance of TiQ with TOSSIM, Avrora, and EmTOS as measured by their execution time as a function of the number of sensor nodes for 10 simulated seconds. We used the scenario from section 4.1 and the same TinyOS application simulated under perfect synchronization conditions. From the results, we observe performance improvements of almost 6X for 1000 nodes and 10 simulated seconds. It should be noted that EmTOS is designed for small scale networks and could not scale beyond 200 nodes [6].

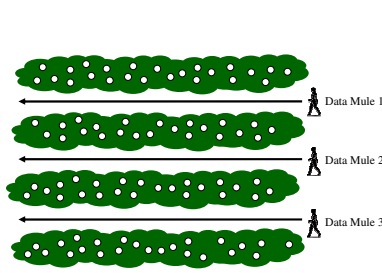
These results demonstrate that TiQ is able to provide high fidelity emulation of realistic TinyOS applications with low overhead for each mote. TiQ's accuracy and scalability makes it a beneficial framework for evaluating large sensor networks.

## 4.3. Evaluating Multi-Tiered Networks

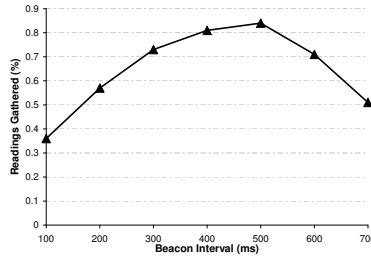
Power is a limitation of most wireless sensor networks and replacing the batteries for all the nodes in a large sensor network is costly. A number of researchers have proposed mobility as a method of extending the lifetime of sensor networks. Mobile elements, data mules, traverse the network and collect data from the sensor nodes and forwards it to an AP. The use of data mules avoids the communication cost of a multi-hop network. Furthermore, the sensor nodes no longer need to form a completely connected network and it can be deployed with focus on the sensing aspects.

To illustrate TiQ's capability to evaluate heterogeneous networks, we use the multi-tiered scenario from [9] shown in Figure 5(a). In this scenario, we need to gather temperature readings from sensor nodes that are deployed on each row of grapes. As the workers, carrying specialized devices, slowly work their way through the vineyard, they act as data mules to gather readings from the nodes.

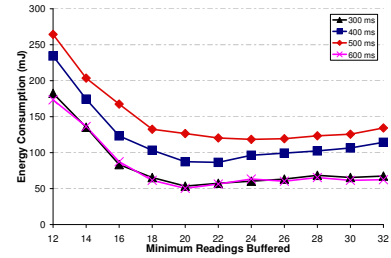
200 nodes are deployed in four rows over a 700m x 700m terrain and gather readings every second. Three data mules go down the rows at 1 km/h (working speed). Each reading is only 1 byte. The simulation ends when all the data mules walk to the other end of the row and back.



(a) The multi-tiered network: sensor nodes are distributed on four rows of grapes and workers act as data mules



(b) The percentage of the overall sensor readings delivered to the data mule



(c) The average energy consumption of a mote at various beacon intervals

Figure 5. Evaluation of the multi-tiered network

In this example, the structure of the network is set, but we can use TiQ to find the desirable values for network parameters, which are, in this case: (1) the beacon rate for the data mules and (2) the buffer size: the minimum number of readings for a mote to buffer before downloading to a data mule when one is in range. The first parameter is how often the data mule beacons to let the nodes know that it is in range. A small beacon interval reduces the time nodes have to transmit to the data mule. A large interval increases the probability that nodes won't detect the mule as soon as it comes into range. The second parameter, buffer size, affects the energy consumption and lifetime of the node. Sending a reading to the mule immediately would result in a large number of transmissions and high energy consumption. In other words, this parameter denotes the number of readings per packet assuming no data padding.

Figure 5(b) shows the percentage of sensor readings gathered by all the nodes during the simulation that were transmitted to a data mule as a function of the data mule's beacon rate. As previously discussed, larger intervals provide more time for the nodes to transmit their readings to the data mule, which explains the increase from 100ms to 500ms. After 500ms, the nodes won't detect the mule as soon as it is in range and transmission time is wasted.

Figure 5(c) shows the average energy consumption of a node as a function of the minimum number of readings the node buffers before sending to the data mule at various beacon intervals. We observe that the minimum energy consumption for all of the beacon intervals is around 20-22 readings. Energy consumption decreases up to this point because of energy savings from less transmissions. However, after this point, energy consumption increases because of the higher cost for retransmission and the higher occurrence of data mules moving out of range before a transmission is finished.

These two graphs show that there is a tradeoff between the energy consumption and the readings gathered by the data mules. More readings transmitted to the data

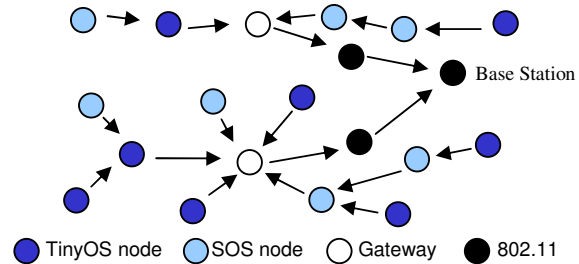


Figure 6. An 802.11 network and sensor network with multiple operating systems (SOS and TinyOS)

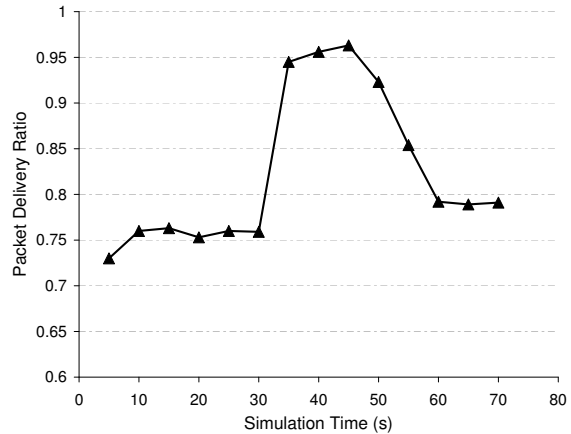


Figure 7. Percentage of packets successfully delivered for the heterogeneous network

mule requires more energy consumption and vice versa. Using these results from TiQ, it is up to the network designers to decide the most desirable parameter values based on their criteria for optimal energy consumption and readings gathered.

The purpose of this case study is to bring out a few key points of TiQ. First, TiQ can simulate multi-tiered networks involving mobile components. Second, TiQ can be used to easily evaluate the effects of parameters on the network so that designers can pick their optimal values based on individual criteria.

#### 4.4. Evaluating Heterogeneous Operating Systems

TiQ’s modular design also allows the underlying network simulator to be extended to support emulation of other operating systems in conjunction with TinyOS emulation. To demonstrate the benefits of multiple operating system support, we extended our implementation of TiQ with SOS [18] emulation from our previous work on SenQ [12].

With emulation of both SOS and TinyOS, TiQ can be used to evaluate scenarios similar to the one shown in Figure 6. In the scenario, we want to deploy a network of 500 TinyOS sensor motes to replace a deployed network of 500 aging SOS motes that are part of a heterogeneous network with 50 IP nodes. The SOS motes route sensor readings to gateway nodes which then processes them and routes the results to a central station using 802.11 radios, the IP protocol, AODV for routing, and UDP for end-to-end communication. In this scenario, we need to ensure that the TinyOS motes can co-exist with SOS motes and take over their functionality as the SOS motes gradually die off.

In this hybrid network, the sensor network is emulated and the IP network is simulated. The gateways are modeled as nodes with two network interfaces (TiQ can also model a gateway as a high end node with an 802.11 interface connected with a sensor mote over a serial port connection). In our experiments, 500 SOS motes, 500 TinyOS motes, and 50 IP nodes are randomly distributed over a 1000m x 1000m terrain. The SOS motes run the *Surge* application and the TinyOS motes run the equivalent *MultihopOscilloscope* application. The routing in both applications was modified to use tree routing with multiple trees rooted at 20 gateways. The TinyOS motes all boot up at 30 simulation seconds and the SOS motes die off at random times between 40 and 60 simulation seconds.

Figure 7 shows the results of this experiment. We observed that the percentage of successful packet deliveries increased after the TinyOS motes are booted. Closer inspection revealed that using 500 motes fails to cover such a large area. After the TinyOS motes were booted, many isolated SOS motes were able to route sensor readings to a gateway. As the SOS motes died off, many TinyOS motes were isolated and no longer able to route packets to a gateway. These results demonstrate two points: (1) the TinyOS application is coded correctly in that they cooperate with deployed SOS motes to route sensor readings and (2) about 25% of the motes are isolated and cannot route their readings to a gateway.

This case study demonstrates how TiQ can be used to evaluate designs in complex scenarios involving heterogeneous operating systems, nodes, and networks. In principle, is scenario could have been evaluated using

a physical testbed, but that would have been costly on such a large scale. Thus, TiQ provides the flexibility of a physical testbed, by allowing users to simulate complex scenarios, without the scalability and cost limitations.

#### 5. Related Work

Existing work in the simulation and emulation of sensor networks can be categorized into four classes: (1) pure network simulators modified to support sensor networks, by adding sensor-specific models, emulation of sensor applications, etc., (2) simulators built from scratch, (3) emulators for sensor applications enhanced with network models, and (4) emulators at the instruction cycle granularity.

Most earlier sensor simulators belonged to the first category, in which sensor models were added to popular network simulators. The popular network simulators used are ns-2 [16], GloMoSim [15], and QualNet [11]. SensorSim [1] extends ns-2 with models for the sensing channel and power consumption. TOSSF [19] uses scripts and source-to-source compilers to translate the TinyOS code from nesC into C++. TOSSF’s approach modifies the original application code to a larger extent than that of TiQ and offers more possibilities for the introduction of bugs not present in the original application. TiQ replaces a few hardware-abstracted components in TinyOS to integrate the application with the DES. The remainder of the code that executes in TiQ is identical to that which runs on sensor motes. SenQ [12] extends QualNet to emulate the SOS [18] operating system.

The second class, simulators that are built from scratch, focus on scalability rather than offering detailed models. SENS [3] is a component-based architecture for modeling sensor applications and the network environment. SWAN [2] also focuses on providing large scale simulations. Both of these simulators lack the accurate models offered by TiQ and uses models to simulate applications, which have the same drawbacks as the first category above.

The most prevalent evaluation tools for sensor applications belong to the third class, in which sensor emulators are enhanced with network models. TOSSIM [4] comes with the TinyOS distribution and provides a rudimentary simulation of the physical environment. EmStar [20] takes a similar approach to TOSSIM but targets higher end sensor motes. EmTOS [6] extends EmStar with TinyOS emulation. All these emulators use highly abstract models, especially for the wireless channel, can lead to inaccuracies in the predicted performance and behavior of simulated sensor networks [12]. Given the significant amount of effort that has already been invested in network simulators, it appears to be more beneficial to integrate this capability directly within sensor emulators. The TiQ framework allows the integration of emerging network simulators like ns-3 into

existing sensor emulators, and we hope that this work suggests an approach in that direction.

The final class of evaluation frameworks for sensor networks is instruction cycle level emulators. Atemu [21] and Avrora [5] are two examples of such emulators. They use an emulated processor to execute applications compiled for sensor motes. Since the code is entirely without modification, this approach offers the highest possible software emulation fidelity at the cost of intensive computation to emulate the processor. These emulators focus on evaluating the correctness of implementation and we consider them to be complementary to TiQ. Designers can use these tools to verify the instruction-level correctness of their applications before they use TiQ to perform large scale evaluations under deployment conditions.

## 6. Conclusion

In this paper, we have presented the design and implementation of the TiQ framework, which provides high fidelity and scalable emulation of networked TinyOS applications. Through analysis and case studies, we have demonstrated the key benefits of TiQ over the existing suite of emulators and simulators: (1) it allows TinyOS applications to be emulated without modification in a controlled and repeatable manner, (2) it can accurately predict the behavior of physical sensor networks with errors of less than 4%, (3) it provides an evaluation platform for diverse scenarios that can involve heterogeneous networks, mobile data mules, and multiple operating systems, (4) it supports networks of over a thousand nodes, (5) it allows various network parameters, such as beacon rate and buffer size, to be easily evaluated and optimized prior to real deployment, and (6) it leverages existing models in network simulators to provide accurate simulations without the need to re-implement physical layer models for a stand-alone simulator.

The TiQ design and our implementation approach, in particular the models for the sensing channel, clock, and battery, can be used to provide TinyOS emulation to any DES. We believe TiQ is complementary to other TinyOS emulators (e.g., Avrora) and physical testbeds that are designed to provide higher hardware-related fidelity. Designers can use these other tools to verify the correctness of their applications on a small scale before using TiQ to evaluate specific optimizations and modifications to their TinyOS applications and network designs before deployment.

## References

- [1] S.Park *et al.*, "Sensorsim: a simulation framework for sensor networks," in *MSWIM '00*. New York, NY, USA:

- ACM, 2000, pp. 104–111.
- [2] F.Perrone *et al.*, "Simulation modeling of large-scale ad-hoc sensor networks," in *Simulation Interoperability Workshop '01*, 2001.
- [3] S.Sundresh *et al.*, "Sens: A sensor, environment and network simulator," in *ANSS '04*. Washington, DC, USA: IEEE Computer Society, 2004, p. 221.
- [4] P.Levis *et al.*, "Tossim: accurate and scalable simulation of entire tinyos applications," in *SenSys '03*. New York, NY, USA: ACM, 2003, pp. 126–137.
- [5] B. L.Titzer *et al.*, "Avrora:z scalable sensor network simulation with precise timing," in *IPSN '05*. Piscataway, NJ, USA: IEEE Press, 2005, p. 67.
- [6] L.Girod *et al.*, "A system for simulation, emulation, and deployment of heterogeneous sensor networks," in *SenSys '04*. New York, NY, USA: ACM, 2004, pp. 201–213.
- [7] A.Mainwaring *et al.*, "Wireless sensor networks for habitat monitoring," in *WSNA '02*. New York, NY, USA: ACM, 2002, pp. 88–97.
- [8] I.Vasilescu *et al.*, "Data collection, storage, and retrieval with an underwater sensor network," in *SenSys '05*. New York, NY, USA: ACM, 2005, pp. 154–165.
- [9] J.Burrell *et al.*, "Vineyard computing: Sensor networks in agricultural production," *IEEE Pervasive Computing*, vol. 3, no. 1, pp. 38–45, 2004.
- [10] MoteLab, "Harvard sensor network testbed." [Online]. Available: <http://motelab.eecs.harvard.edu/>
- [11] Qualnet. [Online]. Available: <http://www.scalable-networks.com>
- [12] M.Varshney *et al.*, "Senq: a scalable simulation and emulation environment for sensor networks," in *IPSN '07*. New York, NY, USA: ACM, 2007, pp. 196–205.
- [13] "Stochastic model estimation of network time variance," white Paper, Symmetricom.
- [14] M.Varshney and R.Bagrodia, "Detailed models for sensor network simulations and their impact on network performance," in *MSWIM '04*. New York, NY, USA: ACM, 2004, pp. 70–77.
- [15] X.Zeng *et al.*, "Glomosim: a library for parallel simulation of large-scale wireless networks," *SIGSIM Simul. Dig.*, vol. 28, no. 1, pp. 154–161, 1998.
- [16] S.McCanne and S.Floyd, "Network simulator ns-2." [Online]. Available: <http://www.isi.edu/nsnam/ns>
- [17] M.Maróti *et al.*, "The flooding time synchronization protocol," in *SenSys '04*. New York, NY, USA: ACM, 2004, pp. 39–49.
- [18] C.-C.Han *et al.*, "A dynamic operating system for sensor nodes," in *MobiSys '05*. New York, NY, USA: ACM, 2005, pp. 163–176.
- [19] L. F.Perrone and D. M.Nicol, "Network modeling and simulation: a scalable simulator for tinyos applications," in *WSC '02*, 2002, pp. 679–687.
- [20] L.Girod *et al.*, "Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks," *ACM Trans. Sen. Netw.*, vol. 3, no. 3, p. 13, 2007.
- [21] J.Polley *et al.*, "Atemu: A fine-grained sensor network simulator," in *SECON'04*, 2004, pp. 145–15.