RaPTEX: Rapid Prototyping Tool for Embedded Communication Systems

JUN BUM LIM, BEAKCHEOL JANG, SUYOUNG YOON, MIHAIL L. SICHITIU, and ALEXANDER G. DEAN
North Carolina State University

Advances in microprocessors, memory, and radio technology have enabled the emergence of embedded systems that rely on communication systems to exchange information and coordinate their activities in spatially distributed applications. However, developing embedded communication systems that satisfy specific application requirements is a challenge due to the many tradeoffs imposed by different choices of underlying protocols and their parameters. Furthermore, evaluating the correctness and performance of the design and implementation before deploying it is a nontrivial task due to the complexity of the resulting system. This article presents the design and implementation of RaPTEX, a rapid prototyping tool for embedded communication systems, especially well suited for wireless sensor networks (WSNs), consisting of three major subsystems: a toolbox, an analytical performance estimation framework, and an emulation environment. We use a hierarchical approach in the design of the toolbox to facilitate the composition of the network stack. For fast exploration of the tradeoff space at design time, we build an analytical performance estimation model for energy consumption, delay, and throughput. For realistic performance evaluation, we design and implement a hybrid, accurate, yet scalable, emulation environment. Through three use cases, we study the tradeoff space for different protocols and topologies, and highlight the benefits of using RaPTEX for designing and evaluating embedded communication systems for WSNs.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]—*Real-time and embedded systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; I.6 [**Simulation and Modeling**]

General Terms: Design, Verification, Experimentation

Additional Key Words and Phrases: Wireless sensor networks, rapid prototyping tool, analytical performance modeling, real code simulation, RaPTEX, TinyOS

ACM Reference Format:

Lim, J. B., Jang, B., Yoon, S., Sichitiu, M. L., and Dean, A. G. 2010. RaPTEX: Rapid prototyping tool for embedded communication systems. ACM Trans. Sensor Netw. 7, 1, Article 7 (August 2010), 40 pages. DOI = 10.1145/1806895.1806902 http://doi.acm.org/10.1145/1806895.1806902

7

This work was sponsored by the National Science Foundation under grant number CNS-0509162.
Authors' addresses: Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27606; email:{jlim,bjang,syoon2,mlsichi,alex_dean}@ncsu.edu.

DOI 10.1145/1806895.1806902 http://doi.acm.org/10.1145/1806895.1806902

1. INTRODUCTION

Owing to advances in microprocessor, sensor, and radio technology, we are gradually moving toward an era of pervasive computing [Weiser 1991]. Tiny smart devices will be deployed and interconnected to be interwoven in the fabric of our everyday life. It is clear that this trend leads to a significant increase in diverse needs and uses of embedded systems that deeply rely on communication systems to exchange information and coordinate their activity in spatially distributed applications.

The wide range of needs for networked embedded systems has led to the creation of many different network protocols. For example, J1850, LIN, and CAN were developed as networking bus-systems for data exchange between electronic control units in vehicles. BACnet, LON, and DALI were devised for building automation systems such as smart elevators, light control, and security. In the Wireless Sensor Network (WSN) area, numerous communication protocols [Ye et al. 2004, 2006; Buettner et al. 2006; Polastre et al. 2004; Woo et al. 2003] have been proposed to meet the different performance requirements such as energy consumption, delay, and throughput.

With the large number of existing protocols, there is little reason for a system designer to reinvent the wheel and design a custom communication protocol for a given application. It is very likely that, for every possible communication need, there already exists a protocol, or a protocol suite that can be easily customized to meet that need.

However, many of these protocols may not be known to system designers, especially nonspecialists, and there are often obvious tradeoffs from different configurations or combinations of different protocols. Furthermore, there is no protocol that performs best in every situation, as shown in Section 6. Therefore, deciding the best communication protocols that fit best to a specific purpose is one of the most time-consuming and difficult tasks for designers of distributed embedded systems.

In addition to choosing protocols, it is required for system designers to systematically validate their design choices and implementations before deploying into the real environment by evaluating the expected system performance. Generally, there are three basic methods for predicting the performance of a network system: theoretical analysis, network simulations, and testbed implementations. Each of these method has the potentials and limitations which have been widely discussed in literature, (e.g., Jain [1991]).

While theoretical analysis produces immediate performance results and offers elegant solutions for estimating the performance of simple networking systems, accuracy is often sacrificed to oversimplifications and assumptions. Simulation is probably the most popular method, which can predict the performance of complex and large networking systems that are theoretically very difficult to analyze without the cost in time and money required for a testbed. The accuracy of the result, however, is dependent on the accuracy of the simulation model, which needs to be written separately from the real implementation. As a result, simulation has not been able to completely supplant the use of real hardware. Finally, a testbed usually produces the most accurate results

because it runs real implementations and takes into account most of the real factors such as signal propagation, interference, and other deployment variables. The main drawback of using testbeds is that it is time consuming and expensive. Furthermore, very large systems (e.g., large sensor networks) are difficult or impossible to replicate in a testbed. Therefore, any single evaluation method limits the accuracy of the result and/or the network size, while utilizing multiple methods requires a considerable effort from experienced system designers.

Our work is motivated by the fact that system designers, especially non-network specialists, face difficulties in developing embedded communication systems with best-fit protocols and in evaluating them. The main goal of the work presented in this article is to bridge the gap between the need for embedded communication systems and the difficulty in their design, implementation, and evaluation. To this end, we have developed a tool enabling rapid prototyping of embedded communication systems, which consists of three major subsystems:

—The first subsystem provides a collection of commonly used communication protocols and a graphical toolbox in which users can build a protocol stack for each node, virtually deploy the nodes, and compose a customized source code. The resulting network stack is used as an input for the other two subsystems, which evaluate the performance of the resulting system.
—The second subsystem offers a method of performing a fast analytical performance estimation for the selected protocols to allow users to quickly evaluate the theoretical performance of the system. This feature enables informed choices and a fast but thorough exploration of the offered tradeoff space.
—Finally, RaPTEX provides an assembly code-level emulation environment to execute the complete source code for each network node with a simulation for physical layer effects and node mobility, while facilitating debugging and postprocessing for performance analysis.

With these three subsystems, RaPTEX offers the user the power of iterative design, numerous protocol choices and configurations, and fast and accurate performance estimation of the system. Hence, the user will be able to iteratively adjust the design choices until the desired performance is achieved. When the system performance is satisfactory, a single command is used to generate the byte code to be uploaded to the microcontrollers of the embedded system.

Because embedded communication systems cover a very broad area, in this article, we focus on the specific application domain of Wireless Sensor Networks (WSNs) based on TinyOS components and the Berkeley motes. For illustration purposes, the current RaPTEX communication library contains two MAC and two routing protocols available with TinyOS. The emulation environment supports Mica2 as its default hardware platform. RaPTEX is also being used for creating ultrasonic underwater communication systems for sensor networks [Peng et al. 2010; Parsons et al. 2008].

The remainder of this article is organized as follows. We provide a summary of the related work and its limitations in Section 2. In Section 3, the communication library and toolbox are described in detail. Analytical and emulation-based performance evaluations are explained in Sections 4 and 5, respectively. We then present how RaPTEX is useful in real system development process through three use cases in Section 6. Finally, we conclude the article and present future directions in Section 7.

2. RELATED WORK

Building an embedded communication system, especially an application for WSNs, is a highly complex task, which involves a variety of design, development, and validation methodologies and tools, making it a nontrivial and errorprone task. Previous projects have partially facilitated each of the developmental steps, but few, studies provided an integrated environment that covers all the developmental procedure with an overall network perspective.

2.1 Developmental Tools

There have been several projects that have provided a developmental environment to facilitate the software design and implementation through graphical composition tools, high-level language, and libraries. TinyDT [Sallai et al. 2005], SNACK [Greenstein et al. 2004], and GRATIS [Volgyesi and Ledeczi 2002] are development tools for WSN applications based on TinyOS.

TinyDT [Sallai et al. 2005] is a TinyOS plug-in for the Eclipse platform. It focuses on providing editing functionalities such as syntax validation, code navigation, and static wiring graph analysis. SNACK [Greenstein et al. 2004] is a sensor network application construction kit, which consists of a new component composition language, configurable high-level service libraries, and a new compiler. Within the boundary of high-level services, SNACK facilitates application design, but the process of developing configurable services is not trivial. Moreover, SNACK requires learning a new language and compiler-specific functionality. GRATIS [Volgyesi and Ledeczi 2002] features graphical block composition interfaces that provide a clear overview of the TinyOS application structure and make it possible for nonspecialists to easily develop WSN applications by simply choosing preexisting nesC components. However, GRATIS only supports source code-level components (nesC components), limiting the ability of tuning the high-level service components. Moreover, none of these tools supports a validation process.

In RaPTEX, we provide the graphical toolbox with predefined high-level protocol components to help users build a protocol stack in an easy and consistent way. Through RaPTEX's top-down approach, a user can explore the system from the system view to a single node. By interconnecting software components in different diagrams, users can easily develop, configure, and deploy their systems. The resulting system is used as an input to the other subsystems for theoretical analysis and emulation-based evaluation without any modifications.

2.2 Analytical Models

Before committing to a particular implementation, designers often use an analytical model for the newly proposed algorithm or designed system to evaluate the performance and explore alternatives.

Many researchers used stochastic analysis to model the performance of WSNs characterized by sleep and active dynamics. By modeling the state of the sensor nodes as Markov chains while considering channel contention and routing issues, performance measures such as energy consumption, data delivery, delay, and network capacity were studied by Chiasserini and Garetto [2004]. Kwon and Agha [2006] also used a discrete Markov chain with a specialized iLTL analyzer.

Some energy-efficient sensor MAC protocols, such as BMAC [Polastre et al. 2004], XMAC [Buettner et al. 2006], and SCPMAC [Ye et al. 2006] also provide analytical models for their energy efficiency. Energy consumption in these models is approximately predicted by calculating how much time is spent in each mode of radio operations. Although these models are simple and offer the insight of the proposed protocols, the protocol-specific assumptions make it difficult to fairly compare different protocols, and analysis is isolated to the MAC in a single-hop network, making it difficult to estimate the influence of protocols at other layers in a multihop environment.

The analytical model adopted in RaPTEX uses the same assumptions for all protocols included in RaPTEX's communication library, allowing users to fairly compare different protocols. The model also uses values from real implementations and takes into account the position and amount of traffic from neighbor and children nodes to increase the accuracy of the estimation in a multihop environment. The accuracy of RaPTEX's analytical model is presented in Section 4.3.

2.3 Simulation

Many simulation-based approaches have also been proposed to validate system designs. Based on classical discrete event network simulators such as ns-2 [Rice University 1999], OMNeT++ [Varga 2001], and J-Sim [Sobeih et al. 2005], node and wireless specific simulation models were added.

SensorSim [Park et al. 2003] extends ns-2 to make it suitable for WSN specific simulations by incorporating a power model, channel sensing, and hybrid simulations with real nodes. JSim [Sobeih et al. 2005], which is a Java-based general-purpose simulator with a component-oriented architecture, provides a sensor network package featuring a power model, a sensor model for phenomenon detection, and connection of real hardware to the simulator. Castalia [NICTA 2009] is a WSN simulator based on OMNeT++, providing configurable radio, channel, and MAC protocol models. In NesCT [Dulman et al. 2005], the nesC syntax is translated into C++ for OMNeT++ simulation using source-to-source compilers.

Due to their model-based nature, which is relatively flexible and scalable, these high-level simulators are useful when looking at systems from a high-level and are suitable for the first-order validation of an algorithm before

moving to implementation on a specific platform. The effect of channel models, topology, mobility, and routing protocols can be appropriately inspected at this level.

However, the coarse-grained representation of the microcontroller, the abstraction of low-level protocols, and the packet-level interaction limit the use of these simulators for studies that require highly accurate low-level timing interactions or fine-tuning, or for debugging implementations of distributed algorithms.

Instead of providing a separate simulator, RaPTEX adopts a hybrid approach in which we integrate an instruction-level emulator and a network-level simulator to combine the benefits of both. From the network-level simulator, RaPTEX inherits the power of node mobility and diversity in channel models. The accuracy (clock cycle level) of single node execution and inspection of low-level run-time properties such as interrupt monitoring, energy consumption tracking, memory monitoring, serial monitoring, or byte-level packet interactions are guaranteed by the underlying emulator.

2.4 Emulation

In order to accurately inspect and verify detailed run-time properties, emulators, which run actual source codes, have been widely used.

TOSSIM [Levis et al. 2003] is a TinyOS-specific network emulator that runs nesC code without any modification except hardware control codes that are replaced by linking to libraries that emulate the behavior of devices. This hardware abstraction leads to a lack of accuracy caused by the loss of fine-grained timing information and prevents low-level inspection (e.g., monitoring interrupts or the behavior of low-level protocols). Power-TOSSIM [Shnayder et al. 2004], an extension of TOSSIM, provides a power model for sensor nodes. Although Power-TOSSIM inherits the high scalability of TOSSIM, the hardware abstraction of TOSSIM makes it impossible to compare different duty cycle MAC protocols. By supplementing a network simulator with the rich and detailed models and capability of node emulation, sQualNet [Varshney et al. 2007] achieves enhanced fidelity, scalability, and heterogeneity. It, however, takes a similar approach to emulation as TOSSIM. Therefore, the drawbacks resulting from using a hardware abstraction remain.

The most accurate approach is an instruction-level emulator that takes assembly code as an input and runs the code at the instruction cycle granularity, emulating hardware devices such as the processor and the radio chip. Both ATEMU [Polley et al. 2004] and Avrora [Titzer et al. 2005] are cycle-accurate AVR emulators. ATEMU offers very accurate results by synchronizing every node at every clock cycle, but, at the same time, it sacrifices the speed and scalability for exact synchronization.

To overcome the scalability problem of ATEMU, Avrora adopts a new synchronization strategy in which multiple nodes run simultaneously as separate threads during a given interval; the default interval is 1 byte time, (i.e. the time to send 1 byte). When a node has to synchronize with its neighbors, for example, to receive data or to perform carrier sense, the node waits

until every node reaches the same simulation time. Therefore, it is allowed for a node to be further ahead in simulation time than other nodes until synchronization is required. By adopting this synchronized run-and-wait approach, Avrora achieves better scalability than ATEMU, while maintaining accuracy.

However, the Avrora's CPU-oriented nature lacks network-level supports such as diversity in channel models, mobility, and topology flexibility. In addition, because Avrora's multithreaded structure uses a shared memory, it can only be run on a single machine. Thus, Avrora has limited scalability, presented in more detail in Section 5.5.

For accurate emulation, RaPTEX adopts Avrora as an underlying emulator. However, to overcome Avrora's drawbacks, we strengthen network-level support by integrating it with a network simulator, OMNeT++ [Varga 2001], and extend Avrora's synchronization mechanism to distribute the emulation over several host machines. The resulting scalability properties are presented in Section 5.5.

2.5 Integrated Development

Despite the availability of several tools facilitating the development and evaluation of WSN systems, few systems provide an integrated developmental and evaluation environment for WSNs.

Worldsens [Fraboulet et al. 2007] is a prototyping tool for WSNs based on two different level of simulators. The tool helps users validate the high-level designs using a network-level simulator (WSNet), and verify the implemented system through an instruction-level simulator (WSim). Although this tool draws from the advantages of the two simulators and enables the cross-validation between different developmental steps, there is no consistent support for system design and real source code implementation. In RaPTEX, we provide a graphical toolbox with a set of commonly used protocols as component blocks to help users compose a network stack, configure each protocol, and deploy virtual network nodes. The configuration and composed source code are used as inputs to the other subsystems for theoretical analysis and emulation-based evaluation without any modification (shown in Figure 1).

Mozumdar et al. [2008], presented a framework that allows users to build and simulate an application algorithm and generate application code for a specific OS, based on MathWorks [The MathWorks, Inc. 1994] tools. Although the framework facilitates application development and validation, the main focus of the framework is on the functional correctness of an application logic itself, not on the entire system (e.g., underlying protocols). The code generator also focuses on generating code for application-level logic, although often the expected performance and correctness of an embedded communication system cannot be accurately determined without considering the underlying protocols. The model-based simulation with the simple packet-level interactions from the fixed two-dimensional connection graph also introduces inaccuracies into the simulation. Being based on commercial software, unfortunately, also prevents it from being widely used.

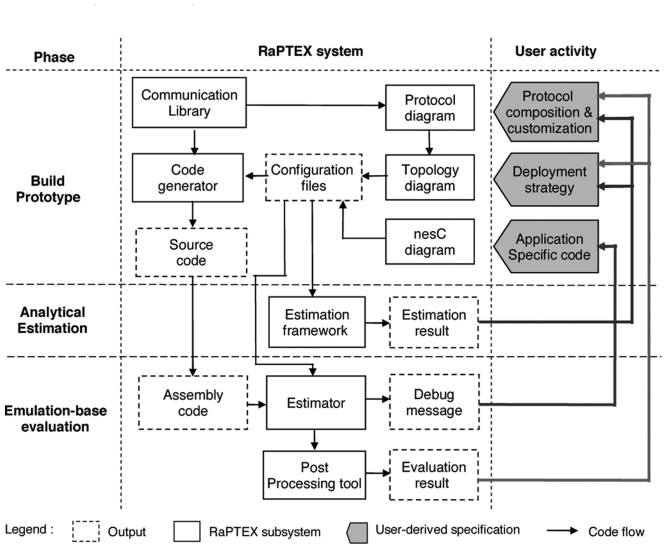

Fig. 1. Developmental process using RaPTEX.

Viptos [Cheong et al. 2005] provides a graphical developmental environment for the nesC component composition and a simulation environment in which TOSSIM [Levis et al. 2003] and VisualSense [Baldwin et al. 2004] are merged into a single simulator. It allows a user to easily explore TinyOS application structures and to simulate actual TinyOS programs. Although the current Viptos simulator only supports TinyOS-based systems, it can be extended to support other languages and can be merged with a non-TinyOS simulator.

A very important feature of RaPTEX is the capability to theoretically estimate the performance of the selected communication protocols, providing instant feedback to the user. This enables informed choices and a thorough exploration of the offered tradeoff space. Furthermore, RaPTEX realistically emulates the designed system using a functional, cycle-accurate emulation of each component of the system and simulation of common physical layer effects. The emulation will enable exact measurements of relevant performance parameters and resource requirements, while facilitating the debugging of these notoriously difficult-to-debug distributed systems.

3. TOOLBOX AND COMMUNICATION LIBRARY

The first characteristic of RaPTEX is that users can design and evaluate their systems in a single integrated toolbox, which was designed with the following requirements:

—A common code is used for both the design and evaluation stages without reimplementation of protocols and separate setup of scenarios for consistency during the entire developmental process.

—To facilitate system design, a user selects, configures, and deploys graphical components to build the desired customized communication system with a whole network perspective.

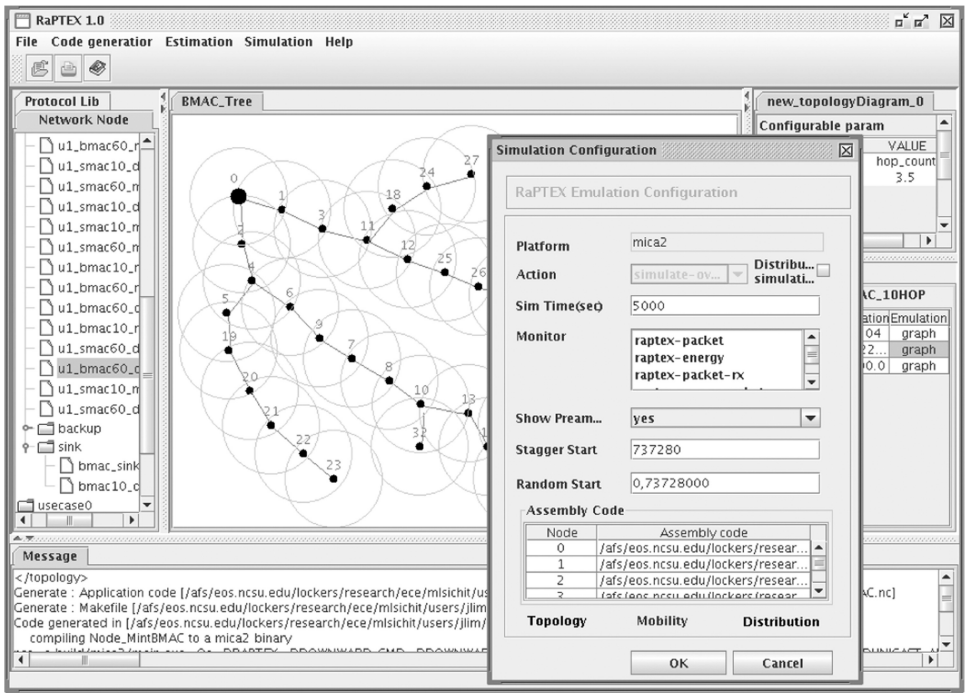

Fig. 2. Screenshot of the network topology diagram.

—The tool contains a collection of commonly used communication protocols with several tuning parameters enabling design choices.

3.1 Developmental Process

RaPTEX uses the approach shown in Figure 1 to allow straightforward protocol stack design, synthesis, and analysis. A user selects and configures a network stack from predefined protocols or directly composes source-level components in the graphical toolbox. From the user's design choices, formulae for theoretical analysis and source code with files are composed. The source code is then compiled to assembly code to be injected in the emulation. The evaluation results derived from theoretical analysis and the emulation are used to refine the system design until it satisfies the performance requirements.

3.2 Toolbox

RaPTEX's toolbox adopts a hierarchical architecture of three different graphical diagrams: the network topology diagram (Figure 2), the protocol diagram (Figure 3), and the source component (nesC) diagram (Figure 4). This architecture allows easy transitions between a network-level view of a system to low-level node implementations. Each diagram handles a different type of XML file.

In the network-level view, RaPTEX provides the network topology diagram (Figure 2), where a user can deploy virtual network nodes. For the topology

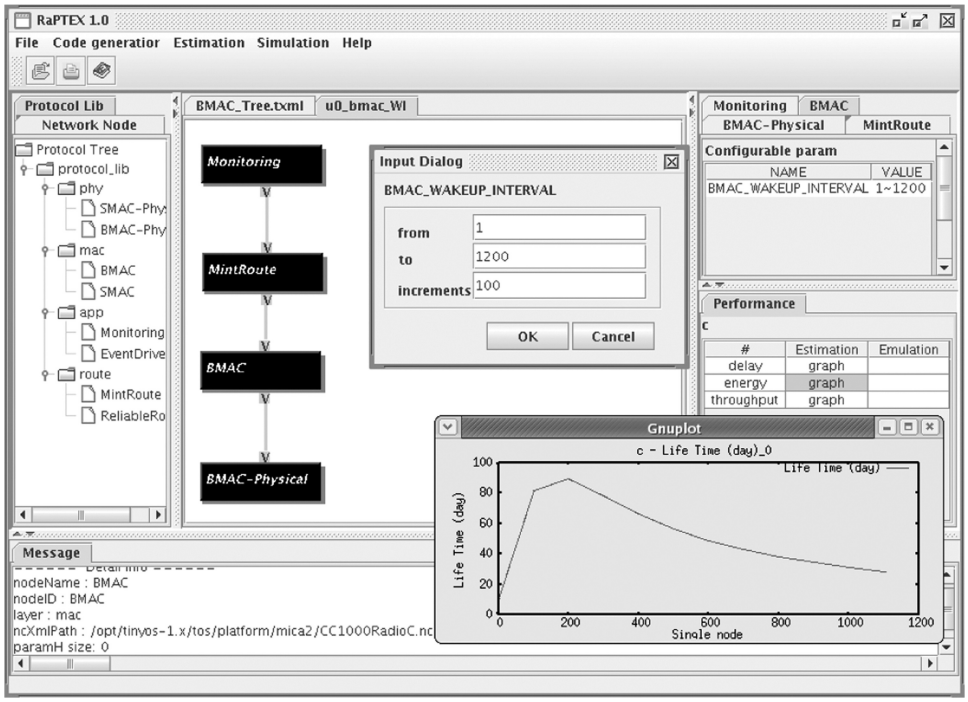

Fig. 3. Screenshot of the node protocol diagram.

representation, we define an XML format called TXML, which contains information about the locations of each node, and each node holds a pointer to another XML (NXML) file referring to detailed node information.

For the theoretical estimation in Section 4, the TXML file is used to calculate the expected number of packet transmissions and receptions of each node as it contains information about neighbor nodes and routing paths toward the selected sink nodes. For the emulation in Section 5, the locations of each node in the TXML are converted to the topology description script in the `omnetpp.ini` file of OMNeT++ [Varga 2001] and used as initial values for mobility patterns of each node. The GUI of the emulation environment is shownt in Figure 5.

In a node-level view, the protocol diagram (Figure 3) is provided, in which the user composes and configures the network stack for a node. For example, in Figure 3, we choose BMAC [Polastre et al. 2004] as a MAC protocol and MintRoute [Woo et al. 2003] as a routing protocol and configure the BMAC's wakeup interval (BMAC_WAKEUP_INTERVAL). To represent a node, we define an XML format named *NXML*, which only contains information about what protocols are selected and how they are interconnected. An NXML file consists of PXML files; each PXML holds outline information about a protocol, including the path to source components, configurable parameters, and interfaces to different layers for the compatibility check.

To allow a user to easily handle protocol source code directly, RaPTEX provides the source component diagram (Figure 4). With this diagram, a user can

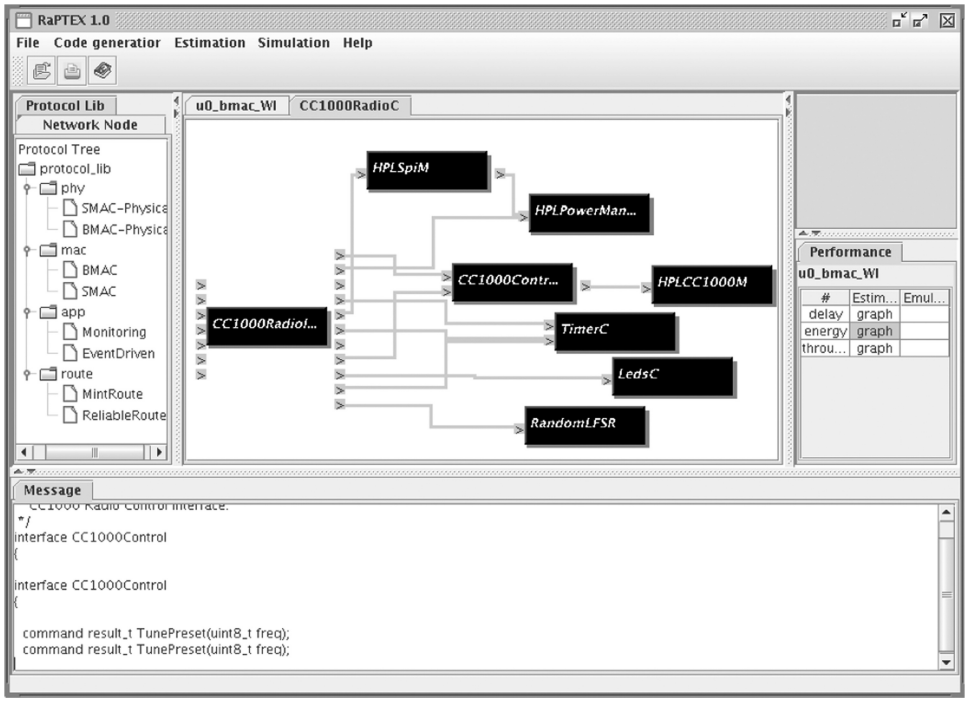

Fig. 4. Screenshot of the source component (nesC) diagram.

navigate each module of the code and directly edit the code. In this version of RaPTEX, we support nesC for TinyOS as the source component diagram. To represent the nesC components, we use the existing capability of the nesC compiler, which generates outline information of nesC components in an XML format by using the `-fnesc-dump` option.

The RaPTEX code generator composes the source files including the selected template application code, the selected protocols, and a Makefile that glues together all the source files and the user's configurations. The composed prototyping source code is used for the emulation as presented in Section 5 or uploaded to the real device without any modifications. If the performance of the prototyping system is acceptable, application developers can concentrate on application logic by modifying the template application code without modifying the details of protocol-level implementation.

3.3 Communication Library

To enable a wide range of design choices, especially for low-energy budget applications for WSNs, RaPTEX provides a communication library with a suite of protocols featuring several options at each layer. For flexibility and interoperability, each class of protocols has an identical interface and different packet types are unified into a single format by transforming packets between the MAC and routing layers. For consistency in the configuration and performance evaluation, we define common configuration interfaces and add interfaces for

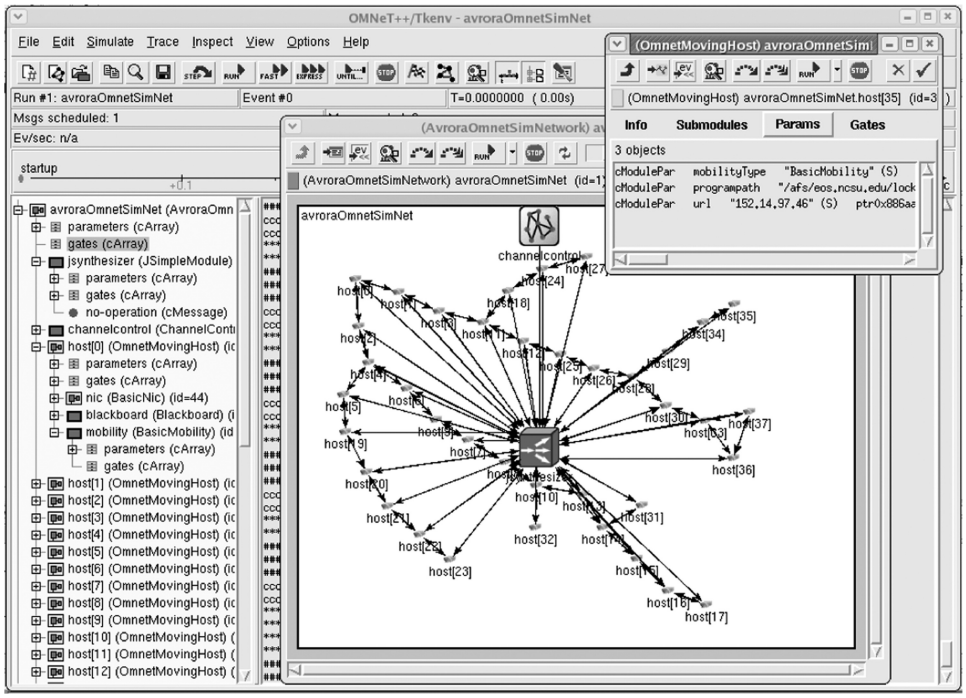

Fig. 5. Screenshot of the emulation environment over OMNeT++.

debugging and energy measurement. Although the internal composition of a protocol and its tunable parameters are protocol specific, the top-level protocol component is exposed through common interfaces and described as a PXML. Therefore, RaPTEX users can treat the protocol as a form of high-level graphical design components in the protocol diagram, and swap out different protocols easily.

We instantiate the communication library based on TinyOS components. Figure 6 illustrates the layered structure of each class of protocols and their interfaces. Since our focus is on providing users with an easy way of choosing protocols and tuning them based on existing protocols, we reuse the existing TinyOS interfaces as much as possible. Although many protocols in TinyOS are implemented through the common interfaces, in cases when a protocol is implemented with proprietary interfaces or tunable parameters are not explicitly exposed (e.g. changing parameters requires changing source code), we modify the protocol to explicitly expose configurable parameters and its functionality through the common interfaces.

For packet I/O, each layer provides and uses `SendMsg` and `ReceiveMsg`. The routing layer provides the `Intercept` interface to allow the application layer to snoop or aggregate multihop messages. `RouteControl` and `MACControl` allow control of the operations of protocols such as `start/startDone()`, `stop/stopDone()`, and `init/initDone()`. The `RaptexEnergy` enables measurements of the energy consumption by tracking the total time a radio device

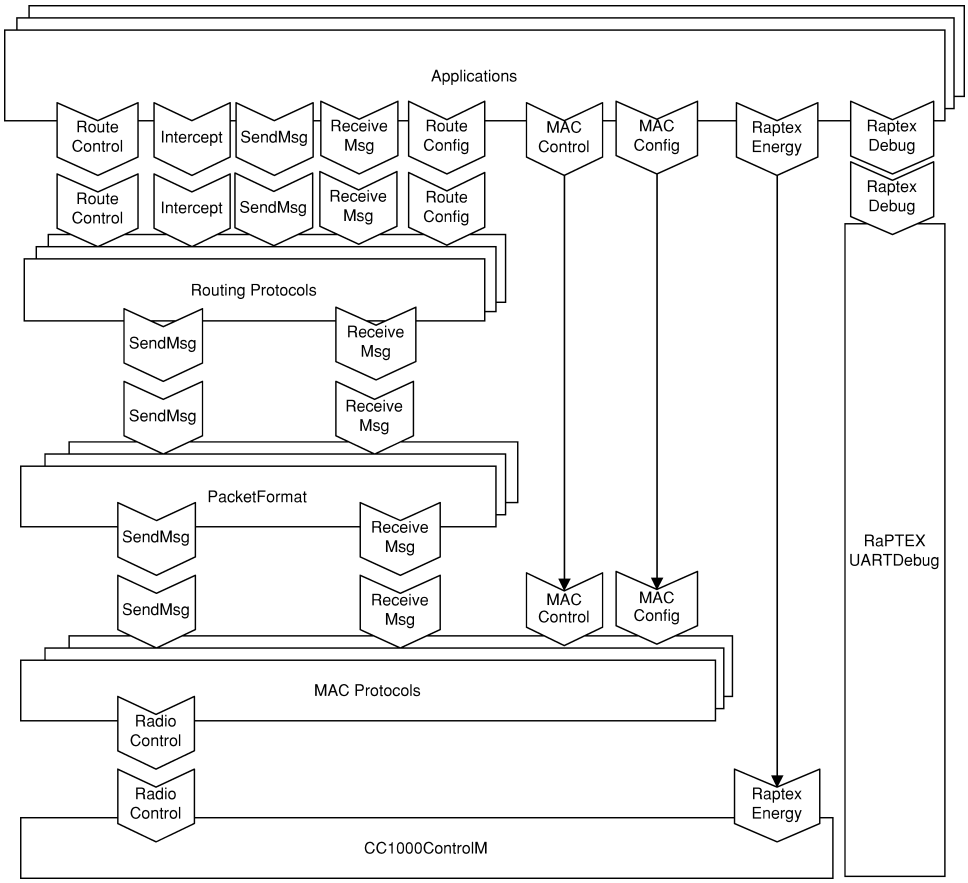

Fig. 6. Interfaces for the RaPTEX communication library for TinyOS.

spend in each mode of radio operations. To allow each protocol to print debug messages through the UART, we define `RaptexDebug`, which cooperates with the `SerialMonitor` of the RaPTEX emulator for the debugging purpose described in Section 5. Because each class of protocols exposes identical interfaces to upper/lower layers, swapping different protocols becomes possible. However, some protocols also need to expose protocol-specific interfaces, especially for tunable behaviors.

There are two ways to define tunable parameters. First, the parameters are simply defined using `#ifndef` and `#define` in the header files. Second, the parameters are accessed through functions defined in protocol-specific configuration interfaces (e.g., `LowPowerControl` shown in Figure 7) and the interfaces are gathered and implemented in a configuration component such as `RouteConfig` and `MacConfig` shown in Figure 8. By changing the connection (wiring) between the protocol logic component (e.g., `RaptexComLibBmacM`) and the configuration component, the actual value of the tunable parameters are decided in a flexible manner, and it also enables preprocessing before being used in the protocol

```
interface LowPowerControl {
  async event int16_t getSleepTime();
  async event int16_t getPreambleLen();
}
```

Fig. 7. The LowPowerControl, which is implemented in BMAC's `MacConfig`.

```
configuration MacConfig {
    provides interface StdControl;
}implementation {
  components MacConfigBMACM, RaptexComLibBmacM, CC1000ControlM, RandomLFSR;

    StdControl = MacConfigBMACM.StdControl;
    MacConfigBMACM.MacControl -> RaptexComLibBmacM.MacControl;
    MacConfigBMACM.MacBackoff -> RaptexComLibBmacM.MacBackoff;
    MacConfigBMACM.LowPowerControl -> RaptexComLibBmacM.LowPowerControl;
    MacConfigBMACM.CC1000Control -> CC1000ControlM.CC1000Control;
    MacConfigBMACM.Random -> RandomLFSR;
}
```

Fig. 8. The `MacConfig` for BMAC, which bridges the BMAC protocol implementation and tunable parameters.

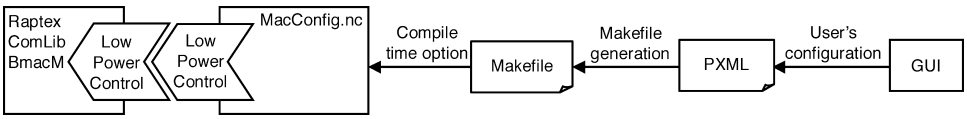

Fig. 9. Steps of parameter passing from the GUI to protocol implementation.

logic components. Protocol logic components are wired with default configuration components for the case when there is no external wiring.

For example, Figure 9 shows how tunable parameters are passed from the GUI to source code. User configurations are passed to the source code as compile time options (`-D`), and `MacConfig` (Figure 8) that is wired with the real protocol component (`RaptexComLibBmacM`) allows the parameters to be accessed by the protocol through function calls. For example, the `LowPowerControl` (Figure 7) defines two functions that will be called by the BMAC implementation. The detail of the functions are decided in BMAC's `MacConfig` at compile time based on the compile option (e.g., BMAC_WAKEUP_INTERVAL=200) passed by the Makefile generated by the RaPTEX code generator. Code snippets of the top-level Makefile are shown in Figure 21 in Section 6.

The current communication library of RaPTEX contains two well-known energy efficient MAC protocols for TinyOS, BMAC [Polastre et al. 2004] and SMAC [Ye et al. 2004]. It also includes DefaultRoute [Buonadonna 2003] and MintRoute [Woo et al. 2003] at the routing layer. In the application layer, we provide two types of templates for continuous monitoring and event-driven applications [Madden et al. 2003], to which users can add their specific codes. The tunable parameters for each layer are summarized in Table I. Tradeoffs from different combinations of these protocols and parameters are discussed in Section 6.2. Although the current communication library does not cover

Table I. Tunable Parameters in RaPTEX

Parameter	Default
Data packet generation interval	10 s
Expected event rate during a sample period	1
Routing packet transmission rate	20 s
Routing forwarding queue size	16 Packets
Routing link estimator period	200 s
BMAC acknowledgment	No
BMAC wakeup interval	20 ms
BMAC Initial backoff window size	32 Slots
BMAC congestion backoff window size	32 Slots
SMAC maximum number of RTS	7
SMAC wakeup interval (duty cycle)	257 ms (50%)
SMAC sync packet period	12000 ms
SMAC data retransmission limit	3
SMAC max. number of different schedules	4
SMAC max. number of neighbors	20
Transmission power (1-255)	15
Frequency	915 MHz

all the TinyOS protocols available, we believe that these protocols constitute a representative sample of existing protocols for WSNs, and by standardizing the top-level interfaces and generating a new PXML, one can add a new protocol in RaPTEX's communication library.

For example, to add SCPMAC [Ye et al. 2006] in the communication library, outline information of the protocol should be described in a new PXML after standardizing the protocol implementation for the common interfaces and packet format. Figure 10 shows code snippets of the PXML, which holds outline information about the protocol, including the path to the top-level source component, the class path for the estimation model, the configurable parameters, and the interfaces to different layers for the compatibility check. Since the parameters defined in the PXML are also defined in the source code as `#define` or SCPMAC's `MagConfig`, the Makefile generated by RaPTEX code generator will bridge the parameters and real source code at compile time. The analytical estimation model [Yoon 2007] for SCPMAC will be invoked by the estimation framework, which is presented in Section 4.

By generating the same format of the XML files, RaPTEX can also support other sets of protocols for different software platforms. For example, to support MANTIS [Bhatti et al. 2005], outline information of existing protocols implemented with the interfaces of MANTIS should be described in PXML. A template application using NET or COM interface should be written to provide users with the start point of writing application logic. The theoretical estimation formula for the protocols can be written by modifying the model in Section 4. If the mechanism of a protocol is same as the one in TinyOS, for example, cc1000_bmac in MANTIS, then the formula can directly be reused by defining the same Java class path for the formula in the PXML. Since an assembly code is used for the emulation, no additional support is required if the hardware platform is the same.

```
<protocol>
    <property name="id">SCPMAC</property>
    <property name="layer">mac</property>
    <property name="ncxml_path">/raptex/comlib/mac/scpmac/RaptexSCP.ncxml</property>
    <property name="model_path">/defaultmodel/mac/scpmac</property>

    <parameter>
        <property name="name">SCP_POLL_PERIOD</property>
        <property name="degree">msec</property>
        <property name="default">1024</property>
        <property name="description">period for each node to poll the channel</property>
        <property name="type">
            <property name="value">double</property>
            <property name="min">10</property>
            <property name="max">614400</property>
            <property name="granularity">1</property>
        </property>
    </parameter>

    <parameter>
        <property name="name">SCP_SYNC_PERIOD</property>
        <property name="degree">msec</property>
        <property name="default">614400</property>
        <property name="description">period for each node to poll the channel</property>
        <property name="type">
            <property name="value">double</property>
            <property name="min">1</property>
            <property name="max">3686400</property>
            <property name="granularity">1</property>
        </property>
    </parameter>

     :
     :

    <interface>
        <property name="type">mac.upper.ReceiveMsg</property>
        <property name="name">SCPMAC.uppergate</property>
        <property name="id">SCPMAC.uppergate</property>
        <property name="is_provide">yes</property>
    </interface>

    <interface>
        <property name="type">mac.upper.SendMsg </property>
        <property name="name">SCPMAC.uppergate</property>
        <property name="id">SCPMAC.uppergate</property>
        <property name="is_provide">yes</property>
    </interface>

     :
     :

</protocol>
```

Fig. 10. Code snippet of an example PXML for SCPMAC.

4. ANALYTICAL PERFORMANCE ESTIMATION

It is important that at least a rough approximation of the projected performance is available at the time of the design of the protocol stack. Thus, the goal of the RaPTEX's analytical performance estimation is to allow users to quickly explore the performance tradeoffs involved when choosing certain protocols or changing parameters.

Table II. Symbol Used in Analytical Model

Symbol	Meaning
E	Energy consumed during T
T	Sampling period
T_X	Time spent for X during T, where $X \in \{tx, rx, overhear, idle, startup, sleep\}$
P_X	Power consumed for X, where $X \in \{tx, rx, oh, idle, startup, sleep\}$
E_X	Energy consumed for Y during T, where $X \in \{tx, rx, oh, idle, startup, sleep\}$
t_f	Frame size ($t_f = t_{sleep} + t_{listen}$)
t_{listen}	Active listening period within t_f
K	Number of frames during one sample period T ($T = T \cdot t_f$)
t_{sleep}	Sleeping period within the frame
t_{lpl}	Time for low-power listening
$t_{sync,interval}$	Synchronization packet interval
t_X	Time to send/receive a packet of type X, where $X \in \{data, rts, cts, ack, sync\}$
$t_{preamble}$	Time to send a preamble
t_{wakeup}	Time for radio transition from sleep to a fully awake state
$t_{route,interval}$	Routing packet transmission interval
ζ_i	Set of children of node i
Υ_i	Set of nodes whose transmissions can be received by node i
$N_e(i)$	Number of events that node i expects to detect during T
$N_{hop}(i)$	Number of hops from node i to sink
$N_{neighbor}(i)$	Number of neighbors of node i (cardinality of Υ_i)
$N_{tx,X}(i)$	Number of packets of type X that node i sent during T, where $X \in \{data, ack, rts, cts\}$
$N_{rx,X}(i)$	Number of packets of type X that node i received during T, where $X \in \{data, ack, rts, cts\}$
$N_{hear,X}(i)$	Number of packets of type X that node i hears during T, where $X \in \{data, ack, rts, cts\}$
$N_{tx,route}(i)$	Number of routing packets sent during T
$N_{rx,route}(i)$	Number of routing packets received from its neighbors during T

In this section, we determine analytically the power consumption, end-to-end average delay, and throughput of several well-known MAC protocols for WSNs. Since the lifetime is the most important design issue in WSNs [Ye et al. 2004], we mainly focus on the power consumption by calculating how much energy is consumed in each operation mode of the transceiver. To validate the analytical model, we measure energy consumption, end-to-end delay, and throughput of both single-hop and multihop networks on a testbed using Mica2 motes and compare the results.

4.1 Analytical Model

The analytical model assumes that node i generates data packets when it detects events at the rate $\frac{N_e(i)}{T}$, where T is the fixed sample period and $N_e(i)$ is the number of events that node i expects to detect during T. In case of the continuous monitoring application, $N_e(i) = 1$. The packets are forwarded to the sink through the multihop network. For simplicity, the network has sufficient capacity to transport the data and there are no collisions. This assumption is a reasonable approximation of reality in systems with very low-duty cycles and low contention. The notation we use for the analysis is summarized in Table II.

The total energy consumption E during each period T is the sum of the expected energy spent in each mode of radio operations during T:

$$E = T_{tx}P_{tx} + T_{rx}P_{rx} + T_{oh}P_{oh} + T_{idle}P_{idle} + T_{startup}P_{startup} + T_{sleep}P_{sleep}, \quad (1)$$

$$T = T_{tx} + T_{rx} + T_{oh} + T_{idle} + T_{startup} + T_{sleep}, \quad (2)$$

where T_{tx}, T_{rx}, T_{oh}, T_{idle}, $T_{startup}$, and T_{sleep} are time spent for transmission, reception, overhearing, idle listening, startup, and sleeping during T, respectively, and P_x is the power needed for radio operation x.

The number of data packets that node i receives ($N_{rx,data}(i)$), transmits ($N_{tx,data}(i)$), and hears from its neighbors ($N_{hear,data}(i)$), in a multihop network during T is dependent on the topology, routing protocol, and aggregation mechanisms. For simplicity, in this model, we assume all traffic is forwarded to a single base station through a shortest-path routing tree without aggregation:

$$N_{rx,data}(i) = \sum_{k\in\zeta_i}(N_e(k)), \quad (3)$$

$$N_{tx,data}(i) = N_{rx,data}(i) + N_e(i), \quad (4)$$

$$N_{hear,data}(i) = \sum_{k\in\Upsilon_i}(N_{tx,data}(k)), \quad (5)$$

where ζ_i is the set of children of node i and Υ_i is the set of nodes whose transmissions can be received by node i. In the current RaPTEX topology diagram (Figure 2), by default, ζ_i is decided by the minimum hop count.

Depending on the routing protocol, routing packets are periodically broadcast, which influences the total number of packet transmissions and receptions. The average number of routing packets that node i transmits and receives during T (not necessarily an integer) is

$$N_{tx,route}(i) = \frac{T}{t_{route,interval}}, \quad (6)$$

$$N_{rx,route}(i) = \frac{T}{t_{route,interval}}N_{neighbor}(i), \quad (7)$$

where $t_{route,interval}$ the routing packet transmission interval, which is configurable in the protocol diagram (Figure 3).

In many MAC protocols for WSNs, nodes save energy by going to sleep if they do not have data to sense, receive, or transmit. The analytical models of MAC protocols vary depending on the energy saving mechanism that each MAC protocol uses. We classify energy-efficient MAC protocols for WSNs into two groups; synchronization-based and preamble-based.

In synchronization-based MAC protocols, the sampling period (T) is divided into fixed size frames, $T = t_f K$. Within the frame, nodes stay awake at least a portion of the frame to send/receive data, or go to sleep to save the energy according to the schedule of the MAC protocol. To send and receive packets, all nodes in a neighborhood wake up and go to sleep at the same time. Therefore, synchronization-based MAC protocols provide (and use) synchronization

mechanisms. Synchronization-based MAC protocols include SMAC [Ye et al. 2004], TMAC [van Dam and Langendoen 2003], and SCPMAC [Ye et al. 2006].

In preamble-based MAC protocols, the nodes are not synchronized with each other. Instead, senders use a long preamble to wake up receivers. In these protocols the receiver sleeps for a long time and wakes up for a very short time to check for the existence of a preamble. If the receiver does not detect any preamble, it goes back to sleep immediately. If it detects the preamble, the node goes back to sleep after performing the protocol-specific radio operations. We define a frame in the preamble-based protocols as the time combining the sleep time and the time to check the preamble. Unlike in synchronized-based protocols, the number of frames during each sampling period in preamble-based protocols varies depending on the traffic load during the sampling period. Preamble-based protocols include BMAC [Polastre et al. 2004] and X-MAC [Buettner et al. 2006].

In this article, we present the model for SMAC [Ye et al. 2004] and BMAC [Polastre et al. 2004]. The detail analytical model for other protocols such as XMAC [Buettner et al. 2006], TMAC [van Dam and Langendoen 2003], and SCPMAC [Ye et al. 2006] are described in [Yoon 2007].

4.1.1 *SMAC.* Each SMAC node sleeps for some time, and then wakes up and listens to determine if it needs to receive a packet. The sleep schedules of all nodes in a neighborhood are synchronized and nodes use SYNC packets to maintain synchronization. SMAC also provides a mechanism that handles the situation where there exist nodes with different sleep schedules (thus being suitable for multihop networks). In this case, the border nodes maintain multiple schedules to bridge the different schedules. However, to make the comparison with other protocols fair, we only consider the power consumption in a single schedule.

For one data packet, the sender spends $t_{data} + t_{rts}$ seconds for transmitting and spends $t_{cts} + t_{ack}$ seconds for receiving, and the receiver spends $t_{data} + t_{rts}$ seconds for receiving and spends $t_{ack} + t_{cts}$ seconds for transmitting. For SYNC packets, each sensor node, on the average, transmits $\frac{T}{t_{sync,interval}} \frac{1}{N_{neighbor}(i)+1}$ and receives $\frac{T}{t_{sync,interval}} \frac{N_{neighbor}(i)}{N_{neighbor}(i)+1}$ SYNC packets during one sampling period T. Therefore, the total time in each sampling period for transmitting and receiving packets are given by (8) and (9), respectively:

$$\begin{aligned} T_{tx} = {} & N_{tx,data}(i)\,(t_{data} + t_{rts}) + N_{rx,data}(i)\,(t_{cts} + t_{ack}) \\ & + t_{sync} \frac{T}{t_{sync,interval}} \frac{1}{N_{neighbor}(i) + 1} \\ & + N_{tx,route}(i) t_{data}, \end{aligned} \tag{8}$$

$$\begin{aligned} T_{rx} = {} & N_{rx,data}(i)\,(t_{data} + t_{rts}) + N_{tx,data}(i)\,(t_{ack} + t_{cts}) \\ & + t_{sync} \frac{T}{t_{sync,interval}} \frac{N_{neighbor}(i)}{N_{neighbor}(i) + 1} \\ & + N_{rx,route}(i) t_{data}. \end{aligned} \tag{9}$$

Overhearing time ($T_{overhear}$) for a sensor node i is the time spent on receiving packets meant for other nodes. Overhearing only occurs for RTS and CTS packets in listen (active) state of all frames because SMAC nodes return to sleep state when the data transmission is completed:

$$T_{overhear} = N_{overhear,rts}(i)t_{rts} + N_{overhear,cts}(i)t_{cts}. \tag{10}$$

To compute $N_{overhear,rts}$, we subtract the number of data packets that node i should receive from the total number of data packets that node i hears from its neighbors:

$$N_{overhear,rts} = N_{hear,data}(i) - N_{rx,data}(i). \tag{11}$$

The number of CTS packets that node i hears from its neighbor j is $N_{rx}(j)$, which is one less than $N_{tx}(j)$ if we assume $N_e(j) = 1$ (recall that $N_e(j)$ is the number of events that node i expects to detect during T). Therefore, from (4) and (5), the total number of CTS packets that node i hears is $N_{hear,data}(i) - N_{neighbor}(i)$. Among $N_{hear,data}(i) - N_{neighbor}(i)$, only $N_{tx,data}(i)$ are CTS packets that node i should receive because node i receives a CTS packet for each of its data packet transmissions:

$$N_{overhear,cts} = N_{hear,data}(i) - N_{neighbor}(i) - N_{tx,data}(i). \tag{12}$$

Idle listening also occurs only during the listen period. We compute the idle listening time by subtracting the radio operation time from the total listen period time of the sampling period:

$$\begin{aligned} T_{idle} = t_{listen}\frac{T}{t_f} &- N_{tx,data}(i)(t_{rts} + t_{cts}) \\ &- N_{rx,data}(i)(t_{rts} + t_{cts}) \\ &- T_{overhear} - t_{sync}\frac{T}{t_{sync,interval}} \\ &- \left(N_{tx,route}(i) + N_{rx,route}(i)\right) t_{data}. \end{aligned} \tag{13}$$

Nodes turn on the radio once per each frame, therefore $t_{wakeup}K$ is needed for radio startup in each period T:

$$T_{startup} = t_{wakeup}K. \tag{14}$$

From (2), the total sleep time T_{sleep} during the sampling period is computed by subtracting the time for ratio operations from the sampling period.

4.1.2 *BMAC.* BMAC [Polastre et al. 2004] uses a preamble to implement a low-power listening (LPL) scheme. Nodes sleep and wake up periodically. When a node wakes up, it turns on its radio and checks for activity on the channel. If the node does not detect any activity on the channel, it goes immediately back to sleep. If it detects the activity, the node keeps the radio on and receives the packet. After receiving the packet, the node goes back to sleep. Senders transmit a long preamble before each payload, such that receivers can detect the channel activity and remain active. To reliably receive the data, the preamble length

should be greater than or equal to the interval that the channel is checked for activity.

When it has a packet to transmit, BMAC first sends a long preamble (of length $t_{preamble}$) followed by the payload. For a meaningful comparison to the other protocols, we assume that BMAC enables acknowledgments (which are optional in BMAC):

$$\begin{aligned} T_{tx} &= N_{tx,data}(i)(t_{preamble} + t_{data}) + N_{rx,data}(i)t_{ack} \\ &\quad + N_{tx,route}(i)(t_{preamble} + t_{data}), \end{aligned} \tag{15}$$

Sensor nodes, on average, receive half the entire preamble for every packet destined to them:

$$\begin{aligned} T_{rx} &= N_{rx,data}(i)(0.5t_{preamble} + t_{data}) + N_{tx,data}(i)t_{ack} \\ &\quad + N_{rx,route}(i)(0.5t_{preamble} + t_{data}). \end{aligned} \tag{16}$$

When the packets are destined to other nodes (overhearing), sensor nodes only receive the preamble and the data, but they do not send ACK:

$$\begin{aligned} T_{overhear} &= N_{overhear,data}(i)(0.5t_{preamble} + t_{data}) \\ &= (N_{hear,data}(i) - N_{rx,data}(i))(0.5t_{preamble} + t_{data}). \end{aligned} \tag{17}$$

Unlike SMAC, where the number of frames per sampling interval is constant and therefore the startup time ($T_{startup}$) is also constant for each sampling period (14), the number of frames in BMAC varies depending on the amount of data to send or receive. There are two types of frames in BMAC: the idle frame and the active frame. If there is any packet to receive when node i wakes up for the LPL, or if node i wakes up for a transmission, then the frame is active. The total number of active frames is

$$K_{active} = N_{tx,data}(i) + N_{tx,route}(i) + N_{hear,data}(i) + N_{rx,route}(i). \tag{18}$$

If there are no packets to receive or transmit at LPL time, then the frame is idle. The total number of idle frames is:

$$K_{idle} = \left(\frac{T}{t_f}\right) - K_{active}. \tag{19}$$

The idle listening time in BMAC is dependent on K_{idle} because a node wakes up periodically and keeps radio in reception mode during t_{lpl} to detect incoming packets:

$$T_{idle} = K_{idle}t_{lpl}, \tag{20}$$

where t_{lpl} is the time for low-power listening. In the real BMAC implementation, the radio remains in the receive mode for 8 ms to detect an incoming packet after waking up.

Because a node wakes up in every frame, the total radio startup time during T is

$$T_{startup} = (K_{idle} + K_{active})\, t_{wakeup}. \tag{21}$$

For delay, we only consider the sleeping delay, which is dominant in systems with on/off sleep schedules. In SMAC, each node sends one data packet after the RTS/CTS exchange during a frame (t_f); therefore, the expected delay from node i to the sink is as follows:

$$D_{sleep,smac}(i) = (t_{sleep} + t_{rts} + t_{cts} + t_{data})N_{hop}(i), \tag{22}$$

where $N_{hop}(i)$ is the hop count from node i to the sink.

To make sure a receiver detects an incoming packet during LPL, BMAC sends a long preamble followed by data packets. The expected end-to-end delay of node i over BMAC is

$$D_{sleep,bmac}(i) = (t_{preamble} + t_{data})N_{hop}(i). \tag{23}$$

Assuming low traffic in the network, we consider the throughput to be equal to the offered load [Chiasserini and Garetto 2004]; hence the expected number of bits transmitted by node i during the period T is

$$Throughput(i) = \frac{N_{tx,data}(i) * L_{packet} * 8}{T}, \tag{24}$$

where L_{packet} is the data packet size in bytes.

4.2 Estimation Framework

RaPTEX offers a method of performing an analytical performance estimation based on an object-oriented performance estimation framework that facilitates the implementation and execution of the adopted analytical model. The framework consists of three basic software elements: the engine, the protocol component, and the pipe. First, the engine acts as a controller that manages the flow of the estimation. Second, the protocol component provides templates (base classes) for the implementation of analytical models in each network layer. By deriving a protocol component from the base classes in the framework, a protocol designer can easily integrate the new analytical model in RaPTEX. Because the PXML contains a Java class path for the corresponding analytical model implementation, the user's protocol choices can be seamlessly transformed to run analytical estimations in a single tool. Last, the input and output pipes are defined in each component and connect components in different layers. As shown in Figure 3, the estimation result will be displayed as a graph.

4.3 Estimation Results

We compare the performances of an application with different MAC protocols, BMAC and SMAC, both in a multihop and a single-hop network using the analytical model, the RaPTEX emulator (Section 5), and a real testbed.

In the single-hop scenario, we show the tradeoffs of different energy-saving mechanisms. Five nodes generate a packet every 5 s and every node can hear all the other nodes. The battery capacity for all energy consumption tests is assumed to be 2500mAh. Figure 11 shows the expected lifetime as a function of the sleep time. With a short sleep time, SMAC's short wakeup interval

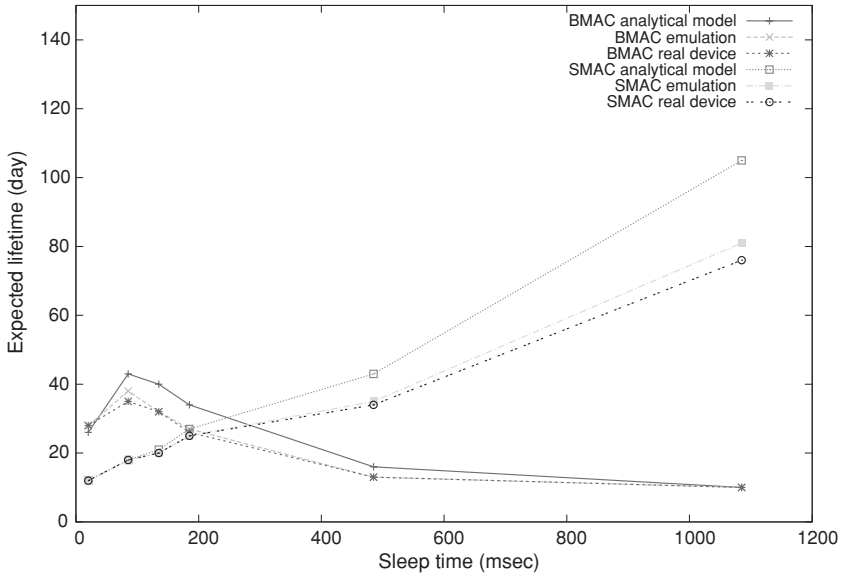

Fig. 11. Expected lifetime (single hop).

reduces the node lifetime due to the frequent idle listening (128 ms), while, as the sleep time increases, the lifetime of BMAC decreases due to the increased preamble size. Although there are small differences between the results from the analytical model and the real testbed, the RaPTEX's emulator provides very accurate results.

In the multihop scenario, we build a 10-hop chain topology to isolate the multihop forwarding effects. Each node generates a packet every 10 s and data packets are forwarded to the sink through DefaultRoute. Every node is configured for 135 ms sleep time for BMAC, and 50% duty cycle (129 ms sleep time) for SMAC. As shown in Figures 12 and Figure 13, the analytical results capture the funneling effect of BMAC; nodes close to sink are exposed to more packets and spend more energy. On the other hand, due to SMAC's fixed idle listening time, all SMAC nodes consume almost the same energy regardless of the traffic load. Figure 14 shows that the end-to-end average delay in the considered WSN is a linear function of the overhead of the sleep time of each MAC over the multihop topology [Polastre et al. 2004].

Although there are differences between RaPTEX's evaluation methodologies and the real testbed in the multihop test, the differences are small and can be explained by the assumption of the absence of collisions and by ignoring contention and retransmissions in the analytical model and the idealized channel model in the emulator, not accounting for the interference in our lab.

5. EMULATION ENVIRONMENT

For quick exploration of the performance tradeoffs of a chosen protocols, RaPTEX provides the estimation environment explained in the previous section,

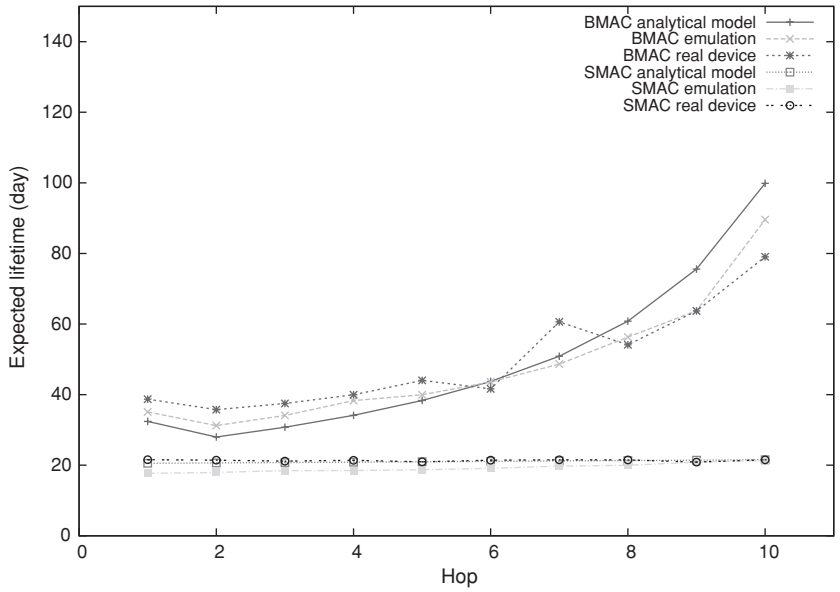

Fig. 12. Expected lifetime (10 hop).

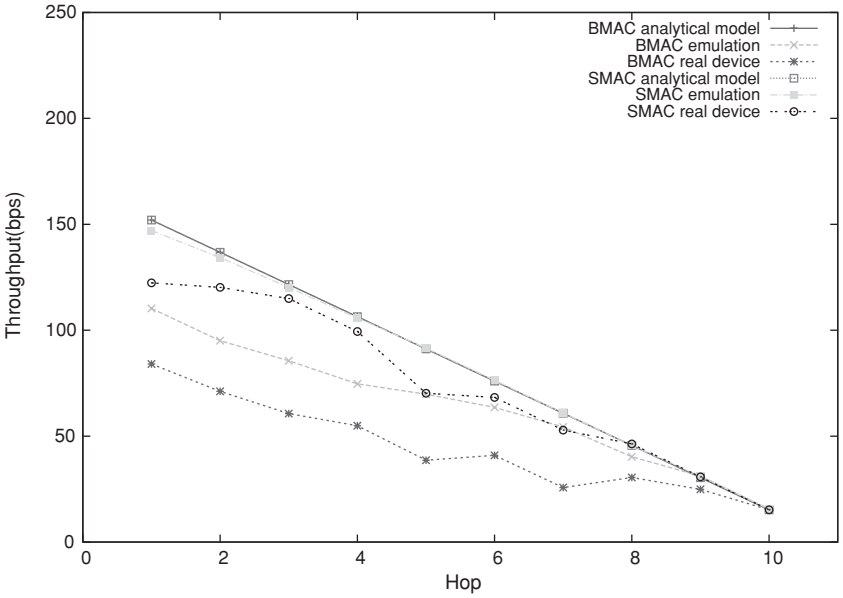

Fig. 13. Throughput (10 hop).

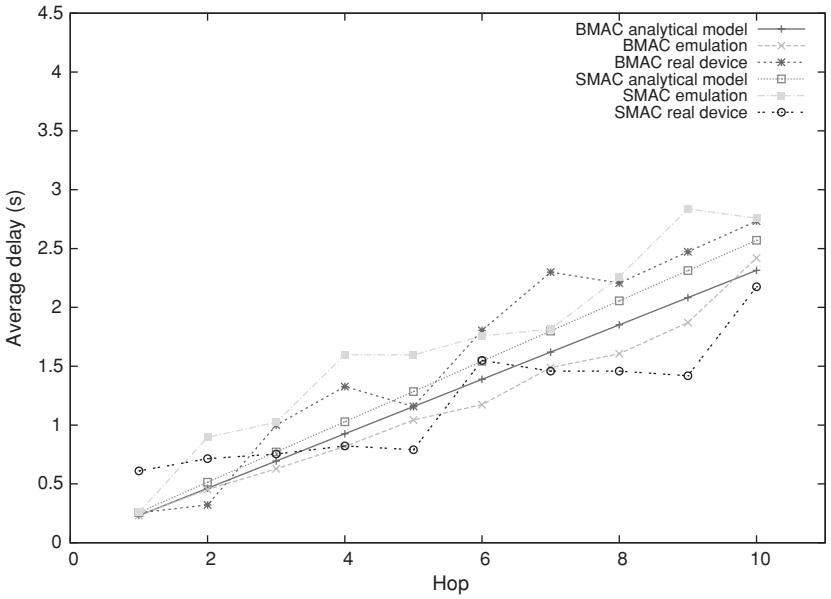

Fig. 14. Average delay (10 hop).

which approximates the projected performance at the time of the composition of the protocol stack. For a more realistic performance evaluation, in this section, we present a hybrid, accurate, yet scalable, emulation environment designed from the following requirements:

—The emulator should run the complete source code either generated from the toolbox, or modified by a user, and the source code can be uploaded to the real device without any modification.

—Run-time properties at every clock cycle during the execution should be accurately captured.

—Network-level properties such as physical layer effects, asymmetric links from different levels of transmission power, hidden terminal effects, and packet collisions should be accurately accounted for.

—High-level system properties such as mobility patterns, heterogenous applications, or deployment strategies should be available.

—The emulation environment should be scalable for large networks in terms of the number of nodes.

5.1 Cycle-Accurate Emulator Over Network Simulator

For performance evaluation of a system design and implementation specific to WSNs, there are many popular simulation and emulation environments available [Levis et al. 2003; Titzer et al. 2005; Polley et al. 2004; Park et al. 2003; Sobeih et al. 2005]. However, none of the previous works studies have satisfied all our design requirements. Our primary approach is to execute the complete

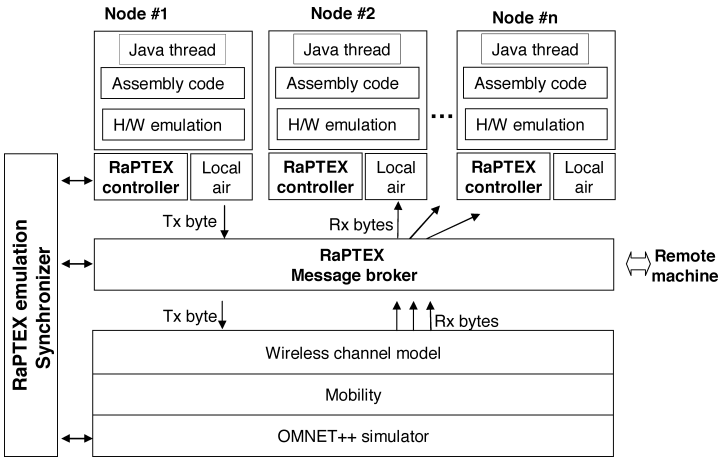

Fig. 15. Software architecture of the RaPTEX emulation environment.

source code (assembly code) of each independent node by emulating the microcontroller with cycle-level accuracy. However, the CPU-oriented nature of node emulation often lacks network-level support such as plug-in channel models, mobility, and topology flexibility and limits scalability, presented in more detail in Section 5.5. To overcome this drawback, we propose a hybrid emulation environment in which we improve the network-level support by integrating the node emulation with a network simulator and solve the scalability problem by supporting distributed emulation.

For accurate execution of real code, we base our emulation on Avrora [Titzer et al. 2005], which emulates the 7.3728-MHz ATMega128L microcontroller and the CC1000 radio at cycle-level accuracy. For supporting network-level effects, we adopt the Mobility Framework (MF) of OMNeT++ [Varga 2001] as a network simulator, which supports node mobility and diverse wireless channel models with the programming framework for extension.

Figure 15 shows the architecture of the RaPTEX emulation environment in which all nodes run as Java threads to execute independent assembly code as if real devices are running on an actual deployment. The internal clocks of each node are synchronized with each other and with the underlying network simulator through the RaPTEX emulation synchronizer. The RaPTEX message broker is responsible for passing all messages to the proper emulation threads either in a local machine or in remote machines for the case of distributed emulation. The RaPTEX message broker also bridges the emulation threads written in Java and the process of OMNeT++ written in C++ through Java Native Interface (JNI).

5.2 Synchronization

Since all node emulation threads execute an independent copy of assembly code with their own internal clocks and event queues, they must be synchronized with each other and with the underlying network simulator. The

RaPTEX emulation synchronizer coordinates individual emulation threads and the underlying network simulator by extending the basic Avrora `Synchronizer` class.

When a node needs to read data bits or sample the RSSI value at some point of emulation time, it must be blocked until all neighbors reach that emulation time to guarantee the reception of all transmissions from the neighbors [Titzer et al. 2005]. However, the number of blocking nodes and their local time should be managed by the synchronizer to prevent deadlock. Every node is also synchronized every byte time (3072 cycles for Mica2) to guarantee byte-level accuracy. Before moving to the next synchronization session, the network simulator's simulation time is also synchronized with the global emulation time to conserve byte-level accuracy of the node emulation, which means that the data interaction and the node mobility is simulated at the byte-time granularity.

5.3 Network-Level Support

To prevent accuracy degradation at the network level, the RaPTEX's basic wireless channel module adopts byte-level rather than packet-level communication. When a node transmits a byte, which is a part of a packet, the RaPTEX message broker passes the data to the selected wireless channel module; the RaPTEX message broker implemented using Varga [2007] is an OMNeT++ module connected to the wireless channel module in the simulation description file (`ned` file in OMNeT++). Because the mobility modules of each node update the current position of the node according to their patterns, the selected wireless channel module dynamically passes the data to all potential receivers within the maximum interference range, based on the sender's transmission power and the current position of the nodes. Then receivers decide whether to take the data from the calculation of the received signal strength. If receivable, the RaPTEX message broker writes the data to the radio buffer of the proper emulation threads either in a local machine or in remote machines for the case of distributed emulation.

The bits received during the transmission collision period are bit-ORed at the receiver's buffer. Thus, the hidden terminal effect and packet collisions are accurately supported by spatial and temporal synchronization between the emulation threads and the network simulator.

The basic channel module is simple, but contains all the details for extensions such as clock-level timing information for byte data interaction, transmission power, and node location. By extending the basic channel module, developers can implement more realistic channel models or plug in the existing models shared in the OMNeT++ community.

5.4 Mobility Support

RaPTEX's mobility architecture is based on the Mobility Framework of OMNeT++. There are two core components in the mobility architecture. First, the global `ChannelControl` module manages the location of nodes and the connections between links. Second, the `MobilityModule` provides a way to describe

the mobility model of every individual node and communicates the location changes of the nodes to the `ChannelControl`, which updates all connections according to the location changes. To define a new mobility pattern, one has to derive the mobility module from the `BasicMobility`, which provides basic `MobilityModule` functionalities, writes a specific mobility pattern by overriding `makeMove()`, and assigns the new mobility module to nodes that need to follow the mobility pattern.

5.5 Distributed Emulation

Avrora's computation-oriented nature enables accurate program emulation; however, the accuracy comes at the cost of scalability. Avrora's multithreaded structure only allows it to be executed in a single machine. To support large networks, we distribute the emulation threads over several host machines, because, as the number of node increases, the emulation performance is limited by the cost of executing each emulation thread rather than the synchronization overhead.

In each emulation host, a subset of the emulation threads is executed. Like the single-host emulation, each emulation thread executes its own assembly code and sends/receives data via the RaPTEX message broker, and the local RaPTEX emulation synchronizer ensures that each emulation thread is synchronized with other threads in the same emulation host. For host-to-host synchronization, each local synchronizer communicates with the RaPTEX distributed synchronizer running in a coordination server. Each host runs an independent network simulator with identical configurations such as position information for all nodes in the network under the same channel model. Whenever data is injected from an emulation thread for transmissions, the corresponding receiver nodes are identified by the channel model in the local host, and the RaPTEX message broker relays the data either to an emulation thread in the local host or to remote hosts that execute the intended receiver threads. In addition to data interactions, when a node updates its location or changes its transmission power, the RaPTEX message broker multicasts control messages to all remote hosts to ensure that all hosts maintain the same snapshot of the network topology.

While we use a peer-to-peer design for data interaction, a central coordinator is used for the host-to-host synchronization. Whenever host emulation status (e.g. all nodes are waiting for neighbors' clock update for reception or all node are waiting for next synchronization session) changes, each local synchronizer sends control messages via the message broker to the distributed synchronizer running in the coordination server. Then the distributed synchronizer gathers the status of each host and coordinates each emulation host under the same synchronization rules described in Section 5.2.

In the distributed emulation, the major source of overhead is network traffic between remote hosts. Therefore, the way of partitioning nodes into emulation hosts can directly influence the emulation performance. In general, the goal of partitioning in graph theory is twofold: equal number of vertices in subdomains and lowest number of edges between subdomains. Our goal is also exactly the

```
RaptexEmulNet.host[0].url = "152.14.97.44";
RaptexEmulNet.host[1].url = "152.14.96.44";
RaptexEmulNet.host[3].url = "152.14.96.45";
RaptexEmulNet.host[4].url = "152.14.96.45";
```

Fig. 16. The code snippet for the distributed emulation in the emulation configuration file (omnet.ini).

the same as for the general case, because we need to evenly distribute the number of nodes (vertices) among emulation hosts (subdomains) to prevent any host from being the bottleneck of the whole emulation. We also need to minimize the number of links (edges) between nodes running in different hosts to reduce network traffic.

The optimal graph partitioning is known to be an NP-complete problem in general, and many heuristic approaches have been proposed, including geometric techniques, combinational techniques, spectral methods, and multilevel schemes. Many open-source software packages for graph partitioning are also available Chaco [Hendrickson and Leland 1995], Jostle [Walshaw and Cross 2007], ParMetis [George Karypis and Kumar 2003], etc. In our distributed emulation, we use Chaco, which is the open source and provides all above-mentioned partitioning schemes. From the RaPTEX's topology diagram, Chaco's input graph file is created and the output file is used to assign a set of emulation threads to each emulation host, which is described in `omnet.ini` file as shown in Figure 16.

5.6 Scalability

We performed the distributed emulation and compared the scalability of RaPTEX emulation environment with that of pure Avrora. Each emulation host was equipped with a Pentium Core2Duo 2.13 GHz and 2048 MB of RAM, and connected to a 100-Mbps Ethernet LAN. We randomly deployed nodes and evenly partitioned the topology into 2, 5, 10, and 20 machines, respectively. Every node ran a simple application, which sent a packet every 1 s for 5 s over BMAC (always-on mode). The average number of neighbors was 10 for the all experiments. The experiment result is shown in Figure 17.

Due to the overhead of data delivery through the wireless channel model, RaPTEX in a single machine is slightly slower than pure Avrora for any number of nodes. Due to the network communication cost for data interactions between remote hosts and host-to-host synchronization, there is also no benefit from distributed emulation when the number of nodes is small.

However, as the number of nodes increased, the cost of running the emulation threads surpassesed the communication overhead. With two machines, for 400 nodes, the RaPTEX emulation was 1.7 times faster than pure Avrora. Moreover, due to the single-machine-running restriction of Avrora, the maximum number of nodes that could be emulated in Avrora was limited to 400 nodes in our experiment, while RaPTEX can emulate a larger number of nodes, depending on the number of hosts available.

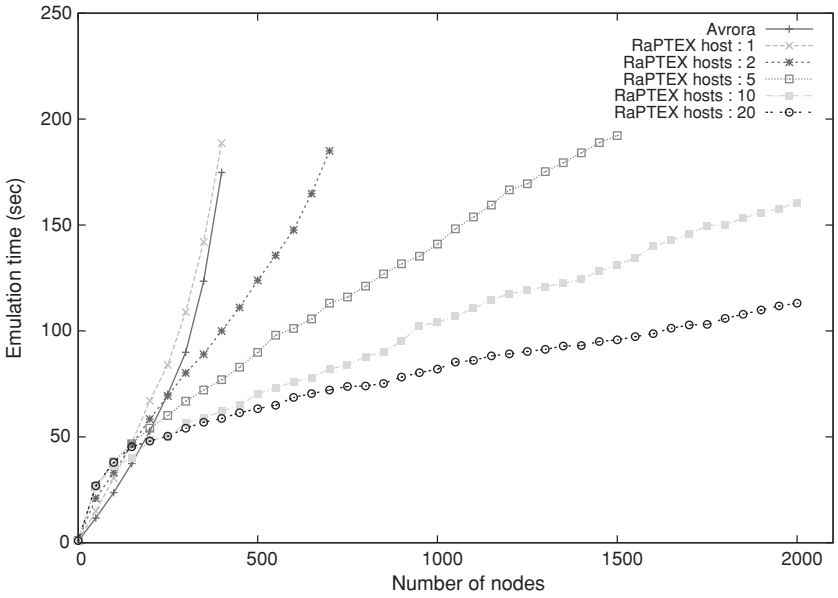

Fig. 17. Emulation speed as a function of the number of nodes, for different number of hosts running the RaPTEX emulator.

6. USE CASE

In this section, we present three use cases. First, we explain the sensor application developmental process using RaPTEX under the given application requirements. Second, we explore the tradeoff spaces of different protocol stacks and configurations in terms of energy consumption, delay, and packet delivery ratio. Finally, we explore the effect of different topologies in a large network. All the protocol choices and configurations as well as evaluations and analyses are performed in RaPTEX.

6.1 Use Case 1: Application Developmental Process

In this subsection, we develop a continuous monitoring application in WSNs, which periodically collects sensor data every 1 min and reports the reading to a sink node via a multihop routing protocol. An important goal in this class of application is to maximize the lifetime of the network because the sensor nodes operate on limited battery power in an unattended environment, and, often, longer delays can be tolerated. Therefore, in this use case, we require that the lifetime of the bottleneck node should be longer than 3 months, and the per-hop delay should be smaller than 1 s. For the simplicity of the presentation, we deploy this application in a 10-hp chain topology and use MintRouting as a routing protocol.

To find the best combination of protocols and protocol configurations, we first compose a protocol stack using SMAC, MintRoute, and the template monitoring application in the protocol diagram, and build the 10-hop chain topology in the

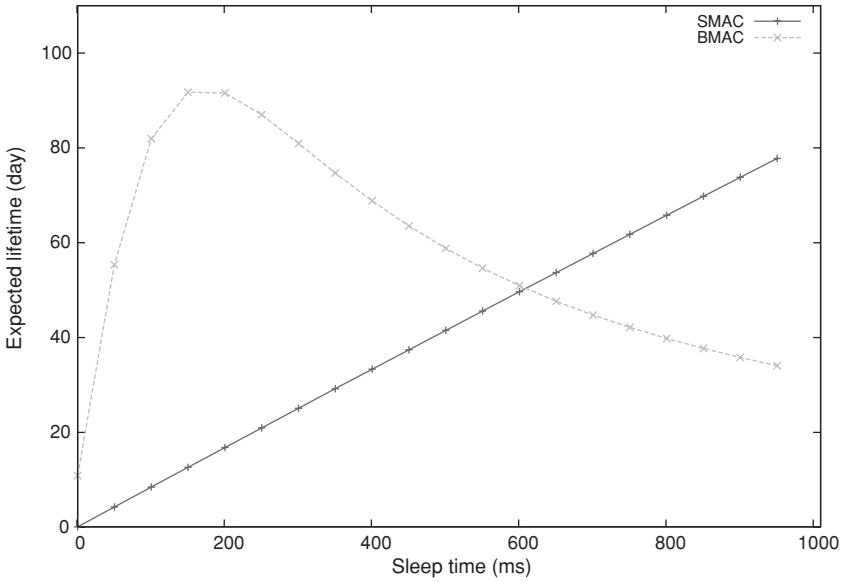

Fig. 18. Analytical estimation result; lifetime as a function of sleep interval at the bottleneck node.

network topology diagram. Since, in this use case, we do not intend to change the internal logic of the protocols, the nesC diagram is not used. Because the lifetime of nodes and per-hop delay is mainly affected by the sleep time of the chosen MAC, we set the SMAC sleep time as a range from 10 ms to 1000 ms. The data packet generation interval is set to 1 min. The number of packets a node overhears, receives, and forwards in the network is decided depending on the location, transmission power, and the path loss model adopted in the topology diagram.

To quickly explore the performance tradeoffs involved with the protocol stack and the configurations, we run an analytical estimation at the bottleneck node (2 hops away from the sink). Since formulae for theoretical analysis are automatically composed by the estimation framework, no extra configuration is required. RaPTEX displays the analytical estimation result as a graph. Figure 18 shows that with SMAC we cannot achieve the required lifetime (90 days) for any sleep time in the given range. Therefore, we change the MAC protocol to BMAC in the protocol diagram, and again set the sleep time as a range from 10 ms to 1000 ms. Figure 18 shows that BMAC provides its maximum lifetime at sleep time 200 ms, which is lower than the per-hop delay constraint, and the lifetime meets the application requirement. From the analytical estimation result, we tentatively decide to choose BMAC in the MAC layer with a sleep time of 200 ms.

To verify the decision more accurately, we execute the prototype application in the RaPTEX emulation environment. Before executing the application, RaPTEX composes the protocol source code and generates a Makefile as shown in Figure 19. This Makefile includes user's tunable parameters and all the

```
#Top level Makefile generated by the RaPTEX code generator
RAPTEX=true
COMPONENT=Sample_Monitoring_APP

# LAYER
APP_LAYER="Monitoring"
ROUTE_LAYER="MintRoute"
MAC_LAYER="BMAC"
PHY_LAYER="BMAC-Physical"

DOWNWARD_CMD="YES"
TX_QUEUE="YES"

#Debug ---------------------------------
CFLAGS+= -DRAPTEX_DEBUG_ENABLE
CFLAGS+= -DRAPTEX_ENABLE_ENERGY_MEASURE

#App ----------------------------------
CFLAGS += -DAPP_SAMPLE_PERIOD=61440.0

#ROUTE ---------------------------------
CFLAGS += -DDATA_FREQ=APP_SAMPLE_PERIOD
CFLAGS += -DDATA_TO_ROUTE_RATIO=2
CFLAGS += -DHOP_CNT=10
CFLAGS += -DMHOP_QUEUE_SIZE=16

#MAC ----------------------------------
CFLAGS += -DBMAC_ENABLE_ACK
CFLAGS += -DBMAC_WAKEUP_INTERVAL=200

#PHY ----------------------------------
CFLAGS += -DCC1K_DEFAULT_FREQ=CC1K_914_077_MHZ
CFLAGS += -DTX_POWER=15

include ../MakeRaptex
```

Fig. 19. The code snippet of a top level Makefile.

selected protocols, which follow the interface structure shown in Figure 6. The path for the real source code is decided in `MakeRaptex` using compile option (`-I`), which is generated based on the NXML and PXML from the protocol diagram or directly from the nesC diagram.

The source code is then compiled to assembly code to be injected into each of the emulation threads. For emulation-specific configurations such as emulation time and monitoring options, RaPTEX provides emulation configuration windows as shown in Figure 2. We run the emulation for 5000 s and the start time of each emulation thread is randomly chosen within the first minute of emulation. During the emulation, we trace all the packets sent and received at the clock cycle level granularity and energy consumption for all nodes. Using the emulation postprocessing tool, we import the emulation trace information into the database and analyze the emulation result. The result shows our choice of protocols and configurations satisfies the given application requirements. In Figure 20, the lifetime of all nodes in the network is over 90 days. Figure 21

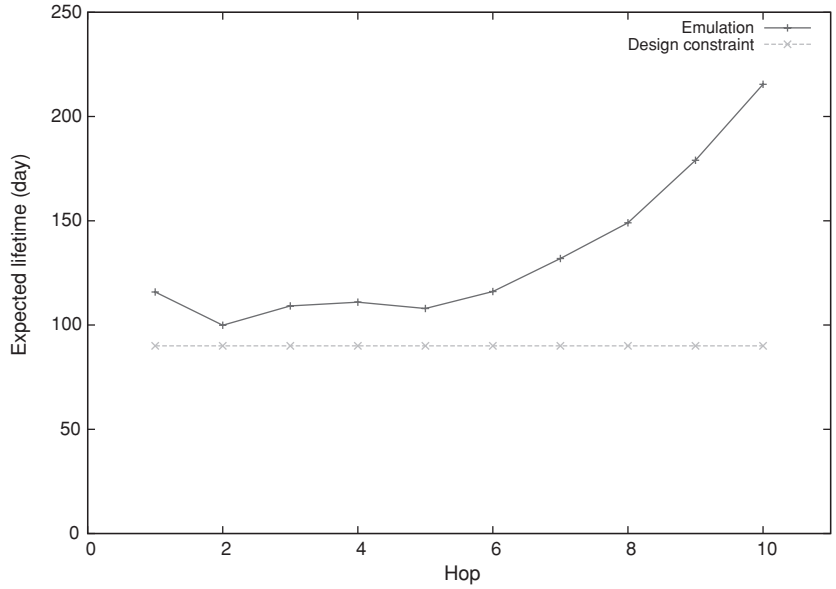

Fig. 20. Emulation result for energy; lifetime as a function of the number of hops from the sink, when BMAC is used.

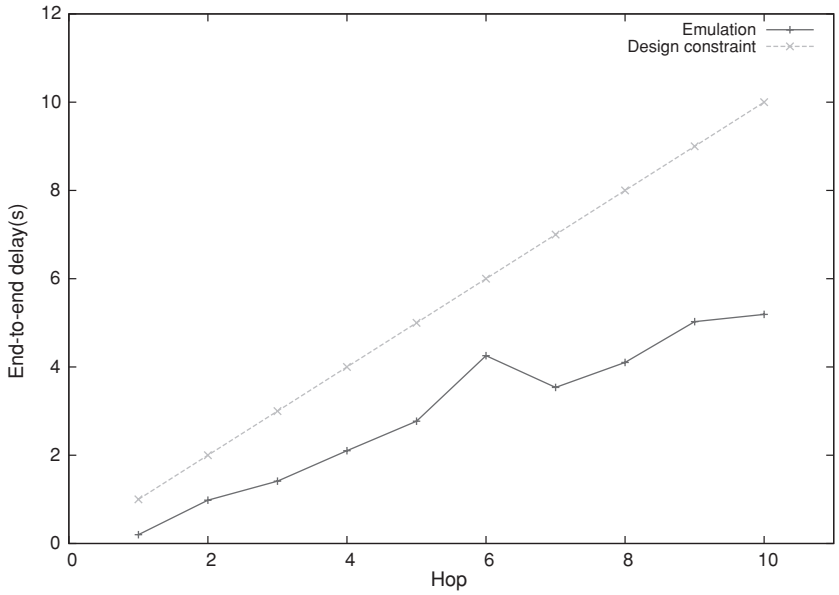

Fig. 21. Emulation result for delay; end-to-end delay as a function of the number of hops from the sink, when BMAC is used.

shows the end-to-end delay as a function of the number of hops from the sink, and it is clear that the per-hop delay is under 1 s. The composed prototyping source code can be directly uploaded to the real device without any modifications.

Table III. Test Cases for the First Use Case

Data	MAC	Routing	Sleep	Symbol
10 s	BMAC	Default	20 ms	HBDS
	BMAC	Default	135 ms	HBDL
	BMAC	Mint	20 ms	HBMS
	BMAC	Mint	135 ms	HBML
10 s	SMAC	Default	15 ms	HSDS
	SMAC	Default	129 ms	HSDL
	SMAC	Mint	15 ms	HSMS
	SMAC	Mint	129 ms	HSML
60 s	BMAC	Default	20 ms	LBDS
	BMAC	Default	135 ms	LBDL
	BMAC	Mint	20 ms	LBMS
	BMAC	Mint	135 ms	LBML
60 s	SMAC	Default	15 ms	LSDS
	SMAC	Default	129 ms	LSDL
	SMAC	Mint	15 ms	LSMS
	SMAC	Mint	129 ms	LSML

6.2 Use Case 2: Tradeoffs of Different Protocols and Configurations

Although numerous communication protocols and performance analysis tools have been proposed in the WSN area [Malesci and Madden 2006], it is still a nontrivial task to find the best combination of protocols and the configurations for different application purposes due to the lack of data available about the interaction between different protocols in different layers and configurable parameters. The main purpose of this use case is to show tradeoffs of different protocol stacks and configurations. Malesci and Madden [2006] shared the same motivation and goal as this use case, but the article evaluated performances only in terms of end-to-end throughput, and was based on the real measurement, which is usually most accurate and realistic, but difficult to set up and reproduce. We explain the sources of overhead of each protocol and explore performance tradeoffs in terms of delay, packet delivery ratio, and energy consumption using the RaPTEX emulation environment, which is easy to set up, yet accurate.

Because there is very wide range of configuration choices, we limited our evaluation for 16 test cases to four factors: (1) data packet generation intervals (**H**igh traffic and **L**ow traffic), (2) MAC protocols (**B**MAC and **S**MAC), (3) routing protocols (**M**int and **D**efault), and (4) sleep time of the MAC protocols (**S**hort and **L**ong intervals). Table III shows the configurations and symbols for each test. To isolate the multihop forwarding effects, we built a 10-hop chain topology for each test using the network topology diagram (Figure 2). After generating source code for each test, we executed the code in the RaPTEX's emulation environment.

Figure 22 shows emulation results for each test case in terms of lifetime, delay, and packet arrival rate. Each performance metric is normalized by the maximum value in that metric. BMAC provides a high lifetime when data traffic is low and sleep time is long (LBDL and LBML) for two main reasons. First, BMAC saves energy by performing the low power listening (LPL) at the

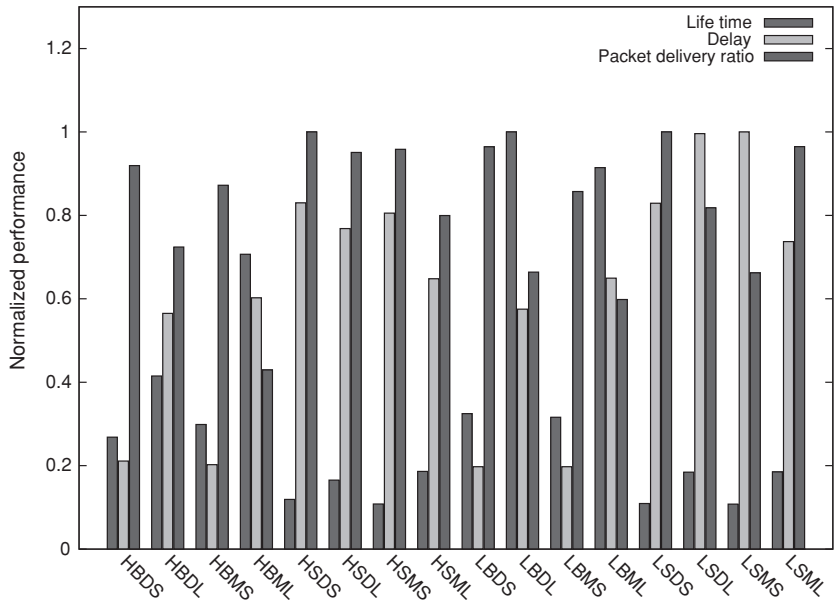

Fig. 22. Performance trade-offs for use case 2; each performance metric is normalized by its maximum value.

expense of long preambles. At a low data generation rate with long sleep time, BMAC's main overhead of sending long preambles is alleviated because of the low data rate, while checking channel activity using LPL is still inexpensive. Therefore, when we use BMAC, the sleep time should be carefully decided based on the expected amount of traffic. Second, due to the absence of retransmissions in BMAC, the arrival rates of LBDL and LBML are relatively low, which means that packets are dropped in the middle of multihop forwarding so that the number of packets to forward decreases. Therefore, if high reliability is required for an application, BMAC's ACK mode should be enabled and BMAC should work with an upper-layer protocol which has a reliability mechanism because BMAC itself does not handle the failure of receiving ACKs.

On the other hand, for both low and high data traffic, SMAC with short sleep time (15 ms) and DefaultRoute combinations (HSDS and LSDS) provides the best sink goodput because SMAC by default performs retransmission for both RTS and data packets. However, the lifetime of SMAC is relatively low mainly due to SMAC's synchronization mechanism and reliability mechanism. The synchronization mechanism periodically keeps the radio device in the receive mode for a fixed listen time (128 ms in Mica2), which is the sum of the required time to receive a SYNCH packet and a data packet. This fixed listen time is the reason why SMAC's energy consumption is not significantly affected by the amount of traffic.

Figure 22 shows that it is not easy to find a single protocol stack and set of parameters, which performs best in every situation for all the performance metrics. Thus, depending on the constraints of a system, the designer may favor one or the other of the choices of protocols and parameters (or continue

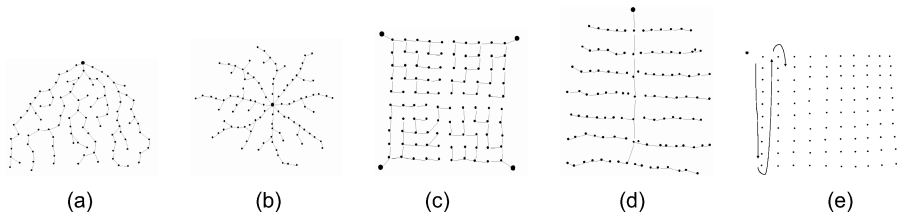

Fig. 23. The five topologies used for use case B: (a) tree, (b) star, (c) grid with four sinks, (d) backbone, and (e) mobile sink.

and refine the search if none of the choices is appropriate). We believe this development and validation process can be facilitated with RaPTEX.

6.3 Use Case 3: Different Topologies

In this third use case, we use the RaPTEX emulation environment to explore the effect of different topologies for a 100-node network, with a continuous monitoring application, which generates a data packet every 60 s.

For simplicity, we use the same network stack in all topologies, using DefaultRoute and BMAC (135-ms sleep time). Nodes are deployed in five different types of network topologies: a tree, star, grid with 4 sinks, relay backbone, and mobile sink (as shown in Figure 23).

The single sink tree (a) and star topologies (b) are typical for many WSN applications. The grid topology (c) has four sinks, one at each corner of the grid. In the relay backbone topology (d), nodes with a higher transmission range form a backbone to the sink through the deployment field. The other nodes are connected (via multiple hops) to the nearest backbone node. By using RaPTEX's mobility capability, the last topology (e) employs a mobile sink that gathers data from the deployed nodes. While moving, the mobile sink informs nodes in transmission range about its presence by sending packets with long preambles, and it never turn its radio off because we assume that batteries of the mobile sink can be replaced. The sensor nodes, on the other hand, sleep for 135 ms at a time, and send packets with minimum preamble size (8 bytes) directly to the sink (when the sink is in their range).

Figure 24 shows the CDF of the packet delivery ratio for all nodes in each network. The network with the mobile sink clearly shows the best delivery ratio because nodes send short packets directly to the mobile sink only when the presence of sink is detected: 5% of nodes show 98% packet delivery ratio and 95% of nodes achieve 100% delivery ratio. In the tree topology, none of nodes can achieve more than a 65% delivery ratio due to the longest hop toward the sink and high congestion near the sink. By placing the sink in the center of the network, the star topology provides a slightly better delivery ratio than that of the tree topology: 30% of nodes can achieve more than 65% delivery ratio, but still none of nodes show more than a 80% delivery ratio. In the grid network with four sinks, some nodes near the sinks can achieve more than 80% of the delivery ratio, but many nodes still drop packets due to the high

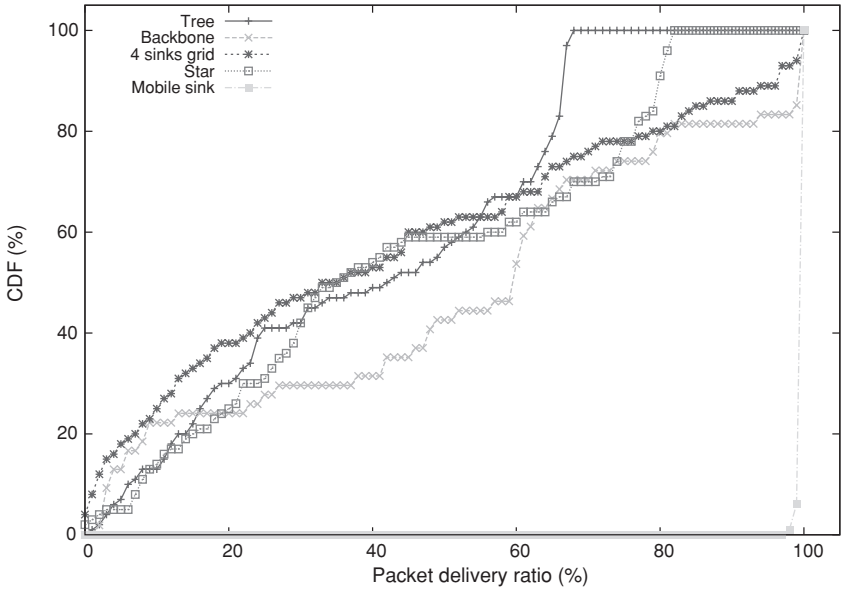

Fig. 24. CDF of the delivery ratio in different network topologies for use case 3.

average node density of the grid leading to contention and a high collision rate. In the backbone topology, backbone nodes avoiding the sleep mode increase the delivery ratio. Therefore, the proximity to the backbone influences the delivery ratio of regular nodes resulting in a stair shape CDF.

Figure 25 shows the distribution of the delays. The delay of the tree topology increases approximately linearly as a function of the distance to the sink. The three extra sinks in grid topology reduce the number of hops in the network, resulting in a smaller delay distribution, similar to the star topology. The delay of the mobile sink WSN is highly dependent on the number of mobile sinks and their speed, and is, therefore, not shown in this graph. For the backbone topology, if a node is close to the backbone, it will experience a shorter delay although it may be physically far from the sink.

In Figure 26, the network with mobile sink shows the best lifetime as the sensor nodes are not required to forward multihop packets and sleep without a long preamble transmission. Due to the forwarding costs, the tree topology indicates the worst life, especially for nodes close to the sink (which have to forward data on behalf of all other nodes in the network). In the backbone network, nodes in range of the backbone are exposed to overhear backbone traffic and nodes outside the range of the backbone only forward their own packets (or for very few other nodes), leading to nodes at both extremes in the lifetime spectrum.

Given the design constraints of a real system, the RaPTEX user can make an informed decision on the best network topology for achieving a target system performance.

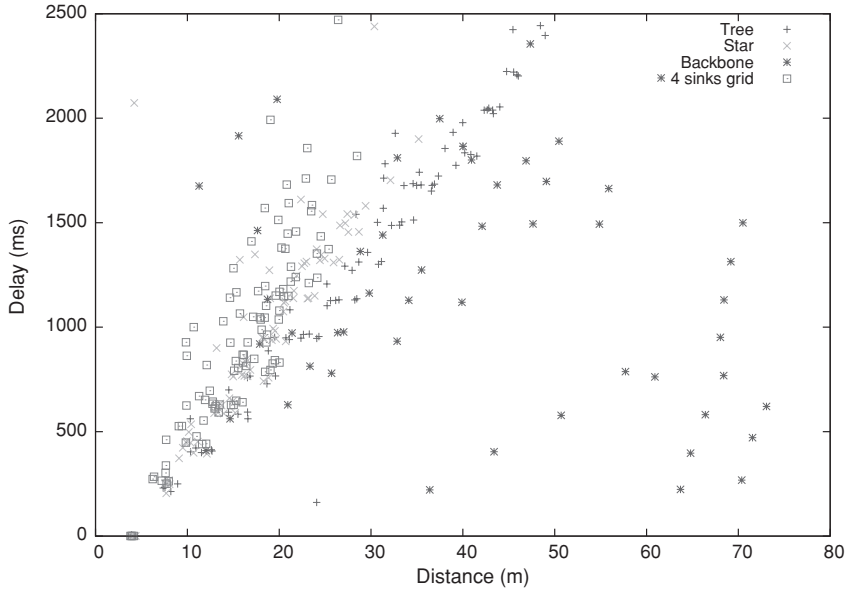

Fig. 25. Distribution of delays as a function of distance to sink for use case 3.

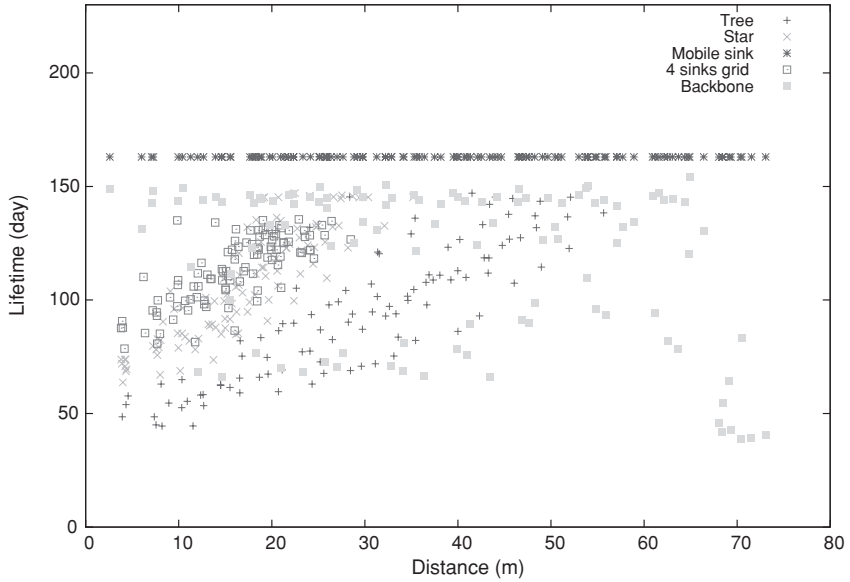

Fig. 26. Distribution of lifetimes as a function of distance to sink for use case 3.

7. CONCLUSION

In this article, we present RaPTEX, a rapid prototyping tool for embedded communication systems, while focusing on applications for WSNs. We designed and developed a toolbox, an analytical model, and an emulation environment to allow nonnetwork specialists to easily customize networking stacks and

to quickly and accurately evaluate their performance. We believe many researchers in unrelated fields, but in need of communication protocols, will benefit from the three subsystems of RaPTEX. Even researchers in the field will benefit from the accuracy and scalability of RaPTEX for performance evaluation. Potential future work will be to support more hardware and software platforms and protocols. The current version of RaPTEX is available at `http://www.ece.ncsu.edu/wireless/MadeInWALAN/Raptex`.

REFERENCES

BALDWIN, P., KOHLI, S., AND LEE, E. A. 2004. Modeling of sensor nets in Ptolemy II. In *Proceedings of IPSN*.

BHATTI, S., CARLSON, J., DAI, H., DENG, J., ROSE, J., SHETH, A., SHUCKER, B., GRUENWALD, C., TORGERSON, A., AND HAN, R. 2005. MANTIS OS: MultimodAl NeTworks of In-situ Sensors. `http://mantis.cs.colorado.edu/index.php/tiki-index.php`.

BUETTNER, M., YEE, G. V., ANDERSON, E., AND HAN, R. 2006. X-MAC: A short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proceedings of SenSys*.

BUONADONNA, P. 2003. tinyos-1.x/lib/route. `http://tinyos.cvs.sourceforge.net/viewvc/tinyos/tinyos-1.x/tos/lib/Route`.

CHEONG, E., LEE, E. A., AND ZHAO, Y. 2005. A graphical development and simulation environment for TinyOS-based wireless sensor networks. In *Proceedings of SenSys*.

CHIASSERINI, C. F. AND GARETTO, M. 2004. Modeling the performance of wireless sensor networks. In *Proceedings of INFOCOM*.

DULMAN, S., KAYA, O. S., AND KOPRINKOV, G. 2005. NesCT. `http://nesct.sourceforge.net`.

FRABOULET, A., CHELIUS, G., AND FLEURY, E. 2007. Worldsens: Development and prototyping tools for application specific wireless sensors networks. In *Proceedings of IPSN*.

GEORGE KARYPIS, K. S. AND KUMAR, V. 2003. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Tech. rep. University of Minnesota, Minneapolis, MN.

GREENSTEIN, B., KOHLER, E., AND ESTRIN, D. 2004. A sensor network application construction kit (SNACK). In *Proceedings of Sensys*.

HENDRICKSON, B. AND LELAND, R. 1995. The Chaco User's Guide 2.0. Tech. rep. SAND95-2344. Sandia National Laboratories, Albuquerque, NM.

JAIN, R. 1991. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, NY.

KWON, Y. AND AGHA, G. 2006. Scalable modeling and performance evaluation of wireless sensor networks. In *Proceedings of the IEEE Real Time Technology and Application Symposium*.

LEVIS, P., N. LEE, M. W., AND CULLER, D. 2003. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of SenSys*.

MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. 2003. The design of an acquisitional query processor for sensor networks. In *Proceedings of SIGMOD*.

MALESCI, U. AND MADDEN, S. 2006. A measurement-based analysis of the interaction between network layers in TinyOS. In *Proceedings of EWSN*.

MOZUMDAR, M. M. R., GREGORETTI, F., LAVAGNO, L., VANZAGO, L., AND OLIVIERI, S. 2008. A framework for modeling, simulation and automatic code generation of sensor network applications. In *Proceedings of SECON*.

NICTA. 2009. Castalia. `http://castalia.npc.nicta.com.au/`.

PARK, S., SAVVIDES, A., AND SRIVASTAVA, M. B. 2003. SensorSim: A simulation framework for sensor networks. In *Proceedings of PLDI*.

PARSONS, G., PENG, S., AND DEAN, A. G. 2008. An ultrasonic communication system for biotelemetry in extremely shallow waters. In *WUWNet, in Conjunction with ACM MobiCom 2008*.

PENG, S., PARSONS, G., AND DEAN, A. G. 2010. RaPTEX: A resource-focused toolchain for rapid prototyping of embedded communication systems. In *INTERACT-14*.

POLASTRE, J., HILL, J., AND CULLER, D. 2004. Versatile low power media access for wireless sensor networks. In *Proceedings of SenSys*.

POLLEY, J., BLAZAKIS, D., MCGEE, J., RUSK, D., AND BARAS, J. S. 2004. ATEMU: A fine-grained sensor network simulator. In *Proceedings of SECON*.

RICE UNIVERSITY. 1999. Wireless and Mobility Extensions to ns-2. `http://www.monarch.cs.rice.edu/`.

SALLAI, J., BALOGH, G., AND DORA, S. 2005. TinyDT: TinyOS plug-in for the eclipse platform. `http://tinydt.sourceforge.net`.

SHNAYDER, V., HEMPSTEAD, M., RONG CHEN, B., ALLEN, G. W., AND WELSH, M. 2004. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of SenSys*.

SOBEIH, A., CHEN, W.-P., HOU, J. C., KUNG, L.-C., LI, N., LIM, H., TYAN, H.-Y., AND ZHANG, H. 2005. J-Sim: A simulation environment for wireless sensor networks. In *Proceedings of the 38th Annual Symposium on Simulation*.

THE MATHWORKS, INC. 1994. MATLAB and Simulink for technical computing. `http://www.mathworks.com`.

TITZER, B. L., LEE, D. K., AND PALSBERG, J. 2005. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of IPSN*.

VAN DAM, T. AND LANGENDOEN, K. 2003. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of Sensys*.

VARGA, A. 2001. The OMNeT++ discrete event simulation system. In *Proceedings of ESM*.

VARGA, A. 2007. Jsimplemodule. `http://www.omnetpp.org/pmwiki/index.php?n=Main.JSimpleModule`.

VARSHNEY, M., XU, D., SRIVASTAVA, M., AND BAGRODIA, R. 2007. sQualNet: An accurate and scalable evaluation framework for sensor networks. In *Proceedings of IPSN*.

VOLGYESI, P. AND LEDECZI, A. 2002. Component-based development of networked embedded applications. In *Proceedings of the 28th Euromicro Conference, Component-Based Software Engineering Track*.

WALSHAW, C. AND CROSS, M. 2007. JOSTLE: Parallel multilevel graph-partitioning software—an overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Civil-Comp. Ltd., Kuppen Stirling, U.K.

WEISER, M. 1991. The Computer for the 21st Century. *Sci. Amer.* Sept., 94–100.

WOO, A., TONG, T., AND CULLER, D. 2003. Taming the underlying chanllenges of reliable multihop routing in sensor networks. In *Proceedings of SenSys*.

YE, W., HEIDEMANN, J., AND ESTRIN, D. 2004. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE/ACM Trans. Netw. 12*, 3 (June), 493–506.

YE, W., SILVA, F., AND HEIDEMANN, J. 2006. Ultra-low duty cycle MAC with scheduled channel polling. In *Proceedings of SenSys*.

YOON, S. 2007. Power management in wireless sensor networks. Ph.D. dissertation, North Carolina State University, Raleigh, NC.

Received July 2009; revised October 2009; accepted December 2009