J-Sim: A Simulation and Emulation Environment for Wireless Sensor Networks

Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang

Abstract

Wireless Sensor Networks (WSNs) have gained considerable attention in the past few years. They have found application domains in battlefield communication, homeland security, pollution sensing and traffic monitoring. As such, there has been an increasing need for defining and developing simulation frameworks for carrying out high-fidelity WSN simulation. In this paper, we present a modeling, simulation, and emulation framework for WSNs in *J-Sim* — an open-source, component-based compositional network simulation environment that is developed entirely in Java. This framework is built upon the autonomous component architecture (ACA) and the extensible internetworking framework (INET) of J-Sim, and provides an object-oriented definition of (i) target, sensor and sink nodes, (ii) sensor and wireless communication channels, and (iii) physical media such as seismic channels, mobility model and power model (both energy-producing and energy-consuming components). Application-specific models can be defined by sub-classing classes in the simulation framework and customizing their behaviors. We also include in *J-Sim* a set of classes and mechanisms to realize network emulation.

We demonstrate the use of the proposed WSN simulation framework by implementing several well-known localization, geographic routing, and directed diffusion protocols, and perform performance comparisons (in terms of execution time incurred, and the memory used) in simulating several typical WSN scenarios in *J-Sim* and *ns-2*. The simulation study indicates the proposed WSN framework in *J-Sim* is much more scalable than *ns-2* (especially in memory usage). We also demonstrate the use of the WSN framework in carrying out real-life, full-fledged future combat system simulation and emulation.

Index Terms

Sensor Networks, Wireless Networks, Network Simulation, Network Emulation, Modeling, J-Sim, Future Combat Systems (FCS)

I. Introduction

Recent technological advances have led to the emergence of pervasive networks of small, low-power devices that integrate sensors and actuators with limited on-board processing and wireless communication capabilities. These sensor networks open new vistas for many potential applications, such as battlefield surveillance, environment monitoring, traffic monitoring, and biological detection [28], [38], [18]. Sensor networks have also been proposed in homeland security to monitor wide open spaces and provide real-time detection of attacks in the forms of chemical, biological or radiological weapons of mass destruction.

A major requirement of wireless sensor networks (WSNs) is for the sensors to reliably disseminate information within a time interval that allows the controller to take necessary action, even in the case of poor spatial distribution of sensor devices, wireless/acoustic interference, and malicious destruction. Out-of-date information is of no use, as the object/event that was tracked may no longer be in the vicinity when the information is received. This presents a key technical challenge in cooperative engagement — how to effectively coordinate and control sensors over an unreliable wireless ad hoc network. In particular, due to the unique characteristics of data-centric sensor networks, many new design issues arise and protocols originally designed for wireline and/or generic ad hoc networks have to be adapted or entirely re-designed.

In order to enable design and development of new protocols and applications for wireless sensor networks and evaluation of their performance, several simulation environments have been extended to include simulation frameworks for wireless sensor networks. In this paper, we report our experiences in building a simulation and emulation framework for WSNs in

Contact Author: Ahmed Sobeih is with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. E-mail: sobeih@uiuc.edu

Wei-Peng Chen is with the IP Networking Research Department, Fujitsu Labs. of America, Inc. E-mail: wchen@fla.fujitsu.com.

Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, and Honghai Zhang are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. E-mail: {jhou,kung,nli,hyuklim,hzhang3}@uiuc.edu.

Hung-Ying Tyan is with the Department of Electrical Engineering, National Sun Yat-Sen University, Taiwan. E-mail: tyanh@mail.ee.nsysu.edu.tw. Except for the first author, co-authors are listed in the alphabetical order of their last names.

J-Sim [9] — an open-source, component-based compositional network simulation environment that is developed entirely in Java.

J-Sim is implemented on top of a component-based software architecture, called the *autonomous component architecture (ACA)*. The basic entities in the ACA are *components*, which communicate with one another via sending/receiving data at their *ports*. How components behave (in terms of how a component handles and responds to data that arrive at a port) is specified at system design time in *contracts*, but their binding does not take place until the system integration time when the system is being "composed." With the separation of contract binding (at system design time) from component binding (at system integration time), *J-Sim* provides a loosely-coupled component architecture, i.e., a component can be individually designed, implemented and tested independently [9], [58]. By closing the gap between hardware and software ICs, the ACA enables new components to be included into *J-Sim* in a plug-and-play fashion. On top of the ACA, a generalized packet-switched internetworking framework (called INET) has been laid based on common features extracted from the various layers in the protocol stack. Both the ACA and the INET have been implemented in Java, and the resulting code, along with its scripting framework and GUI interfaces, is called *J-Sim*. Finally, an essential suite of wireline and wireless network components and protocols have been implemented in *J-Sim* (Table I).

J-Sim possesses several desirable features. The fact that *J-Sim* is implemented in Java, along with its autonomous component architecture, makes *J-Sim* a truly platform-independent, extensible, and reusable environment. *J-Sim* provides a script interface that allows its integration with different script languages such as Perl, Tcl, or Python. (In particular, the latest release of *J-Sim* (version 1.3) has been fully integrated with a Java implementation of Tcl interpreter, called Jacl, with the Tcl/Java extension.) Therefore, similar to ns-2 (ns version 2) [6], *J-Sim* is a dual-language simulation environment in which classes are written in Java (for ns-2, in C++) and "glued" together using Tcl/Java. However, unlike ns-2, classes/methods/fields in Java need not be explicitly exported in order to be accessed in the Tcl environment. Instead, all the public classes/methods/fields in Java can be accessed (naturally) in the Tcl environment. For all these reasons, we have chosen *J-Sim* as the base environment to be augmented with a simulation framework for WSNs. Interested readers are referred to [57], [1] for a detailed qualitative and quantitative comparison between *J-Sim* and other network environments (such as ns-2 [22] and SSFNet [26]).

We have defined and built the proposed simulation framework for WSNs upon both the ACA and the INET, and specifies the components of (i) target, sensor and sink nodes, (ii) sensor channels and wireless communication channels, and (iii) physical media such as seismic channels, mobility models and power models (both energy-producing and energy-consuming components). Application-specific, new models can be defined by sub-classing appropriate classes defined in the simulation framework and (re)defining their behaviors. We demonstrate the use of the proposed simulation framework by implementing several well-known localization, geographic routing, and directed diffusion protocols. We show how these protocols can be readily implemented by sub-classing classes defined in the framework and customizing their behaviors (i.e., methods). We also perform detailed performance comparisons (in terms of execution time incurred, and the memory used) in simulating several typical WSN scenarios in *J-Sim* and *ns-2*. The simulation study indicates *J-Sim* and *ns-2* incur comparable execution time, but the memory allocated to carry out simulation (of length ≥ 1000 in *J-Sim* is at least two orders of magnitude lower than that in *ns-2*. As a result, *ns-2* often suffers from out-of-memory exceptions and was unable to carry out large-scale WSN simulation, while the proposed WSN framework in *J-Sim* exhibits good scalability.

We have also included in *J-Sim* a set of classes and mechanisms that realize network emulation in sensor network environments, where by network emulation, we mean the virtual simulation environment is integrated with a small number of real hardware devices to facilitate performance evaluation of real-life devices in a large-scale, but well-controlled environment [40], [29], [60]. As real-life packets have to be seamlessly transported between the two environments, the main challenge is to synchronize the virtual time used in the simulation engine with the wall time, and to convert packet headers and payloads from the real-life format to that used in the simulation environment. We leverage packet capturing tools (such as pcap [5] in Linux and Windows and SerialForwarder in TinyOS [34]) to capture real-life packets at the device driver level, redirect them through a raw socket to the user space, and perform proper reformatting. We also devise a mechanism to synchronize virtual clocks with the wall clock. We demonstrate the use of the proposed WSN framework in realistic scenarios by carrying out large scale, full-fledged Future Combat System (FCS) simulation [53].

The rest of the paper is organized as follows. We provide in Section II an overview of our simulation framework for WSNs, and give in Section III the implementation details of the simulation framework. In Section IV, we elaborate on how we leverage the simulation framework to implement protocols under three different categories: localization, geographic routing and directed diffusion. In Section V, we elaborate on how we realize in *J-Sim* network emulation for WSNs. In Sections VI–VII, we present a performance study. We first empirically evaluate the simulation framework in Section VI by simulating several typical WSN scenarios in J-Sim and *ns-2*. This is then followed by using the proposed WSN framework

to carry out a full-fledged FCS simulation in *J-Sim* in Section VII. Finally we give in Section VIII an overview of existing simulation environments, and conclude the paper in Section IX with a list of research avenues for future work.

II. Overview of the Proposed Simulation Framework

As mentioned in Section I, a major objective of wireless sensor networks is to monitor, and sense events of interests in, a specific environment. Upon detecting an event of interest (e.g., change in the acoustic sound, seismic, or temperature), *sensor nodes* send reports to *sink (user) nodes* (either periodically or on demand). Events (or termed as stimuli) are generated by *target nodes*. For instance, a moving vehicle may generate ground vibrations that can be detected by seismic sensors. From the perspective of network simulation, a wireless sensor network typically consists of three types of nodes: sensor nodes (that sense and detect the events of interest), target nodes (that generate events of interest), and sink nodes (that utilize and consume the sensor information).

Our simulation framework for WSNs is derived from the SensorSim simulation framework presented in [49], [15]. In a nutshell, sensor nodes detect the stimuli (signals) generated by the target nodes over a *sensor channel* and forward the detected information to the sink nodes over a *wireless channel*. Figure 1 depicts the top-most view of the proposed simulation framework [49], [48]. It should be noted that the nature of signal propagation between target nodes and sensor nodes over the sensor channel is inherently different from that between sensor nodes and sink nodes over the wireless channel. Two different models for signal propagation are therefore included: a *sensor propagation model* and a *wireless propagation model*. A sensor node is equipped with (1) a sensor protocol stack, which enables it to detect signals generated by target nodes over the sensor channel, and (2) a wireless protocol stack, which enables it to send reports to the other sensor nodes (and eventually to sink nodes) over the wireless channel. On the other hand, a target node has only a sensor protocol stack and a sink node has only a wireless protocol stack. A sensor node also has a *power model* that embodies the energy-producing components (e.g., battery) and the energy-consuming components (e.g., radio and CPU). Finally, in order to enable simulation of mobile nodes (e.g., moving tanks), a *mobility model* is included. Figures 2— 3 depict, respectively, the internal view of a target/sink/sensor node defined and implemented in the proposed simulation framework. All these nodes are constructed by sub-classing key classes in *J-Sim*.

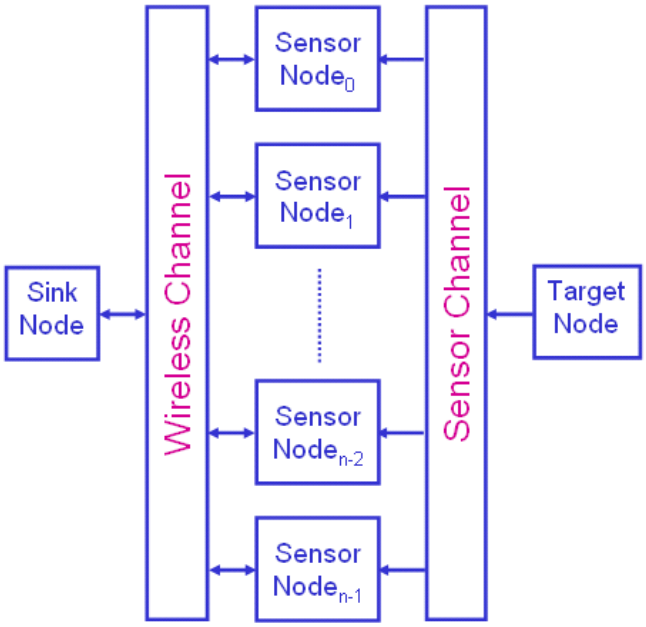
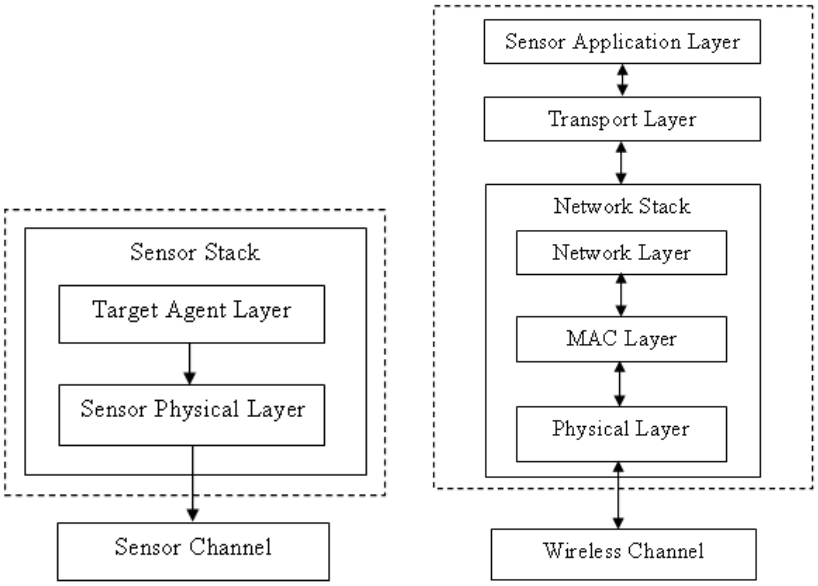


Fig. 1. The model of a typical wireless sensor network (WSN) environment.

The operation of the proposed simulation framework can be illustrated by considering a fairly simple event-to-sink transport protocol: A stimulus is periodically generated by a target node and propagated over the sensor channel. It should be noted that, as shown in Figure 2 (a), a target node can only send (but not receive) data packets over the sensor channel. The neighboring sensor nodes (e.g., sensor nodes that are within the sensing radius of the target node) will then receive



(a) Internal view of a target node (b) Internal view of a sink node

Fig. 2. Internal views of a target node and a sink node.

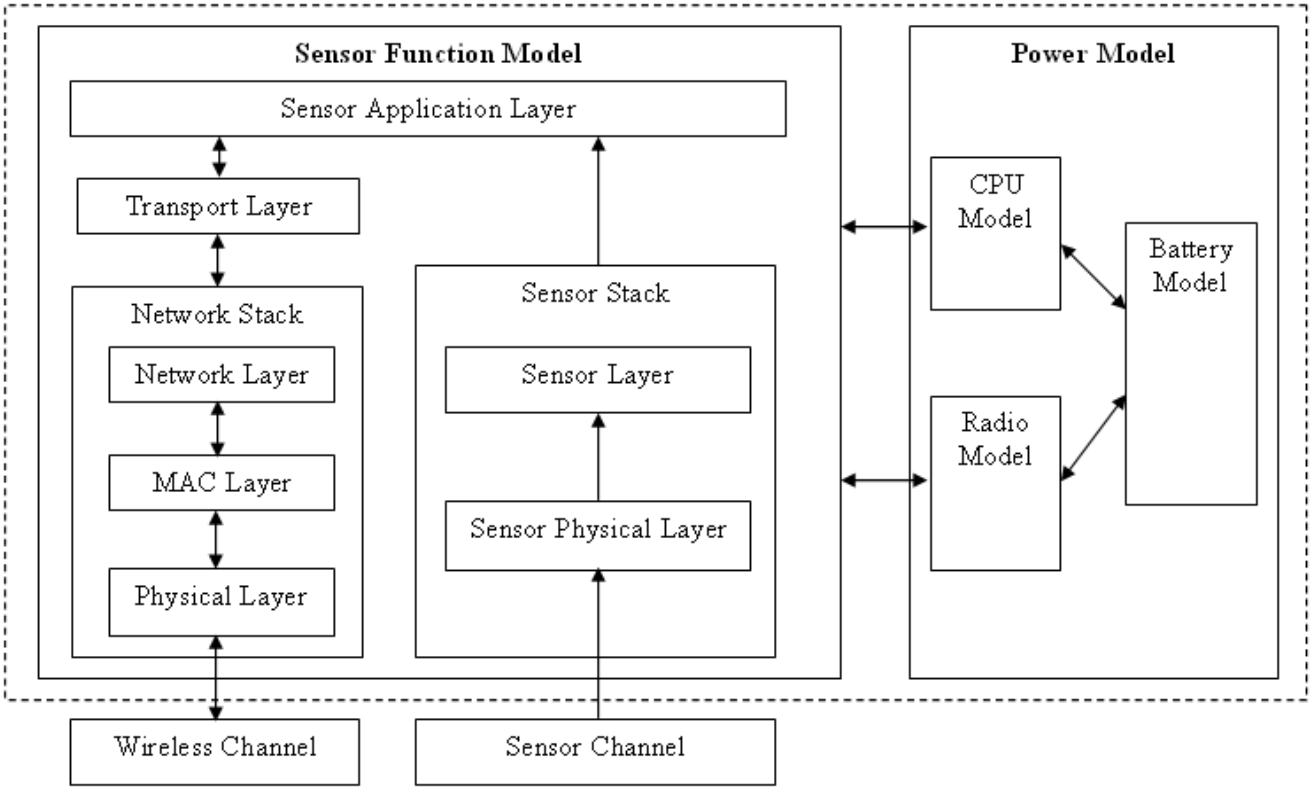


Fig. 3. Internal view of a sensor node (dashed line).

the stimulus over the sensor channel. As shown in Figure 3, a sensor node can only receive (but not send) stimuli over the sensor channel. However, due to the fact that the signal may be attenuated in the course of being propagated over the sensor channel, a sensor node receives and *detects* a stimulus only if the received signal power is at least equal to a pre-determined receiving threshold. The calculation of the received signal power is determined by the sensor propagation model used in the model (e.g., seismic or acoustic).

As mentioned above, each sensor node that receives *and* detects over the sensor channel has to forward its sensing result to one or more sink nodes over the wireless channel. Inside a sensor node (Figure 3), the coordination between the sensor protocol stack and the wireless protocol stack is done by the sensor application and transport layers. For instance, depending on the application for which the sensor network operates, a sensor node may either forward data packets as soon as they detects the stimuli, or process them first (e.g., compute the average temperature measured within a few minutes) and then forward processed data (e.g., the average temperature) to the sink node. Any in-networking processing mechanism such as that discussed in [32] can be implemented in the sensor application layer.

As the sink node may not be in the vicinity of a sensor node, communication over the wireless channel is usually *multihop*. Specifically, in order to send a packet from a sensor node s_i to a sink node snk_j, intermediate sensor nodes between s_i and snk_j have to serve as relays (routers) to forward that packet along the route from the source (s_i) to the final destination (snk_j). This implies why sensor nodes have to be able to both send and receive data packets over the wireless channel (as shown in Figure 3). As sensor nodes may fail or die of power depletion, the network topology of a WSN may change dynamically and the multihop routing protocol has to adapt to the topology change (e.g., ad hoc routing such as Ad-hoc On-demand Distance Vector routing (AODV) [51] or geometric routing such as Greedy Perimeter Stateless Routing (GPSR) [39], [43], [44]). The latest version of *J-Sim* includes classes for AODV, and we will elaborate on how we implement GPSR in the proposed framework in Section IV. Similar to signal propagation over the sensor channel, a sensor/sink node receives, and will further process, a data packet from the wireless channel only if the received signal power exceeds a pre-determined receiving threshold. Calculation of the received signal power is determined by the wireless propagation model used in the model. The latest release of *J-Sim* includes classes for three wireless propagation models: the free space model, the two-ray ground model, and the irregular terrain model [52].

The information received at the sink node over the wireless channel can be further analyzed by a control server and/or a human operator. Based on the content of the information, the sink node may have to send commands/queries to the sensor nodes. This explains why, as shown in Figure 2 (b), sink nodes have to be able to both send and receive data packets over the wireless channel.

As shown in Figure 3, the power model in a sensor node includes both the energy-producing components (e.g., battery) and the energy-consuming components (e.g., CPU and radio). The sensor function model (i.e., combination of the sensor protocol stack, the network protocol stack and the sensor application and transport layers) is subject to the power model. For example, the energy incurred in handling a received data packet is dictated by the CPU model, and the energy incurred in sending and/or receiving data packets is dictated by the radio model. In the proposed simulation framework, both the CPU and radio models can be in one of the several different operation modes. For example, the radio model can be in one of the following operation modes: *idle*, *sleep*, *off*, *transmit* or *receive*. The amount of energy consumed by an energy consumer depends on the operation mode in which the power model operates. The CPU and radio models can report their operation mode to the sensor function model, and the sensor function model can also change the operation mode of the CPU and radio models.

III. Detailed Description of the Proposed Simulation Framework

In this section, we describe the software architecture, and the implementation details, of the proposed WSN simulation framework in *J-Sim*. Specifically, we elaborate on the components and contracts defined and implemented in the simulation framework. A more detailed account of the names of ports in each component and the names of messages that are sent/received over these ports can be found in [56].

A. Target Node

In order to realize a target node (Figure 4), we have implemented the following classes in *J-Sim*:

1) ***TargetAgent*** provides the basic functionality of a target node. *TargetAgent* periodically generates stimuli (signals) and passes them to the lower layer in order to be transmitted over the sensor channel. *TargetAgent* is a subclass of *Module*, a key *J-Sim* class that has a *down port* (down@) used to pass data packets down to the lower layer in a protocol stack and a *timer port* (.timer@) used to set up (and cancel) timers. When the timer expires, a timeout()

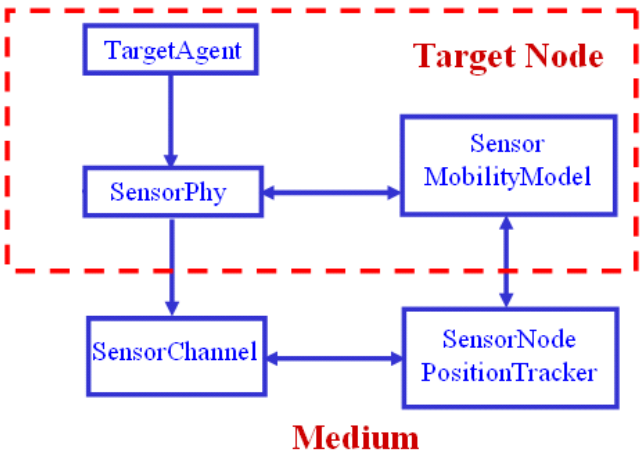


Fig. 4. Architecture of a target node (dashed line) with its connections to other components.

callback function is invoked to handle the timeout event, e.g., generating a new stimulus in this case. *TargetAgent* implements the *ActiveComponent* interface, a key *J-Sim* interface that has to be implemented for a component to be active as a data source.

2) ***TargetPacket*** implements the "packet" that encloses the stimulus (signal) generated by *TargetAgent*.
3) ***SensorMobilityModel*** maintains the location, speed and mobility pattern of a target node. *SensorMobilityModel* is a subclass of *MobilityModel*, a key *J-Sim* class that simulates the movement of mobile nodes. The coordinates of the location of a mobile node can be either specified in terms of (longitude, latitude, height) or (X, Y, Z). *MobilityModel* supports two different mobility models: trajectory-based and random waypoint. In a trajectory-based mobility model, a trajectory array provided by the user is used to needs to specify how target and sensor nodes move. A mobile node moves from one point in the trajectory to the next at a constant speed. In a random waypoint mobility model, a mobile node starts from its original location, randomly chooses a location in the simulated area as the destination, and moves in a straight line to that destination location at the speed that is uniformly distributed between 0 and a configurable maximum speed. When the mobile node reaches the destination location, it chooses a next destination location and repeats the procedure again.
4) ***SensorPhy*** implements the sensor physical layer. *SensorPhy* plays two different roles depending on whether an instance of it exists in the protocol stack of a target node or a sensor node. In the case that *SensorPhy* exists at a target node, its role is to receive a stimulus generated by *TargetAgent*, query *SensorMobilityModel* to get the up-to-date location of the target node and forward the generated stimulus (together with the location information) to the sensor channel component (*SensorChannel*). *SensorPhy* is a subclass of *Module* because both the *down port* (down@) and the *up port* (up@), provided by *Module*, are needed to pass data packets down to the lower layer (*SensorChannel*) and receive data packets from the higher layer (*TargetAgent*) respectively. When data arrives at the *up port*, a dataArriveAtUpPort() callback function is invoked to handle the newly received data.
5) ***SensorPositionReportContract*** defines the contract needed to define the information exchange between *SensorPhy* and *SensorMobilityModel*. *SensorPositionReportContract* is a subclass of *Contract*, a key *J-Sim* class that defines a contract; i.e., how an initiator (caller) and a reactor (callee) fulfill a certain function. It should be mentioned that a contract specifies the causality of information exchange between components but *not* the components that may participate in information exchange. Two components, acting respectively as the initiator and the reactor, are bound at system integration time to fulfill the contract. In this example, *SensorPhy* is the initiator (that sends a query message, as specified by the contract, requesting the up-to-date location of a node) and *SensorMobilityModel* is the reactor (that responds by providing the up-to-date location of a node to the initiator). At system design time, neither the initiator nor the reactor knows the identity of the other. The connection between the initiator and the reactor takes place only at the system integration time. This ensures a loosely-coupled component architecture.

B. Sensor Channel

In order to realize a sensor channel, we have implemented the following classes in *J-Sim*:

1) ***SensorNodePositionTracker*** maintains the location of all the nodes in a wireless sensor network. This location information is reported by the *SensorMobilityModel* component of each node to *SensorNodePositionTracker*. A major function performed by *SensorNodePositionTracker* is to determine which sensor nodes are within the sensing radius of a target node and hence, should receive the stimulus generated by that target node.
2) ***SensorChannel*** implements the sensor channel. The function of the sensor channel is that it receives a stimulus from a target node, queries *SensorNodePositionTracker* to get the list of sensor nodes that are within the sensing radius of that target node, and then sends the generated stimulus to each sensor node that is on the list after a fixed (but configurable) propagation delay.
3) ***AcousticChannel*** implements an acoustic channel. *AcousticChannel* is a subclass of *SensorChannel*. The propagation delay in *AcousticChannel* is a function of the speed of sound and the distance between the sender (i.e., target node) and receiver (i.e., sensor node). Specifically, the propagation delay (τ) is calculated according to the following equation:

$$\tau = \frac{\max(d, d_0)}{v},$$

where v is the speed of sound, d is the distance between the sender and receiver and d_0 is a configurable parameter of the acoustic channel.
4) ***SensorNeighborQueryContract*** defines the contract needed to define the information exchange between *SensorChannel* and *SensorNodePositionTracker*.

C. Sensor Propagation Model

In order to realize the sensor propagation model, we have implemented the following classes in *J-Sim*:

1) ***SensorPropagationModel*** is an abstract base class for different types of signal propagation models on the sensor channel.
2) ***SeismicProp*** implements a seismic propagation model. *SeismicProp* is a subclass of *SensorPropagationModel*. The major function of *SeismicProp* is to calculate the received signal power as a function of the distance between the sender (target node) and the receiver (sensor node) and the attenuation factor. Specifically, the received signal power (P_r) is calculated according to the following equation:

$$P_r = \frac{P_t}{\max(d, d_0)^{f_a}},$$

where P_t is the power with which the signal was transmitted, d is the distance between the sender (i.e., target node) and receiver (i.e., sensor node) and d_0 and f_a (signal attenuation factor) are configurable parameters of the seismic propagation model.
3) ***AcousticProp*** implements an acoustic propagation model. *AcousticProp* is a subclass of *SensorPropagationModel*. In *AcousticProp*, the received signal power (P_r) is calculated according to the following equation:

$$P_r = N(p \times \mu_g, \sigma_g^2),$$

where

$$p = \frac{P_t}{\max(d, d_0)^{f_a}}, \mu_g = U(min_g, max_g),$$

P_t is the power with which the signal was transmitted and d is the distance between the sender (i.e., target node) and receiver (i.e., sensor node). min_g, max_g, μ_g and σ_g^2 are respectively the minimum, maximum, mean and variance of the microphone gain. d_0, f_a (signal attenuation factor), min_g, max_g and σ_g^2 are configurable parameters of the acoustic propagation model.

D. Sensor Node

In order to realize a sensor node (Figure 5), we have implemented the following classes in *J-Sim*:

1) ***Battery Model*** includes the following classes:
   a) ***BatteryBase*** is an abstract base class for different types of battery models. *BatteryBase* defines the ports that are needed for any type of battery models (e.g., to interface with the CPU and radio models). *BatteryBase* is a subclass of *Component*, a key *J-Sim* class that implements a component in the autonomous component architecture (ACA) and provides basic functionality of a generic component (e.g., creating ports).

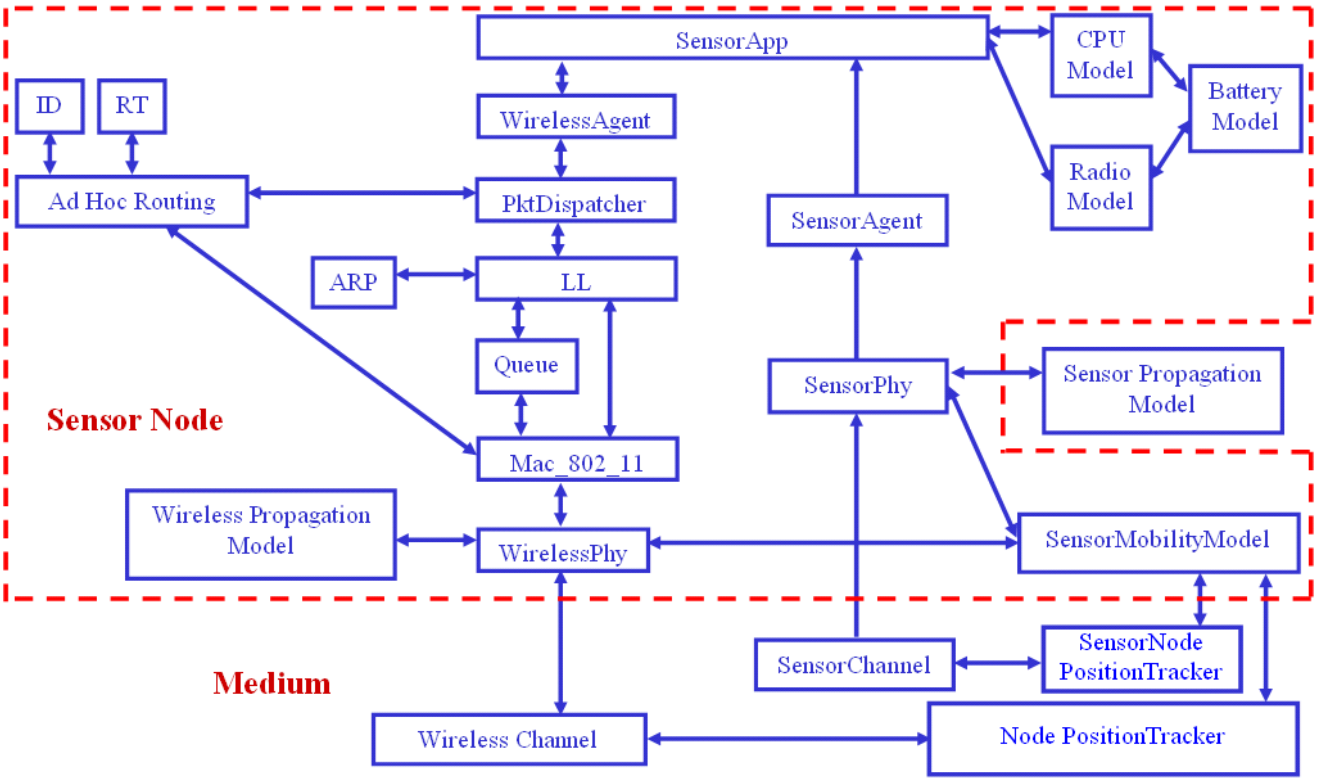


Fig. 5. Architecture of a sensor node (dashed line) with its connections to other components.

b) ***BatteryTable*** defines a table that specifies the capacity of a battery as a function of its current. Capacities corresponding to the current values that do not exist in the table are calculated by interpolation.

c) ***BatteryCoinCell*** implements a Coin Cell battery [50]. *BatteryCoinCell* is a subclass of *BatteryBase*. *BatteryCoinCell* contains a table (instance of *BatteryTable*) that specifies the capacity of the battery as a function of its current.

d) ***BatterySimple*** implements a simplistic battery model whose capacity is assumed to be always constant (i.e., not a function of the current). *BatterySimple* is a subclass of *BatteryBase*.

e) ***BatteryContract*** defines the contract needed to define the information exchange between the battery model and the CPU and radio models. For instance, the CPU and radio models need to inform the battery model about the amount of current that will be drained from the battery, depending on their operational modes.

2) ***CPU Model*** includes the following classes:

a) ***CPUBase*** is an abstract base class for different types of CPU models. *CPUBase* defines the ports that are needed for any type of CPU models (e.g., ports needed to interface with the battery model).

b) ***CPUAvr*** is a subclass of *CPUBase* and provides values of the current that has to be drained from the battery model in each of the CPU operation modes: *idle*, *sleep*, *off* or *active*.

3) ***Radio Model*** includes the following classes:

a) ***RadioBase*** is an abstract base class for different types of radio models. *RadioBase* defines the ports that are needed for any type of radio models (e.g., ports needed to interface with the battery model).

b) ***RadioSimple*** is a subclass of *RadioBase* and provides values of the current that has to be drained from the battery in each of the radio operation modes: *idle*, *sleep*, *off*, *transmit* or *receive*.

4) ***Sensor Protocol Stack*** includes the following classes:

a) ***SensorPhy*** As mentioned above, this class plays two different roles depending on whether its instance exists in the protocol stack of a target node or a sensor node. In the case that *SensorPhy* exists in the protocol stack of a sensor node, its role is to receive from the sensor channel a stimulus (signal) generated by a target node, the

location of the target node at the time of generating the stimulus and the power with which the stimulus was generated. *SensorPhy* then queries the sensor propagation model to calculate the received signal power (P_r). The current location of the sensor node, the location of the target node at the time of generating the stimulus and the power with which the stimulus was generated are included in the query. If (P_r) is below a certain receiving threshold (which is one of the member variables of *SensorPhy*), the signal is discarded, otherwise, it is forwarded up to the higher layer in the sensor protocol stack.

b) ***SensorPropagationQueryContract*** defines the contract needed to define the information exchange between *SensorPhy* and the sensor propagation model.

c) ***SensorAgent*** implements the sensor layer. *SensorAgent* receives the stimulus from the lower layer (*SensorPhy*) in the sensor protocol stack, computes/extracts the application-specific data (e.g., the strength and duration of the stimulus, the signal-to-noise ratio or the location of the target node) and forwards it up to the sensor application layer.

5) ***Sensor Application and Transport Layers*** include the following classes:

a) ***SensorApp*** implements the sensor application layer. *SensorApp* receives the application-specific data from *SensorAgent*, perform certain in-network processing tasks, and passes the resulting data digest down to the transport layer. The digest goes through the wireless protocol stack and will eventually be transmitted over the wireless channel to the sink node.

b) ***SensorPacket*** implements the data packet that will be transmitted over the wireless channel. *SensorPacket* can be either unicast to a specific destination (e.g., the sink node) or broadcast. *SensorPacket* is a subclass of *Packet*, a key *J-Sim* class for implementing data packets that are transmitted over wired/wireless networks.

c) ***WirelessAgent*** implements a transport layer between the sensor application layer and the wireless protocol stack of a sensor node. *WirelessAgent* receives from *SensorApp* the application-specific data that is to be sent to the sink node, encloses this data in a *SensorPacket* and passes it down to the wireless protocol stack in order to be eventually transmitted over the wireless channel to the sink node. *WirelessAgent* is a subclass of *Protocol*, a key *J-Sim* class for implementing transport protocols.

6) ***Wireless Protocol Stack*** of a sensor node is built in a plug-and-play fashion using the *J-Sim* classes that constitute the *J-Sim* wireless network extension [10]. *PktDispatcher* provides the functionality of the IP layer; i.e., the data sending/delivery services to the upper layer protocols. Specifically, it forwards incoming packets to an appropriate set of output ports connected either to an upper layer protocol or a lower layer component. *ARP* implements the address resolution protocol (ARP). *LL* implements the link layer functions. It receives IP packets (instances of the *InetPacket J-Sim* class) from *PktDispatcher*, queries *ARP* to find out the MAC address of the next hop to which the IP packet should be forwarded, encapsulates the IP packet in an *LLPacket* and inserts it in the interface queue of the underlying wireless interface card. Outgoing IP and ARP packets are buffered in the *Queue* component. *Mac_802_11* implements the IEEE 802.11 MAC protocol. *Mac_802_11* sends link failure notification messages to the ad hoc routing component in the case of link failures. *WirelessPropagationModel* implements the radio propagation model over the wireless channel. *WirelessPhy* implements functionalities of the physical layer of a wireless card. It queries *WirelessPropagationModel* to determine the received signal power and delivers a data packet only if the received signal power is at least equal to a certain receiving threshold.

E. Sink Node

As shown in Figure 6, a sink node can also be constructed in a plug-and-play fashion using a sensor application layer (*SensorApp*), a transport layer (*WirelessAgent*) and a wireless protocol stack as explained above.

IV. Demonstrative Use Of The Simulation Framework

In this section, we demonstrate how we leverage the simulation framework and implement three different protocols in WSNs: localization, geographic routing and directed diffusion. In each of the subsections below, we first give a succinct summary of each protocol and then elaborate on where (in which classes) and how we implement the protocol in the proposed simulation framework.

A. Localization

Localization in wireless sensor networks — how each sensor node obtains its accurate position, even in the presence of different geographic shapes of the monitoring region, different node densities, irregular radio patterns, and anisotropic

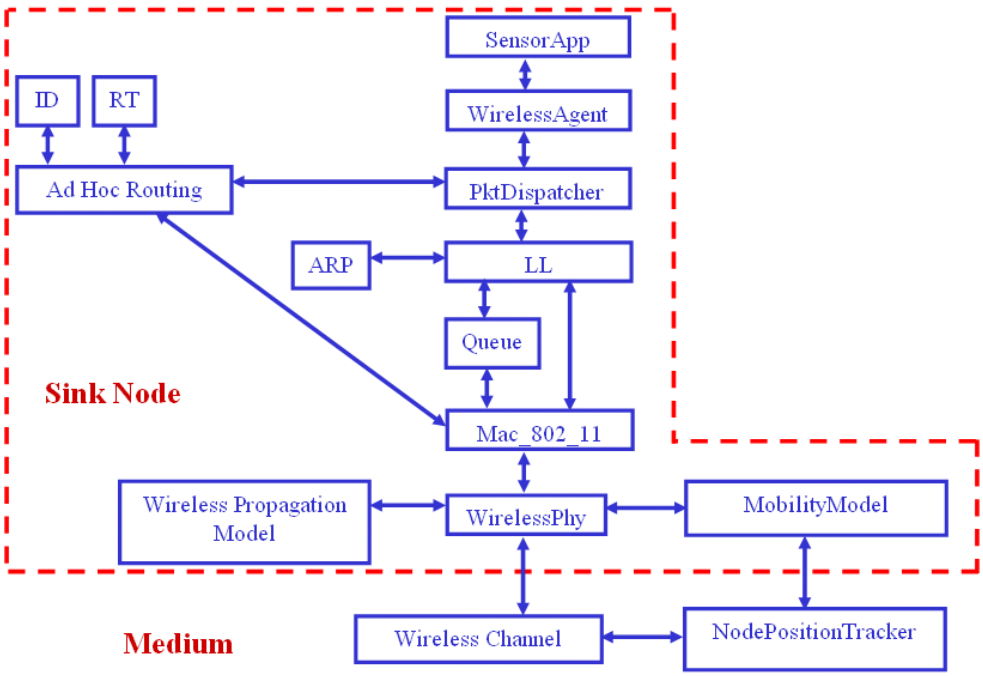


Fig. 6. Architecture of a sink node (dashed line) with its connections to other components.

terrain conditions — has become an important and critical issue in deploying WSNs. The most straightforward approach is to equip all the sensor nodes with a global positioning system (GPS) [24], but this approach is not scalable because of the cost and power consumption requirements. Several novel localization methods have been proposed to determine the positions of sensor nodes with *unknown* positions. A common assumption made in most of the methods is that a small portion of sensor nodes (called *beacon nodes*) are aware of their positions by means of manual configuration or using GPS [27], [47], [54], [46], [30]. A node with unknown position then estimates its distances to beacon nodes based on either the ranging techniques or the proximity measurements, and calculates its position with the use of lateration techniques (simplified versions of GPS triangulation).

To demonstrate how the localization service can be implemented in the framework, we have implemented a distributed positioning algorithm, called *APS/DV-hop*, [47]. In *APS/DV-hop*, each node exchanges distance tables that contain the locations of, and the hop-counts to, beacon nodes with its neighboring nodes. Once a beacon node obtains these distance tables from other beacon nodes, it computes an average per-hop distance by dividing the sum of distances to the other beacon nodes by the sum of hop-counts. A node with unknown positions then exploits the average per-hop distance calculated by each beacon nodes to estimate its geographic distance to each of the beacon nodes, and calculates its location by performing the lateration technique.

We embody *APS/DV-hop* in a *SensorLocApp* class, a subclass of *SensorApp* by extending the functionality of *SensorApp* and adding several member functions. A flag *SensorLocApp::isAnchor* in *SensorLocApp* determines whether the instance of *SensorLocApp* resides at a beacon node or a node with unknown position. Beacon nodes periodically send probing packets of *ProbePacket* that contain the position (*senderPos*), the average distance per hop (*hopDistance*), and the hop-count (*hopCount*). The probing period is defined in *SensorLocApp::probeInterval*. Upon receipt of a probing packet, nodes with unknown positions compute the geographic distance to beacon nodes by multiplying the average distance per hop by the hop-count, and estimate the geographic position by a gradient method that minimizes the squared sum of the differences between the measured distance and the computed Euclidean distance to beacon nodes. This gradient method is implemented in *SensorLocApp::estimateLocation*, while the updating gain and the threshold to check whether or not the gradient method converges are defined, respectively, in *laterationGain* and *laterationThreshold*.

B. Geographic Routing

As mentioned in Section II, after detecting and processing the stimuli generated by target nodes, sensor nodes have to forward sensed information (or a digest of it) to sink nodes. As sensors are usually *not* associated with IP addresses, but instead are attributed by their geographic positions, geographic routing has been used to route data packets (that contains sensed information) to sink nodes. Most of the geographic routing protocols operate under the assumption that each node knows its own geographic position and nodes can exchange their position information with their neighbors. Instead of building a routing table with the use of shortest paths and transitive reachability, geographic routing protocols make hop-by-hop routing decisions by using geographic positions of nodes.

To demonstrate how geographic routing can be incorporated into the proposed simulation framework, we have implemented in *J-Sim* the Greedy Perimeter Stateless Routing (GPSR) algorithm [39]. Conceptually, GPSR uses, whenever possible, *greedy forwarding* to forward packets to nodes that are the closest (and progressively closer) to the destination. If no neighbor is closer to the destination than the current node, GPSR forwards packets using the *perimeter mode*. In order to forward packets in the perimeter mode, GPSR performs polarization by constructing the Relative Neighborhood Graph (RNG) or Gabriel Graph (GG) (both of which are planar graphs), either periodically or when a node cannot perform greedy forwarding. In the *perimeter mode*, GPSR forwards packets using a simple planar graph traversal, in which a packet traverses the faces on the constructed planar graph successively closer to the destination. GPSR resumes greedy forwarding when the packet reaches a location that is closer to the destination than the location where greedy forwarding failed previously to forward that packet.

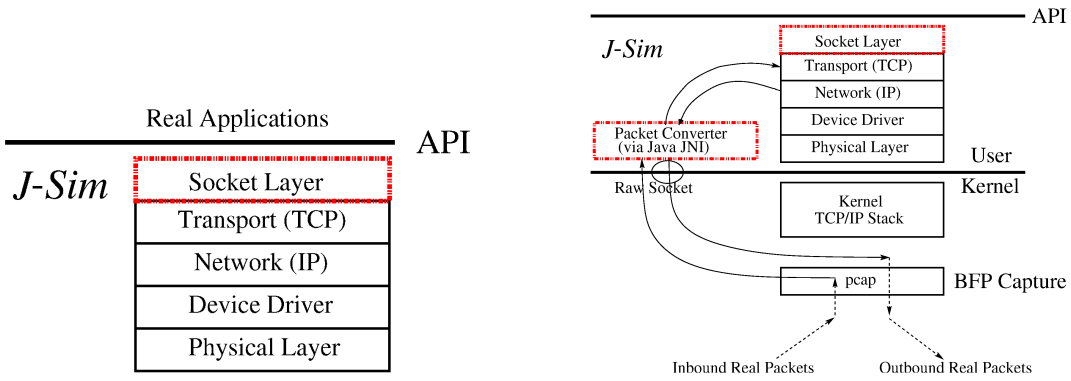
We have implemented GPSR as an active component by sub-classing the *Routing* class in *J-Sim*. Unlike other routing protocols which pre-compute routing tables and forward packets based on the pre-computed routing tables, GPSR has to calculate the next hop node for each incoming packet (unless the current node is the destination). In a node which uses GPSR as the underlying routing protocol, both the *PktDispatcher* component and the *AdHocRouting* component (Fig. (5)) are replaced by the *GPSR* component. Specifically, GPSR contains an up port which is connected to the transport layer or the *WirelessAgent*, and a down port which is connected to the *LinkLayer*. Similar to the *AdHocRouting* component, *GPSR* has an ID port that is connected to the *Identity* component, so as to obtain the identity information. However, unlike the *AdHocRouting* component, *GPSR* does not possess a port connected to the *RoutingTable* component. On the other hand, to obtain the position information under node mobility, *GPSR* contains a *mobility* port that is connected to the *NodePositionTracker* model, and and a *LinkBrokenEvent* port that receives link broken events from the Mac layer.

C. Directed Diffusion

Directed diffusion is a *data-centric* information dissemination paradigm for WSNs [35], [36], [31]. Conceptually, *data* in sensor networks is the collected or processed information of a physical phenomenon. In directed diffusion, a sink node periodically broadcasts an *interest* message, containing the description of a sensing task that it is interested in knowing (e.g., detecting a vehicle in a specific area), to its neighbors. *Interest* messages are diffused throughout the network (e.g., via selective flooding) and set up *gradients* within the network. Specifically, a gradient is direction state created in each node that receives an *interest* message. The gradient direction is set toward the neighboring node from which the interest message is received.

When an *interest* message arrives at a sensor node that senses data with the matches the interest, the sensor node prepares data messages, each of which includes an event description. The sensor node marks these data messages as *exploratory* and sends them to each neighbor for whom it has a gradient. Exploratory data is reproduced every *exploratory interval*. A node that receives a data message from one of its neighbors may forward the data message to each neighbor for whom it has a gradient. As a result, exploratory data messages are forwarded toward the originators of interests along (possibly) multiple gradient paths. Upon receipt of exploratory data, a sink node *reinforces* its preferred neighbor based on several data driven local rules. For instance, the sink node may reinforce any neighbor from which it receives a previously unseen event. To reinforce a neighbor, the sink node sends a *positive reinforcement* message to the neighbor to inform it of sending data at a smaller interval (i.e., higher rate) than the exploratory interval, thereby establishing a reinforced gradient towards the sink node. The reinforced neighbor reinforces its neighbor in turn, all the way back to the data source, resulting in a chain of reinforced gradients from all sources to all sinks. Subsequent data messages, which are sent on reinforced gradients, are not marked as exploratory.

We have implemented directed diffusion by defining a new class *DiffApp*, a subclass of *SensorApp*. Each of the interest and reinforcement messages is enclosed in a *SensorPacket* and transmitted over the wireless channel. Since all the communication activities in directed diffusion are neighbor-to-neighbor, the *AdHocRouting* component in Figs. 5-6 is



(a) top-down network emulation (b) bottom-up network emulation

Fig. 7. Network emulation in *J-Sim*.

not needed. On the other hand, an interest cache and a data cache are required to maintain the gradients established on the path(s) from a sink node to sensor nodes with matched interest information. Each item in the interest cache corresponds to a distinct interest and stores the information of the gradients that a node has to each of its neighbors for that interest. The data cache keeps track of recently seen data items, and facilitates in-network processing (e.g., data aggregation in which identical data sent by different sources are suppressed). In particular, the data cache is used to determine from which neighbor a node first received the latest event matching an interest. This information is needed to determine the preferred neighbor that should be positively reinforced. Periodically, both the interest and data caches are purged to delete stale entries.

The data-driven local rules (that are used for positive and negative reinforcements) are implemented in the member functions of the *DiffApp* class. The rule for positive reinforcement is to reinforce a neighbor, which sends a previously unseen event (i.e., the neighbor from whom a node first received the latest event matching the interest). The rule for negative reinforcement is to negatively reinforce a neighbor from whom no new events have been received within a window of N events (i.e., the neighbor that consistently sends previously seen events). Other rules for positive and/or negative reinforcements can be defined by overriding the corresponding member function(s) in *DiffApp*.

V. Incorporating Network Emulation into the Simulation Framework

As mentioned in Section I, network emulation is an inexpensive approach to testing, validating, and/or evaluating protocols/approaches in a realistic but well-controlled network environment. The protocol/mechanism to be tested is usually executed in the real environment, while other components that interact with the tested protocol/mechanism are executed in the well-controlled, virtual environment. In this section, we elaborate on how we realize network emulation for wireless sensor networks in *J-Sim*.

We have realized the notion of network emulation in *J-Sim* in both the *top-down* (Fig. 7 (a)) and *bottom-up* (Fig. 7 (b)) fashions. In the *top-down* approach, we develop a Java-compliant socket layer, on top of which real applications can be readily ported. As shown in Figure 7(a), the socket layer essentially gives applications the illusion that they are interfacing with the operating system, rather than with a virtual network environment. In the *bottom-up* approach, real-life packets are intercepted at the device driver level and transported to the *PacketConverter* that converts packet headers and payloads from the real-life format to that accepted by *J-Sim*. Packets can then be directed to different layers (depending on the amount of header information that is retained during the conversion) as desired. As shown in Figure 7(b), to implement this technique, we have leveraged the packet capturing facility (e.g., `pcap` in Linux and Windows [5]) to intercept real-life packets and re-direct them to the *PacketConverter*. Outbound packets will be processed by the *PacketConverter* and directed via IP raw sockets to real device drivers.

We have further extended the notion of network emulation to Berkeley Mica motes-based WSNs (Figure 8). Berkeley motes, equipped with sensors and RF circuitry, are used as the real "small dust" devices to extract physical environment data. With the use of the *SerialForwarder* program, a generic two-ways communication tool (that comes with the TinyOS distribution), real-life data are relayed from motes to a I/O device and vice versa[1]. These real-life packets are then

[1]The current implementation of *SerialForwarder* only supports serial links of PCs.

intercepted at the serial link, converted by the *PacketConverter* for proper formats, and fed into one of the virtual *J-Sim* classes.

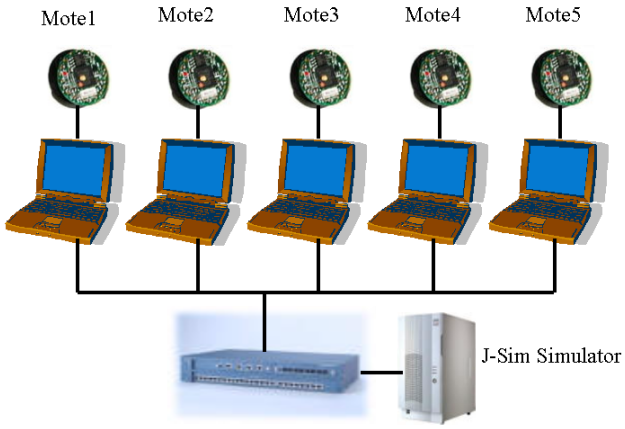


Fig. 8. A system setting for network emulation in wireless sensor networks.

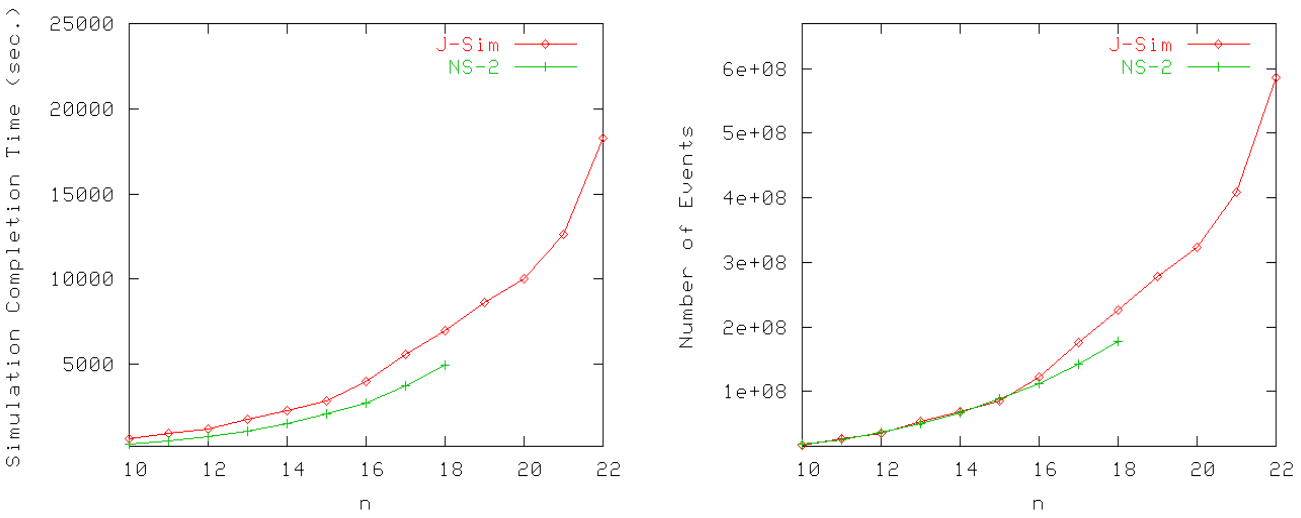
We have defined and implemented several forms of network emulation in WSNs (from the simplest to the most complex):

(i) Extracting sensor data from real devices: In this form, motes simply serve as sensor hardware that provides one-way data traffic from real devices to the simulation environment. Specifically, the functionality of the *SensorPhy* component is implemented in motes. All the other WSN functions such as in-network processing and information relay (to the sink nodes), are simulated in *J-Sim*. In the system initialization phase, the program running in motes accepts the command from the virtual component in *J-Sim* to determine the type of sensing signal and the sampling rate. The sensing data that arrives at the *SensorApp* component in *J-Sim* is considered as the data that originates from a virtual *SensorPhy* component.

(ii) Processing sensor data in real devices: Compared with the first form, the task of processing sensor data is moved from the simulation environment to real devices. The functionalities of both the *SensorPhy* and *SensorApp* components are implemented in motes, while the Communication over the shared wireless channel is still simulated in *J-Sim*. In this form, packets are forwarded bi-directionally between motes and *WirelessPhy* components in *J-Sim*.

(iii) Processing and transmitting sensor data in real devices: In this form, both data processing and wireless communications take place in real devices (as well as in the simulation environment). In this form, real devices communicate with virtual sensor nodes, and have to synchronize their operations and wireless communication events in the shared channel. We leverage the fact that the simulation paradigm *J-Sim* adopts is real-time process-driven simulation. Specifically, each event in *J-Sim* is executed in an independent execution context and event executions are carried out in real time as opposed to at fixed time points in discrete-time event-driven simulation. The simulation engine in *J-Sim* is simply the ACA runtime with one additional function: a "leap forward" operation is performed in time to the nearest future so that at least one execution context can be active. The simulation engine keeps track of the following three variables: (i) last_time_updated: the last (wall) time at which the current simulation time is adjusted; (ii) time_scale: the ratio of the wall time to the simulation time; and (iii) time_advances: the amount of time units (in simulation time) advances so far. The simulation engine then calculates the current simulation time as:
current_simulation_time = (current_wall_time - last_time_updated)/ time_scale + time_advances;
and properly updates the variables when the simulation time advances:
time_advances += nearest_simulation_future_time - current_simulation_time; and
last_time_updated = current_wall_time.
To synchronize the operations and interaction between real devices and virtual sensors, we set *time_scale* to 1 and *time_advances* to zero (i.e., no time advance is allowed in the simulation environment).

With these utilities in place, we will be able to interface modeling and simulation modules with real systems and validate



(a) Execution time

(b) Number of events

Fig. 9. The execution time and the number of events versus the network size $n^2 + 2$.

real-life systems prototypes in much larger, but still well-controlled networking environments.

VI. Performance Evaluation Of The Simulation Framework

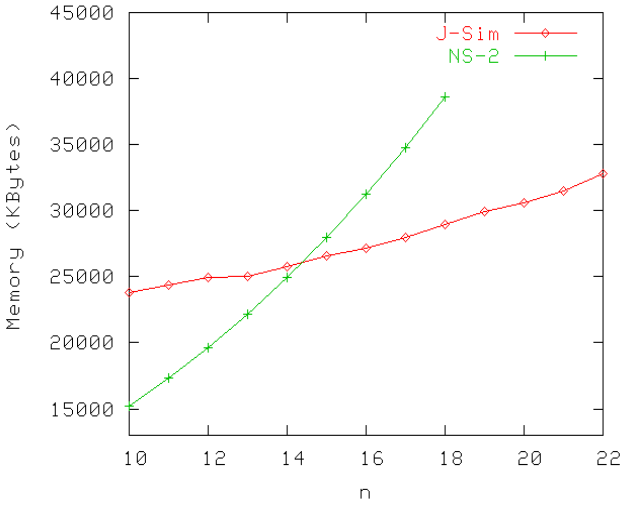
In this section, we conduct a performance study to evaluate the proposed *J-Sim* simulation framework and compare its performance with that of *ns-2* in several typical WSN scenarios. (Due to the page limit, we report below only results from two representative scenarios.) We study the effect of network sizes on (i) the execution time required to complete a simulation runs, and (ii) the number of events thus generated and the memory thus consumed. Note that the execution time includes both the time required to set up the nodes (i.e., the time incurred in creating and configuring the network before the simulation starts) and the time required to conduct a T-second simulation run. All the experiments are conducted on a dual-processor AMD Athlon 1.5 GHz machine running Red Hat Linux kernel 2.6.6 with 3.5 GB RAM. Each data point reported below is an average of 20 simulation runs.

A. Scenario A: Target Tracking

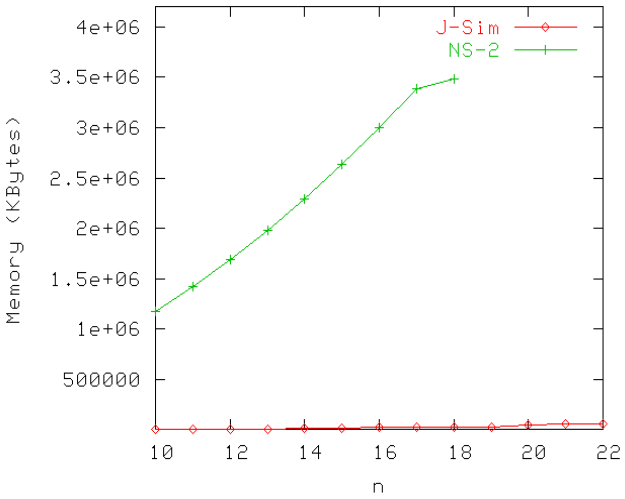
The simulation scenario consists of one sink node, two target nodes and $n^2 - 1$ sensor nodes. The size of the WSN is controlled by increasing n from 10 to 22. The sink node is located at the origin $(0, 0)$, while the sensor nodes are evenly distributed over a 1500×1500 m^2 region. The two target nodes are initially located at $(0, 1500)$ and $(0, 1350)$, and move according to the random waypoint model with a maximum speed equal to 10 $m/sec.$. Each target node generates a stimulus every one second, and the sensing radius is 200 m. The seismic propagation model is used on the sensor channel with $d_0 = 1.0$ and $atnFactor = 1.0$. The receiving threshold in the sensor physical layer is set to 3.0. In the wireless protocol stack, the transmission power is set to 0.2818 W (for a 260 m transmission range), the receiving threshold is set to 1.0×10^{-11} and the carrier sense threshold is set to 1.0×10^{-12}. Directed diffusion is used as the information dissemination paradigm, GPSR provides the localization service, and AODV is used as the underlying ad-hoc routing protocol. The simulation time, T, is 1000 sec.

Figure 9 gives the execution time and the number of events generated versus the network size $n^2 + 2$. As shown in Fig. 9, the graphs corresponding to *ns-2* are cut off at $n = 18$ because *ns-2* ran out of memory for $n > 18$. *J-Sim* incurs a comparable though longer execution, even though the number of events generated is almost the same (especially in the range $10 \leq n \leq 16$) as that in *ns-2*. This result is not surprising because a Java program is inherently slower than a C/C++ program. Specifically, the execution time in *J-Sim* is up to 41.6 % higher than that in *ns-2* and the number of events generated by *J-Sim* is up to 27.5 % higher than that in *ns-2*.

We have also measured the amount of memory allocated before the start *and* before the end of the simulation. The memory usage before the start of the simulation represents the amount of memory allocated to set up the nodes and all the other components in the simulation (e.g., wireless and sensor channels), while that before the end of the simulation

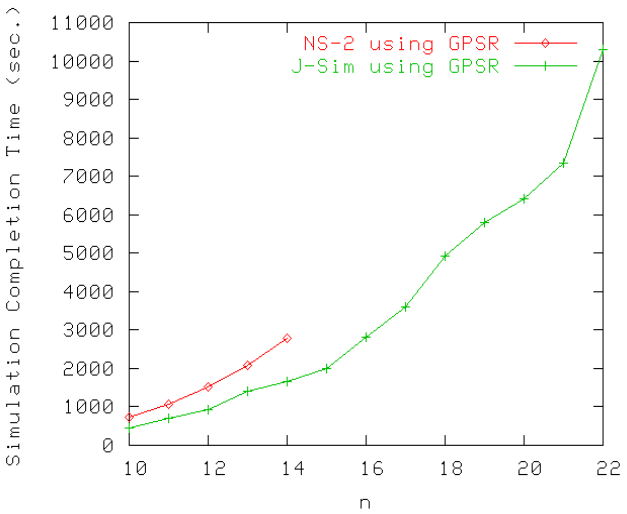


(a) Memory usage - before the simulation

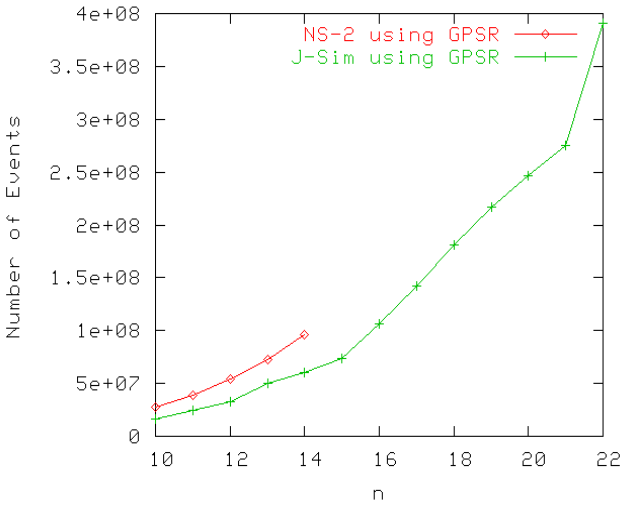


(b) Memory usage - before the end of the simulation

Fig. 10. Memory usage before the start and ending of the simulation versus the network size $n^2 + 2$.



(a) Execution time



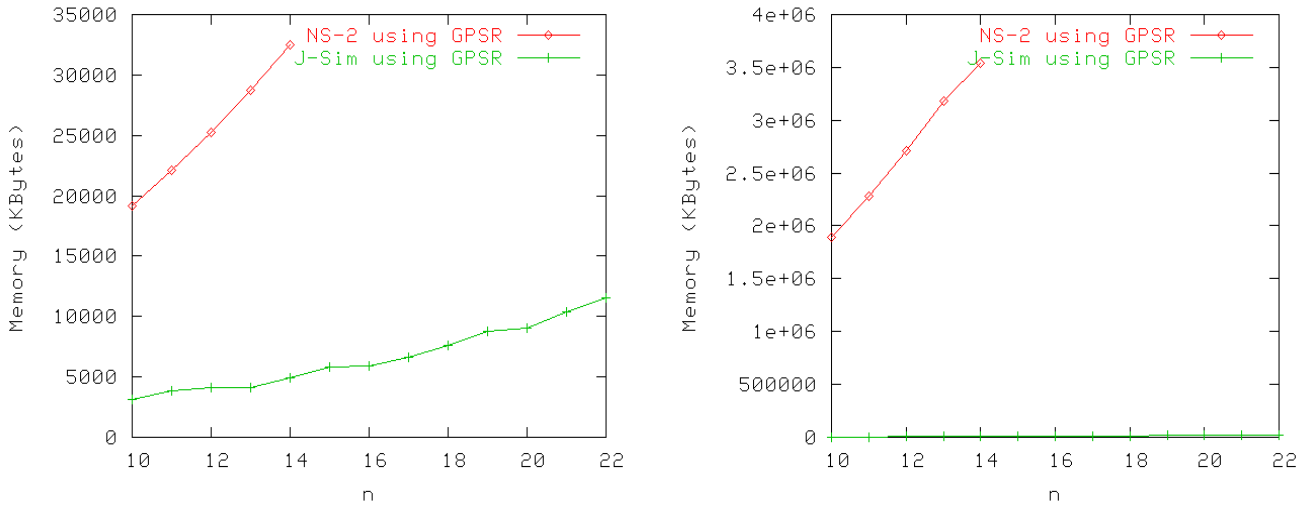
(b) Number of events

Fig. 11. The execution time and the number of events versus the network size $n^2 + 2$. GPSR is used as the underlying routing protocol.

represents the amount of memory allocated to complete the 1000-sec. simulation. As shown in Figure 10(a), the rate of increase in memory usage before the start of the simulation in ns-2 is higher than that in *J-Sim* causing *J-Sim* to outperform ns-2 for $n \geq 15$. This demonstrates that the data structures are used in a more scalable manner in *J-Sim* to represent different classes and their interaction in the WSN framework. In addition, as shown in Figure 10(b), the memory allocated to complete the 1000-sec. simulation in *J-Sim* is at least two orders of magnitude lower than that in ns-2. This is credited to the better garbage collection mechanism used in Java to reclaim unused memory.

B. Scenario B: Using GPSR (Instead of AODV) as the Routing Protocol

The simulation scenario is identical to that in the previous subsection except that now GPSR is used as the underlying routing protocol. Figure 11 gives the execution time and the number of generated events versus $n^2 + 2$. The graphs for ns-2 are cut off at $n = 14$ now because ns-2 ran out of memory again for $n > 14$. Again we observe comparable performance between ns-2 and *J-Sim* in terms of these two metrics. In fact, *J-Sim* incurs a smaller execution time to carry out simulation. As comparing with AODV, GPSR generates much less events (and hence much less execution time) as



(a) Memory usage - before the simulation

(b) Memory usage - before the end of the simulation

Fig. 12. Memory usage before the start and end of the simulation versus the network size $n^2 + 2$. GPSR is used as the underlying routing protocol.

AODV incurs significant routing overhead in exchanging routing information and maintaining its routing table.

Figure 12 gives the memory usage before the start ((a)) and end ((b)) of the simulation under ns-2 and *J-Sim*. Again the memory usage in *J-Sim* is at least two orders of magnitude lower than that in ns-2. As compared with AODV, GPSR incurs much less memory as well, again due to the fact that AODV incurs significant routing overhead.

C. Summary

The ability of *J-Sim* to carry out simulation for $n > 18$ at the cost of a reasonable amount of memory (the maximum memory usage observed in *J-Sim* for these two sets of simulation was 61.25 MBytes) demonstrates the scalability of the WSN simulation framework in *J-Sim*. This, coupled with the fact that the proposed framework inherits the virtue of the component architecture (ACA) and is hence more extensible (i.e., new components can be defined by sub-classing appropriate base classes and readily inserted into the framework with *matched contracts*), make the proposed WSN framework a very attractive candidate in evaluating newly emerging protocols for WSNs and their composite performance under various simulation scenarios.

VII. Simulation and Emulation of Future Combat Systems

To demonstrate the use of the proposed WSN framework in realistic scenarios, we have carried out large-scale, full-fledged simulation and emulation of FCS (as defined in [53]), that includes the satellite link model, (omni-directional and directional) antenna models [33], [42], the irregular terrain model [52] (that emulates TIREM [16]), the IEEE 802.11 MAC interference and contention model, the ad hoc routing protocol (AODV [51] or GPSR [39]), and IP packet forwarding. In what follows, we first give a succinct overview of FCS, and then discuss, for the comopleteness of the paper, implementations specific for FCS simulation.

Overview of FCS*:* The notion of FCS is supported by DARPA in view of developing network-centric, multi-mission combat systems. One of the major missions is to design lethal, strategically deployable, self-sustaining and highly survivable combat system through the use of networked manned and unmanned ground and air platforms, sensor networks, and weapon systems. As shown in Fig. 13, the FCS is envisioned to have a hierarchical communication structure with three layers: i) ground units, including troops, vehicles and sensors that are geographically distributed over a battlefield and form one or more ground ad-hoc networks; ii) low-altitude unmanned aerial vehicles (UAVs) for surveillance and maintenance of connectivity of ground ad hoc networks; and iii) satellites that connect UAVs and some of the ground vehicles with the joint force command center. Soldiers are only equipped with short-range radio and can only communicate with other ground units that are within the transmission range of each other. Vehicles are quipped with long-range radio and can communicate with other ground units and UAVs. Some of the vehicles are also equipped with satellite antennas and can communicate with satellites. UAVs are equipped with long-range radios and satellite antennas, and can communicate

Fig. 13. A typical FCS scenario. Also shown is how network emulation is performed by transporting real-life images captured by a WebCam (that emualtes the view of a soldier) through the *J-Sim* virtual environment to a real-life device PDA (that emulates the command center) and virtual nodes in *J-Sim*.

with both the ground vehicles and satellites. The satellites connects the UAVs and some of the ground vehicles with the command center. In addition, sensors self-organize themselves into a WSN, serve to track enemy vehicles (with different signatures of acoustic sounds), and relay sensor information to one or more of the ground vehicles.

All the ground/air platforms, sensor networks, and systems are prohibitively expensive to be readily deployed in trial tests. Moreover, tuning of systems parameters to optimize the performance is extremely time-consuming in real life. As a result, there has been a pressing need to carry out high-fidelity (and preferrably real-time) simulation of a full-fledged FCS based on real-life traces.

Classes implemented for FCS simulation*:* To faithfully simulate the FCS scenario, several components have to be implemented: (1) a full-fledged networking stack including MAC, routing, transport and application layer protocols; (2) detailed radio propagation models and interference models to accurately model the physical characteristics of wireless channels; (3) (irregular) terrain model to take into account of the physical signal attenuation effect due to terrains; and (4) interfaces to allow input of real-life traces that give the movement characteristics of military units in the battlefield. Furthermore, in order to emulate different types of military wireless links (ground-to-ground, ground-UAV, ground-to-satellite, and UAV-to-UAV), each warfare entity must be equipped with different parameters (e.g. sending power, frequency, bandwidth, receiving threshold, carrier sense threshold, and interference threshold). We have leveraged both the wireless network extension and the proposed WSN framework to define and implement ground ad hoc networks (composed of ground units) and suvelliance sensor networks. We have also devised RTI-compliant trace interfaces in *J-Sim* to read real-life movement traces of ground entities provided by SAIC, Inc. In addition, we have completed the following implementation:

UAV Placement*:* One of the major functions of UAVs is to maintain the connectivity of the ground ad hoc networks. To this end, we have designed and implemented a UAV placement algorithm that continuously optimizes, based on the movement information of ground entities and the path loss information the altitudes and the flight paths of UAVs, so as

to maintain network connectivity and to recover from temporary network partition [45]. The UAV placement algorithm is periodically executed in the simulation. In each run, the algorithm first builds connected components of ground units based on the Irregular Terrain Model (ITM) [52]. It then sorts the connected components in the decreasing order of their sizes (where the size of a componenet is the number of ground units in the component) and then searches for (near) optimal locations for UAVs to connect components in this order. Finally, for each UAV location newly determined, the UAV that is closest to the new location in the last run is dispatch to fly to that location.

Network emulation*:* To demonstrate the capability of network emulation, we have ported a real-life WebCam application on top of *J-Sim* and transported real-life images captured by WebCam through the *J-Sim* virtual simulation environment to another real-life device, a PDA. The WebCam is associated with a (virtual) vehicle node in *J-Sim*, and is used to emulate the view of a soldier that operates the vehicle. The PDA, on the other hand, is associated with the (virtual) command center in the simulation scenario. Network emulation is then used to demonstrate how the view of a soldier in the battlefield can be transported, with the use of the top-down approach (Section V), to the remote command center, and how its quality is subject to the mobility of ground entities, the terrain effect, and the paramter setting in the protocol stack.

J-Sim 3D terrain visualization*:* We have also implemented, based on Java3D [4] technology, a 3D terrain visualization tool (Fig. 14) as a component in *J-Sim*. The terrain visualization tool reads the altitude data of any location on the earth from the GLOBE (Global Land One-km Base Elevation [11]) database, and renders the terrain. It also displays the movement of all the ground vehicles, soldiers, and UAVs. The 3D terrain images can be zoomed in/out, translated, and/or rotated. Performance statistics (such as the number of packets sent/received) pertaining to a node can be display in real-time by clicking on the particular node.

Use of wireless sensor networks in target tracking in FCS*:* With the above implementation, we are able to simulate various FCS scenarios. One scenario that pertains to the use of WSNs is to simulate how WSNs can be used to facilitate target tracking in a FCS scenario. The FCS contains 121 sensors that are evenly distributed over a 11×11 square mile region, 40 ground vehicles, three UAVs, and one command center. The sensors are equipped with acoustic sensors to detect enemy vehicles (that generates accoustics sound of supposedly different signatures). The radio frequency is chosen to be 900MHz for short-range radio and 2GHz for long-range radio to avoid collision between the two types of radios. The transmission power is set to be 0.2818 W and the receiving threshold is set to be $1.58 \times 10^{-10}W$ for short range radios and $2.7 \times 10^{-11}W$ for long range radios. The wireless bandwidth of each sensor is set to 20Kbps and the radio bandwidth of all the other entities is set to 2Mbps. The carrier sense threshold is set to be the receiving threshold divided by 10.

In the simulation, two enemy tanks enter this battlefield. The sensors detect their exitence, continuously monitor their movement and send their locations and moving directions to one of the ground vehicles (that acts as the sink). After being relayed the information, the ground vehicle sends control messages to several ground vehicles to chase after the enemy vehicles, while the UAVs continuously adjust their altitudes and fly paths to maintain connectivity of ground vehicles. Figure 15 shows a snapshot of the movement information of the two enemy tanks received at the sink node. A video demo of the entire simulation can be downloaded at [2].

VIII. Related Work

In this section, we give an overview of existing network simulators that support wireless network simulation and/or sensor network simulation. Examples of network simulators that support wireless network simulation are GloMoSim [8] and OPNET [12]. Examples of network simulators that support both wireless network simulation and sensor network simulation are ns-2 [6] and Ptolemy [7].

GloMoSim (Global Mobile Information System Simulator) [21], [8] is a simulation environment for wireless mobile networks. GloMoSim is designed as a set of modules in the layered architecture; each module simulates a specific protocol in the protocol stack. GloMoSim has been designed using the parallel discrete-event simulation capability provided by PARSEC (Parallel Simulation Environment for Complex Systems) [20], [13]. PARSEC is a C-based sequential and parallel simulation language, which can be used to program new modules that can be added to GloMoSim.

OPNET [12] is a commercial network modeler and simulator provided by OPNET Technologies, Inc. In OPNET, a network is modeled in a hierarchical approach that closely matches the hierarchical architecture of the Internet: networks, subnets and nodes (fixed, mobile or satellite). Each node is modeled as a set of processes where each process is modeled as a finite state machine (FSM). The entire network is simulated using a discrete-event simulator. OPNET supports three types of links: point-to-point, bus and wireless. A wireless link is used in wireless, mobile or satellite network simulation. Each stage in the 14-stage transceiver pipeline models an aspect of the channel's behavior (e.g., line-of-sight, signal

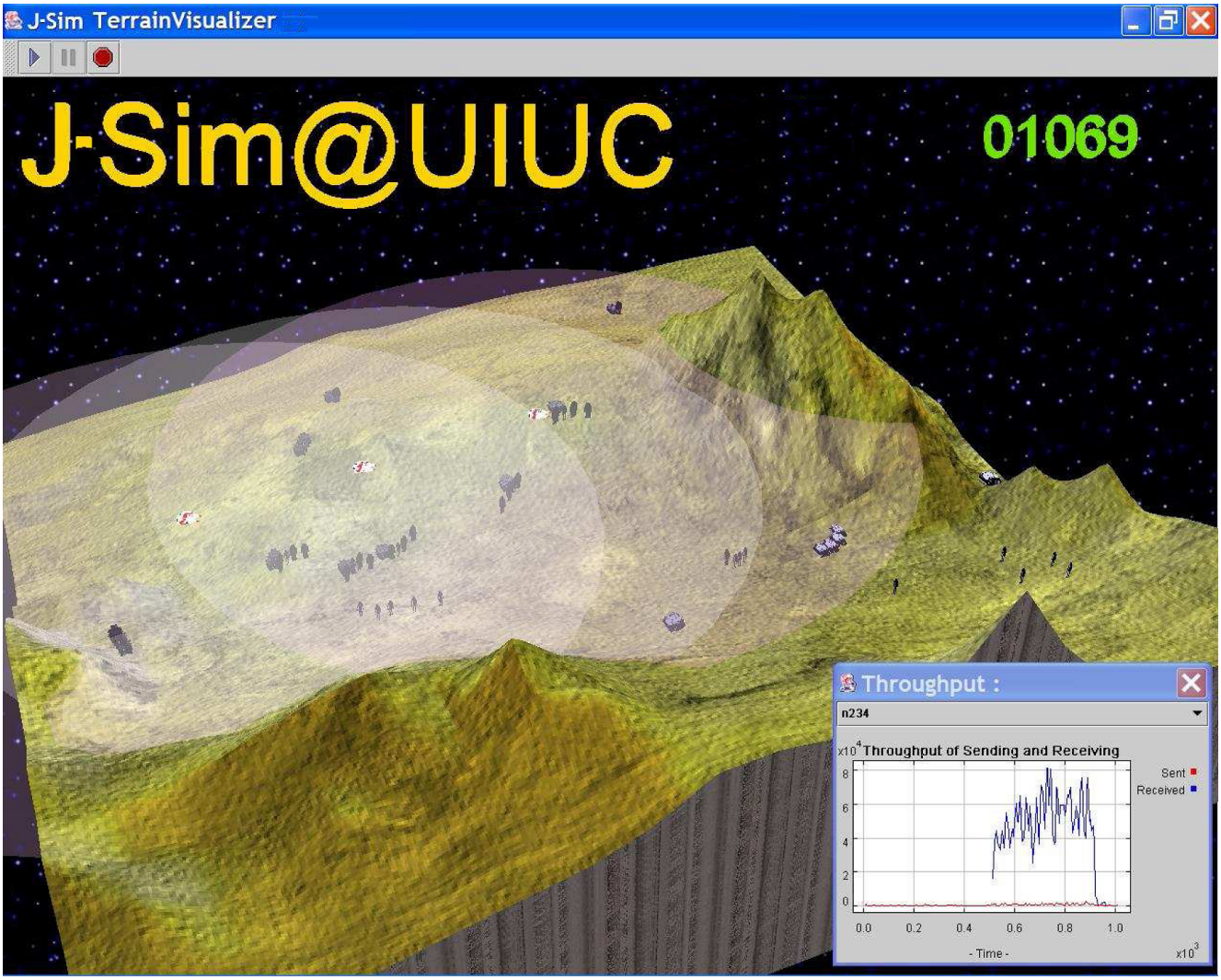


Fig. 14. A snapshot of the terrain and warefare entity visualization tool that shows a FCS scenario in San Diego, CA. Each circle denotes the coverage area of an UAV.

strength or bit errors). The transceiver pipeline stages calculate the received signal power in order to determine whether the signal can be received by the receiver.

Ns-2 [22] began as a variant of the REAL network simulator [41], and has evolved substantially over the past few years. It provides substantial support for simulation of TCP, routing, and multicast protocols. The support for wireless and mobile network simulation was included in ns-2 by the Rice University Monarch (Mobile Networking Architectures) Project [14][2]. The Monarch project provides various modules for (mobile) wireless network simulation; e.g., radio propagation models, the IEEE 802.11 MAC protocol, mobility models, different ad-hoc routing protocols (e.g., AODV and DSR) and Mobile IP. The latest version of ns-2 supports the simulation of pure wireless LANs, multihop ad-hoc networks and the combined simulation of wired and wireless (known as "wired-cum-wireless") networks. SensorSim [49], [48], [15], from UCLA, has further extended ns-2 by including the support for sensor network simulation. Similar to our simulation framework, SensorSim also includes the definition of target, sensor and sink nodes, sensor and wireless communication channels, physical media, mobility model and the power model. However, at the time of this writing, the public release of SensorSim is no longer available at [15].

Ptolemy [7] is an ongoing project at UC Berkeley that studies modeling, discrete-event simulation, and design of

[2]formerly known as The CMU Monarch Project (http://www.monarch.cs.cmu.edu/).

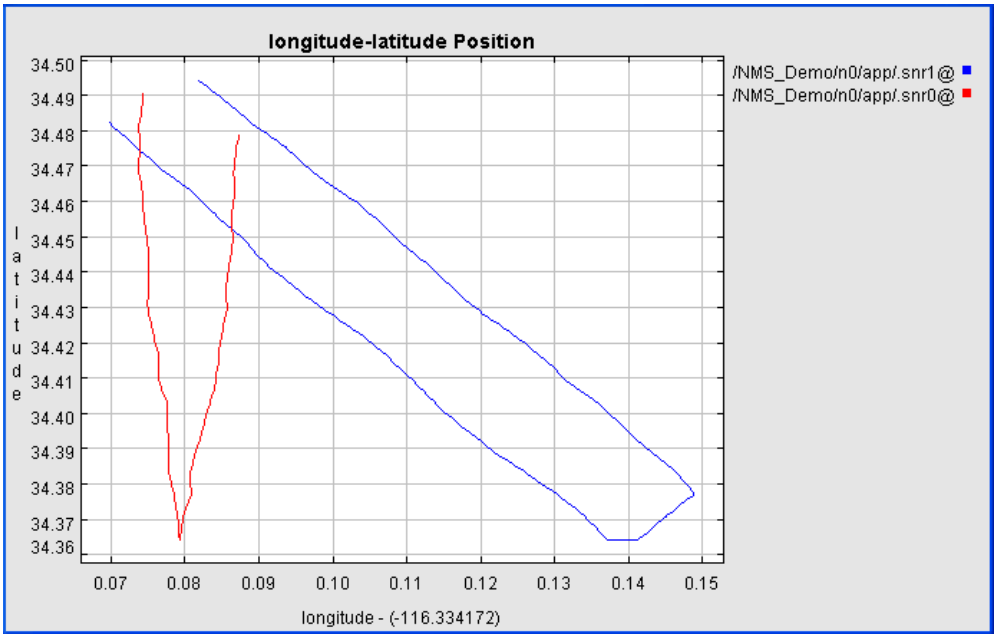


Fig. 15. Movement information of the two enemy tanks received by the sink node.

concurrent, real-time, embedded systems. The key underlying principle in Ptolemy is the ability to use multiple models of computation (e.g., continuous-time, dataflow, finite-state machine) in a hierarchical heterogeneous design environment. VisualSense [23] is a modeling and simulation framework that builds on and leverages Ptolemy to support design, simulation and visualization of sensor networks. In VisualSense, a sensor node can be modeled either in Java or by using conventional discrete-event models (e.g., block diagrams) or Ptolemy models (e.g., continuous-time or dataflow models). Similar to our simulation framework, VisualSense supports sensor and wireless channels, antenna gains, terrain models and battery models. However, Ptolemy does not support network emulation. On the other hand, *J-Sim* supports network emulation as explained in the Future Combat System (FCS) simulation presented in this paper.

TOSSIM [17] is a bit-level simulator for TinyOS[34] wireless sensor networks. The goal of TOSSIM is to provide an accurate, scalable simulator that bridge the gap between algorithms and implementations. TOSSIM can compile unchanged TinyOS applications directly into its framework, which means most of the codes written for TOSSIM can be directly used in TinyOS. It only replaces a few low-level TinyOS systems that touch hardware and thus maintains the accuracy of the simulation. The link layer is modeled using an independent bit error model. The packet is encoded with CRC and can correct one-bit error and detect two-bit error. TOSSIM models the wireless network with a directed graph, where each vertex is a node and each edge has a bit error probability. This simple abstraction increases its scalability. However, TOSSIM is very specific to TinyOS and Berkeley motes. It may not be appropriate for simulating a general sensor network.

IX. Conclusions and Future Work

In this paper, we have defined and implemented a simulation and emulation framework for wireless sensor networks (WSNs) in *J-Sim* — an open-source, component-based compositional network simulation environment. The framework provides an object-oriented definition of (i) target, sensor and sink nodes, (ii) sensor and wireless communication channels, and (iii) physical media such as seismic channels, mobility models and power models (both energy-producing and energy-consuming components). Customized application-specific models can be readily defined and implemented by sub-classing appropriate classes the simulation framework and customizing their behaviors. We have also included in *J-Sim* a set of classes and mechanisms to realize network emulation [40], [29], [60]. We leverage packet capturing tools (such as pcap [5] in Linux and Windows and SerialForwarder in TinyOS [34]) and devise mechanisms to synchronize the virtual time and the wall time.

We have demonstrated the use of the proposed WSN simulation framework by implementing several well-known localization, geographic routing, and directed diffusion protocols. We have also performed detailed performance comparisons

(in terms of execution time incurred, and the memory used) in simulating several typical WSN scenarios in *J-Sim* and *ns-2*. The simulation study indicates *J-Sim* and *ns-2* incur comparable execution time, but the memory allocated to carry out simulation (of length ≥ 1000 in *J-Sim* is at least two orders of magnitude lower than that in *ns-2*. As a result, while *ns-2* often suffers from out-of-memory exceptions and was unable to carry out large-scale WSN simulation, the proposed WSN framework in *J-Sim* exhibits good scalability. Finally, we have demonstrated the use of the WSN framework in carrying out real-life, full-fledged future combat system simulation and emulation.

We have also identified several research avenues for future work. First, the proposed framework did not explicitly model the lateration mechanism by which sensor nodes determine the location of a target node. As part of our future work, we will lay out classes to model how sensors cooperate to determine the location of a target node. Second, we will define and implement in *J-Sim* generic classes for several newly emerging WSN functions, such as coverage [61], [59], time indexing [62], and power management [63], [25].

REFERENCES

[1] Evaluation of J-Sim. http://www.j-sim.org/comparison.html.
[2] Future Combat System (FCS) simulation and demo. http://lion.cs.uiuc.edu/demo/demo.html.
[3] Irregular Terrain Model (ITM). http://elbert.its.bldrdoc.gov/itm.html.
[4] Java 3D API. http://java.sun.com/products/java-media/3D/.
[5] The libpcap packet capture library. ftp://ftp.ee.lbl.gov/libpcap.tar.z.
[6] Ns-2. http://www.isi.edu/nsnam/ns/.
[7] Ptolemy. http://ptolemy.eecs.berkeley.edu.
[8] GloMoSim. http://pcl.cs.ucla.edu/projects/glomosim/.
[9] J-Sim. http://www.j-sim.org/.
[10] J-Sim wireless extension tutorial. http://www.j-sim.org/v1.3/wireless/wireless_tutorial.htm.
[11] NGDC-GLOBE Project. http://www.ngdc.noaa.gov/seg/topo/globe.shtml.
[12] OPNET. http://www.opnet.com.
[13] PARSEC. http://pcl.cs.ucla.edu/projects/parsec/.
[14] The Rice University Monarch Project. http://www.monarch.cs.rice.edu/.
[15] SensorSim: A simulation framework for sensor networks. http://nesl.ee.ucla.edu/projects/sensorsim/.
[16] Terrain Integrated Rough Earth Model (TIREM). TIREM/SEM handbook, ECAC-handbook-93-076.
[17] TOSSIM: Accurate and scalable simulation of entire TinyOS applications, 2003.
[18] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, March 2002.
[19] G. Asada, M. Dong, T. S. Lin, F. Newberg, G. Pottie, and W. Kaiser. Wireless integrated network sensors: Low power systems on a chip. In *Proc. of the 1998 European Solid State Circuits Conference*, 1998.
[20] R. Bagrodia, R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: a parallel simulation environment for complex systems. *IEEE Computer Magazine*, 31(10):77–85, October 1998.
[21] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla. GloMoSim: A scalable network simulation environment. Technical Report 990027, Computer Science Department, University of California, Los Angeles, May 1999.
[22] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving simulation for network research. Technical Report 99-702, University of Southern California, 1999.
[23] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modeling of sensor nets in Ptolemy II. In *Proc. of the ACM/IEEE International Symposium on Information Processing in Sensor Networks (ACM/IEEE IPSN'04)*, April 2004.
[24] N. Bulusu, J. Heidemann, and D. Estrin. GPS-less low-cost outdoor localization for very small devices. *IEEE Personal Communications*, pages 28–34, 2000.
[25] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *Proc. of MobiCom*, July 2001.
[26] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards realistic million-node Internet simulations. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.
[27] L. Doherty, K. Pister, and L. Ghaoui. Convex position estimation in wireless sensor networks. In *Proceedings of IEEE INFOCOM*, 2001.
[28] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proc. of the ACM International Conference on Mobile Computing and Networking (ACM MobiCom'99)*, August 1999.
[29] K. Fall. Network emulation in the vint/ns simulator. In *Proceedings IEEE ISCC99*, July 1999.

[30] T. He, C. Huang, B. M. Blum, J. A. Stankovic, and T. Abdelzaher. Range-free localization schemes for large scale sensor networks. In *Proceedings of ACM MOBICOM*, 2003.
[31] J. Heidemann, F. Silva, and D. Estrin. Matching data dissemination algorithms to application requirements. In *Proc. of the ACM Conference on Embedded Networked Sensor Systems (ACM SenSys'03)*, 2003.
[32] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. 5th ACM MobiCom*, August 1999.
[33] A. Higgins, G. Lehtola, R. Meyer, and K. Peterson. A quasi-optical adaptive antenna communications system for 37 GHz. In *Proc. of the IEEE Military Communications Conference (IEEE MILCOM'01)*, October 2001.
[34] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. S. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Boston, MA, USA, Nov. 2000.
[35] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. of the ACM International Conference on Mobile Computing and Networking (ACM MobiCom'00)*, 2000.
[36] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, February 2003.
[37] J. Kahn, R. Katz, and K. Pister. Next century challenges: Mobile networking for "Smart Dust. In *Proc. of the ACM International Conference on Mobile Computing and Networking (ACM MobiCom'99)*, 1999.
[38] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for small dusts. In *Proc. of the ACM International Conference on Mobile Computing and Networking (ACM MobiCom'99)*, August 1999.
[39] B. Karp and H. Kung. Greedy perimeter stateless routing for wireless networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 243–254, Boston, MA, August 2000.
[40] Q. Ke, D. Maltz, and D. B. Johnson. Emulation of multi-hop wireless ad hoc networks. In *Proc. of the Seventh International Workshop on Mobile Multimedia Communications*, October 2000.
[41] S. Keshav. REAL: A network simulator. Technical Report 88/472, University of California, Berkeley, 1988.
[42] W. Kishaba, G. Vardakas, J. J. Garcia-Luna-Aceves, L. Bao, and Y. Kang. Adhoc networking with beam-forming antennas. In *Proc. of the IEEE Military Communications Conference (IEEE MILCOM'01)*, October 2001.
[43] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric ad-hoc routing: of theory and practice. In *Proc. of the ACM Symposium on Principles of Distributed Computing (ACM PODC'03)*, July 2003.
[44] F. Kuhn, R. Wattenhofer, and A. Zollinger. Worst-case optimal and average-case efficient geometric ad-hoc routing. In *Proc. of the ACM International Conference on Mobile Computing and Networking (ACM MobiHoc'03)*, June 2003.
[45] N. Li and J. C. Hou. Improving connectivity of wireless ad-hoc networks. Submitted to IEEE Int'l Conf. on Distributed Computing Systems (ICDCS'05), October.
[46] R. Nagpal, H. Shrobe, and J. Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *Proceedings of IPSN*, 2003.
[47] D. Niculescu and B. Nath. Ad hoc positioning system (APS). In *Proceedings of IEEE GLOBECOM*, 2001.
[48] S. Park, A. Savvides, and M. Srivastava. Simulating networks of wireless sensors. In *Proc. of the 2001 Winter Simulation Conference*, 2000.
[49] S. Park, A. Savvides, and M. Srivastava. SensorSim: A simulation framework for sensor networks. In *Proc. of the ACM international Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2000.
[50] S. Park, A. Savvides, and M. Srivastava. Battery capacity measurement and analysis using lithium coin cell battery. In *Proc. of the ACM International Symposium on Low power Electronics and Design (ACM ISLPED'01)*, 2001.
[51] C. Perkins, E. Royer, and S. Das. Ad hoc on demand distance vector (aodv) routing. IETF Draft, January 2002.
[52] K. Sarabandi, I. Koh, G. Liang, and H. Bertoni. Propagation modeling for FCS. In *Proc. of the IEEE Military Communications Conference (IEEE MILCOM'01)*, October 2001.
[53] P. Sass and J. Freebersyser. FCS communications technology for the objective force.
[54] A. Savvides, C. C. Han, and M. B. Srivastava. Dynamic fine-grained localization in ad-hoc wireless sensor networks. In *Proceedings of ACM MOBICOM*, 2001.
[55] D. J. Schwartz, Y. Yemini, and D. Bacon. NEST: A network simulation and prototyping testbed. *Communications of the ACM*, 33(10):63–74, October 1990.
[56] A. Sobeih and J. C. Hou. A simulation framework for sensor networks in J-Sim. Technical Report UIUCDCS-R-2003-2386, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2003.
[57] H.-Y. Tyan. *Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation*. PhD thesis, Department of Electrical Engineering, The Ohio State University, 2002.
[58] H.-Y. Tyan and J. C. Hou. *JavaSim*: A component-based compositional network simulation environment. In *Proc. of Western*

Simulation Multiconference, CNDS'01, January 2001.

[59] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. Gill. Integrated coverage and connectivity configuration in wireless sensor networks. In *ACM Sensys'03*, Nov. 2003.

[60] B. White, J. Lepreau, and S. Guruprasad. Lowering the barrier to wireless and mobile experimentation. *ACM SIGCOMM Computer Communications Review*, 33(1), January 2003.

[61] F. Ye, G. Zhong, S. Lu, and L. Zhang. Peas: A robust energy conserving protocol for long-lived sensor networks. In *The 23nd International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[62] R. Zheng, G. He, I. Gupta, J. Hou, and L. Sha. Time indexing in sensor networks. In *Proc. 1st IEEE International Conf. on Mobile Ad Hoc and Sensor Systems (MASS'04)*, October 2004.

[63] R. Zheng and R. Kravets. On-demand power management for ad hoc network. In *Proc. of IEEE INFOCOM*, 2003.

I. Wired networks

Traffic model	Packet scheduler	Routing algorithm/protocol	Resource reservation
Periodic message	FCFS	Unicast shortest path	RSVP
Peak rate model	RM	Unicast minimum load path	
Leaky bucket model	EDF Unicast QoS routing		
Token bucket model	stop-and-go	Multicast shortest path	
IETF/Intserv Flowspec	DCTS	Multicast minimum load tree	
(r, t)-smooth model	*VirtualClock*	Multicast spanning tree	
(C, D)-smooth model	LFVC	Multicast steiner tree	
	SCFQ	Distance vector routing	
	PGPS	OSPFv2	
	STFQ	QoS-enhanced OSPFv2	
	WF^2Q	Dist. Unicast QoS routing	
	Leave-in-time	DVMRP	
	FIFO-r	MOSPF	
		CBT	
		QoS-enhanced CBT	

Data transport	Socket layer	Tagger/marker	Buffer management
TCP-Reno	Java-compliant	Token bucket model	RED
TCP-Tahoe		TSW	FRED
TCP-Vegas		ETSW	SRED
TCP with delayed ack option			BRED
TCP with ECN			
UDP			

PGPS: packet-by-packet generalized processing sharing.
STFQ: start time fair queuing.
LFVC: leap forward virtual clock.
DVMRP: distance vector multicast routing protocol.
CBT: core based tree protocol.
ECN: explicit congestion notification.
TSW: time sliding window.
RED: random early drop.
SRED: stable random early drop.
WF^2Q: worst-case fair weighted fair queuing.
SCFQ: self-clocked fair queuing.
OSPFv2: open shortest path first protocol (Version 2).
MOSPF: multicast extension to open shortest path first.
TCP: transport control protocol.
UDP: user datagram protocol.
ETSW: enhanced time sliding window.
FRED: fair random early drop.
BRED: balanced random early drop.

II. Wireless networks

Antenna model	Signal propagation	Datalink model	Ad hoc routing
Parabolic reflective antenna	Free space model	IEEE 802.11	DSR
Directional antenna	Two-ray ground model	IEEE 802.11 with PSM	AODV
	Irregular terrain model	Military link emulation with different sending powers, frequencies, bandwidth, receiving/ carrier sense/ interference thresholds	AODV multicast extension

Satellite link	Mobility model
Satellite link with detailed wireless error model and propagation delay	Random way-point
	SDF trace-based model

DSR: dynamic source routing.
PSM: power saving mode.
AODV: ad hoc on-demand distance vector.

TABLE I

NETWORK AND COMMUNICATIONS ENTITIES SUPPORTED IN *J-Sim*.