DiSenS: Scalable Distributed Sensor Network Simulation

UCSB Computer Science Technical Report Number CS2005-30

Ye Wen, Rich Wolski, and Gregory Moore
Department of Computer Science
University of California, Santa Barbara, 93106
Email: {wenye,rich}@cs.ucsb.edu, gmoore@umail.ucsb.edu

***Abstract*— Simulation can be a very useful tool for developing, evaluating and analyzing sensor network applications, especially when deploying a large scale sensor network remains an expensive and labor intensive endeavor. However, the difficulty of achieving both fidelity and scalability has limited its use as a design and analysis tool.**

In this paper, we describe DiSenS (DIstributed SENsor network Simulation) – a complete scalable and extensible distributed simulation system for sensor networks. DiSenS provides a cycle-accurate device emulator that is extendable by various fidelity-enhancing models (radio, power, etc.) for tunable simulation accuracy. A key distinguishing feature of DiSenS is that it is implemented for distributed-memory parallel cluster systems. To achieve efficiency in this computational setting we have designed a simple and efficient distributed synchronization protocol and coupled it with a sophisticated node partitioning algorithm (not of our design) to achieve new scalability and performance levels.

I. Introduction

Sensor networks make possible the instrumentation and actuation of potentially a large variety of environmental phenomena. By making the provisioning of computing power non-invasive and inexpensive, the ability to apply computation as a way of analyzing "the world" ubiquitously becomes a possibility, the potential impact of which cannot be overstated.

However, despite the potential for transformative scientific and even social change that sensor networks seem to promise, their development is, at present, still nascent. While various technological and economic obstacles exist, a key impediment to their development is the lack of a scalable simulation capability that provides the fidelity necessary to support both coherent design and efficient engineering of sensor network systems. State-of-the-art sensor network research and development relies on labor and resource intensive trial-and-error using physical devices and *in situ* deployments. Few other systems of similar complexity and potential expense (e.g. computational processors, embedded systems, network architectures, etc.) are investigated and engineered in the same way: without high-quality and multi-fidelity simulation support.

To accelerate possible research and development advances for sensor networks, our work focuses on the development of a sensor network simulation capability in the form of DiSenS – a distributed software infrastructure for scalable sensor network simulation. DiSenS is intended to serve as a research tool for the development of simulation models targeting different fidelity levels, and to allow these investigations to take place at scales unattained by previous systems.

There are two general approaches to sensor network simulation that have been explored previously. Discrete-event systems such as those described in [16], [26], [40] model device functionality and communication as a set of partially ordered events modifying distributed state. Often, these systems have focused on communication interactions (which takes place via unreliable and difficult-to-model communication radios) and only roughly approximate the behavior of the constituent devices themselves. By sacrificing device fidelity, discrete event simulators can achieve very high performance and have good scalability.

Full-system simulators [33], [38], [43], [27] take an alternative approach. They simulate the internal device functionality in detail and allow ensemble behavior to emerge from the interactions of independent-but-communicating simulated devices. These systems achieve good fidelity levels, but the need to coordinate multiple simulated devices has limited their scalability.

Our work attempts to extract and combine the benefits of both approaches. DiSenS supports high-fidelity and high-performance emulation of individual sensor devices as well as a basic radio communication simulation that scales. The infrastructure is designed to use dedicated clusters of commodity "PC" class machines interconnected by high-performance local-area networking technology (heretofore termed "cluster computing" technology). Because the processing and network elements of clusters are so much more powerful than the corresponding device and radio technologies for sensor networks (without the power constraints) it should be possible to achieve useful simulation performance levels.

In addition, DiSenS admits "pluggable" augmenting models for power consumption and radio behavior. Indeed, our original motivation for developing DiSenS has been to provide a simulation capability for our own use in developing high-quality statistical characterization of radio communication behavior. The infrastructure is thus parameterizable by models which characterize different forms of component performance response. This flexibility is intended to allow users of DiSenS

to experiment with the tradeoff between simulation fidelity and simulation performance explicitly.

To support sensor network design system architecting activities, we have taken the need for *simulation transparency* as a primary design goal in developing DiSenS. That is, sensor network software (including the operating system) should be able to run unmodified on both a DiSenS virtual sensor network and a corresponding physical (actual) network of sensor devices. That is, the simulations should be *complete* enough so that the software cannot "see" the difference between executing on a real sensor network or a DiSenS virtual simulation of a sensor network.

Thus our ultimate goal in developing DiSenS is to build a simulation framework that permits exploration of fidelity, completeness, scalability, and bridging, as outlined in [16]. We report on the degree to which we currently achieve this goal with DiSenS using both benchmarks designed to exercise various component features, and publically available sensor network operating system and application code that we treat as inviolate. In so doing, we believe that this work makes the following research contributions.

- We describe the distributed implementation methodology we have chosen for DiSenS, with a particular emphasis on the protocol we use to synchronize the emulated device clocks, and the partitioning strategy for mapping simulation components to cluster processors.
- We report on the fidelity that our full-system device emulations are able to achieve.
- We describe and demonstrate how different "plug-in" models can be incorporated by the infrastructure.
- We provide a detailed exposition and analysis of DiSenS's efficacy in terms of simulation performance, completeness, and scalability.

As a whole, we believe these contributions extend the state-of-the-art in distributed sensor network simulation. We do not, however, claim to similarly extend the state-of-the-art in other important research areas such as power modeling or radio communication modeling. Rather, our intention is to show how previous results in these areas can be applied with greater scale and efficiency than has been previously reported.

In rest of paper, Section II discuss the device simulation framework, including the hardware emulation core and the pluggable models. Section III studies the synchronization problem and presents DiSenS distributed simulation algorithms. We present measurements and analysis of DiSenS functionality in Section IV, survey the related work in Section V and finally conclude in Section VI.

II. Parameterizable Device Simulation Framework

We first introduce the device simulation framework, which faithfully emulates the sensor device hardware and lays the foundation for distributed simulation. The core of the framework is a cycle-accurate hardware emulator. It is parameterized by a set of pluggable fidelity enhancing models, e.g. radio model, power model, etc., to allow experimentation with different fidelity levels and modes of investigation.

A. Cycle-Accurate Hardware Emulator

As the basis for accurate simulations, we have stressed the development of simulation tools that achieve timing accuracy. While accurate power and radio simulation techniques are the subject of much current research activity, we believe that successful approaches will depend, ultimately, on the ability to simulate device cycle timings correctly.

At the core of our device simulation framework is a hardware emulator with extensive support for various popular sensor network devices. In the current implementation, we emulate the Mote [23] devices (the Mica2 and MicaZ platforms), Stargate devices [39], and iPAQ devices [13] and we are adding the support of other devices, like Telos [42]. Thus, the system is capable of heterogeneous sensor network simulations. In this work, however, we focus only on simulating ensembles of Mica2 and MicaZ devices exclusively.

The emulation core supports the following sensor node functionality and components by emulating

- the AVR instruction set,
- the ATmega128L microcontroller, including most on-chip functions: program memory, RAM, EEPROM, timers, serial devices (UARTs), SPI (Serial Peripheral Interface), ADC (Analog/Digital Converter), Watch Dog Timer and fuse bit setting (for boot loader and self programming),
- the 512KB on-board flash,
- the Serial ID chip,
- the CC1000 (Mica2) and CC2420 (MicaZ) radio chips,
- the LEDs and the sensor boards.

The heart of hardware emulator is a cycle-accurate AVR instruction interpreter. Hardware emulation is a mature area yielding several good technologies for simulating one architecture on another with high efficiency [33], [5]. However, we choose to use a fairly simple switch-based interpreter, that is similar to SimpleScalar [1]. The biggest reason is for portability. Since we intend to implement simulations using collections of machines, the ability to run on a broad range of architectures is essential. Moreover, the relatively simple nature of the AVR architecture and the high clock speeds available from commodity powered workstations makes it possible to achieve faster-than-real time emulations of many sensor devices. For example, our system is able to emulate Motes using a 3.2GHz x86 processor at approximately 9 times real-time speed.

The interpreter emulates each instruction, changes the state of microcontroller and drives an internal clock cycle by cycle, which in turn fires the asynchronous events in an event queue, generated by hardware components like timers, USARTs and ADCs. The collection of emulated devices is rich enough and accurate enough to boot and execute unmodified TinyOS [11] binaries. Thus applications and operating systems cannot distinguish execution on the emulator from execution on the actual hardware.

B. Pluggable Models

Our device simulation framework provides a set of common interfaces for integrating the core hardware emulator to various extensions for power and communication. Our intention is to provide a platform for experimentation with different "plug-in" models, both to support the development of new models as well as to provide a way to trade simulation speed for fidelity using a suite of models. Thus researchers using the system can plug in the best models according to their simulation needs. In this work, we demonstrate this flexibility using a small set of candidate models from the literature. We make no claims regarding the relative value of these models. Rather, we attempt to show how they can be incorporated into the framework and can subsequently be used in scalable device simulations.

1) Radio Models: The system includes a "simple" or "ideal" radio model in which radio packets are sent losslessly to all the neighbor nodes within its radio range. While the ideal model is typically highly inaccurate, it is often used for initial code development and debugging as well as to achieve an upper bound on potential performance. Under this model, each sensor node buffers the packets sent to it even if it is not in receiving mode. Packets are time stamped and when a sensor node receives, it checks the packet buffer and reads the packets that match its current clock time. In addition, packets from different nodes may conflict with each other. When conflicting transmissions interfere, the ideal model performs a bit-wise *OR* of the bits received during the conflict period. As a result, this basic radio model is able to simulate transmission conflicts and thus the "hidden terminal" effect [44]. Also, packet loss due to the partial reception of packet preamble (because of the mis-synchronization of packet receiving and packet transmitting) is naturally modelled as part of the radio chip emulation logic.

The ideal model can be made more realistic through the addition of channel loss models. There are different ways to model the channel loss. Analytical techniques use a mathematical description of a physical electromagnetic radiation propagation. Thus, loss or signal perturbation is based on the "physics" of the intervening communication medium. There are is large body of literature on such physical models [30]. Despite their accuracy, however, their complexity and potential computational expense make them difficult to use in sensor network simulations.

A more popular approach is based a statistical description of channel loss, often derived from measurement trace data [6], [49], [48]. In this approach, a large set of radio transmission data is collected using different parameters. The trace data is then "mined" using statistical methods to derive distributional descriptions of characteristics such as reception rate. Cerpa et al. [6] explored this approach and achieved some noteworthy results. They have also proposed methods of generating realistic network instances based on the discovered feature distribution. In our work, we have developed a plug in that uses a loss rate distribution generated from our own measurement trace data using a similar methodology as in [6]. Thus, using the basic model and the trace-derived loss model, our system can incorporate both deterministic models based on mechanism and statistical models based on off-line analysis of trace data.

2) Power and Battery Models: At present, perhaps the most active area of sensor network simulation research focuses on modeling power dissipation. Sensor network simulators are required to provide accurate energy consumption estimation for any reasonable study based on simulation. A number of power models for sensor network devices have been proposed and investigated in the literature [15], [36], [35]. These models are typically based on the measurements obtained by using benchmarks to exercise the sensor device in various modes yielding different levels of fidelity. In this work, we incorporate one such model [15] in our simulator.

We also provide a simple linear battery model. Several battery models have been proposed in the literature[41], [4], [31]. Linear model is the simplest, again representing the ideal case in debugging and "back-of-the-envelope" settings. Moreover, in fast, lower-fidelity simulations of "steady-state" a linear model is often preferred [17] since the middle of the discharge curve is often close to linear.

C. Debugging and Profiling Facility

As mentioned previously, one of the main motivations for scalable sensor network simulation is debugging. While many programming errors only manifest themselves *in situ*, often more systematic problems can be most effectively exposed in simulation. Moreover, a critical advantage offered by software device emulation (the method of simulation we have chosen) is that the software emulator can be more easily and completely instrumented than the actual hardware.

To support these activities, we have developed two complementary functionalities for the simulator: a distributed debugger and a virtualized instrumentation package. The debugger is able to connect to any sensor node in our distributed simulator and debug the node on the fly. The debugger can also broadcast commands to multiple nodes for coordinated debugging.

However, our goal in this work has been to develop a simulation environment that is completely transparent to the application and operating system under study. As such, we have not yet explored solutions that would allow the programmer to embed debugging functionality in the simulation version that would then need to be removed or altered when the application is executed in native mode. Indeed, we believe it is important to be able to execute and debug the same binaries without modification in both environments.

This transparency goal limits the debugging options that are available, however, since the actual sensor devices have fairly restricted I/O capabilities. Typically, sensors can only display internal status through a set of LEDs in a way that is tedious and error-prone to decode. That is, there is no easy way to include print statements (still a debugging staple) in a typical sensor application. Moreover, because the emulator works at the hardware level, in cannot "see" into the application or operating system code that is using it. We

TABLE I

VIRTUAL REGISTERS FOR DEBUGGING AND PROFILING

Virtual Register Address	Name	Functionality
0x75	VDBCMD	Command Register
0x76	VDBIN	Input Register
0x77	VDBOUT	Output Register

have included typical machine-level debugging functionality (e.g. break-points, memory interrogation, etc.) but without intimate knowledge of the emulator it would be difficult for a programmer to associate the state of the emulated machine with high-level program behavior. We approach this problem by introducing virtual debugging and profiling hardware into the simulated device in a way that allows unmodified execution of the instrumented code on the native sensor hardware.

1) Virtual Hardware Based Debugging and Profiling: We introduce three virtual registers that are allocated in the reserved I/O register space of ATmega128L microcontroller. Table I shows the functionalities of these registers.

The *VDBCMD* register is used to issue debugging and profiling commands. The *VDBIN* is for reading data from simulator and the *VDBOUT* is for output. These virtual registers comprise a "channel" that sensor programs can use to communicate with simulator and/or the simulator user which, while it is not available on the native devices, is none the less transparent. For example, it is possible to print debugging information through the virtual registers by first sending an ASCII character output command through *VDBCMD* register and then continosly writing debugging text (a character at a time) to the *VDBOUT* register. The simulator is instructed to interrogate *VDBCMD* and, when it is loaded with the command indicating ASCII output, to direct the output stream appearing in *VDBOUT* to the simulator console. Through the virtual registers, it is also possible to access simulator's status in sensor program. Thus, the availability of these additional virtual registers provides developers with more powerful debugging capabilities than the simple breakpoint and memory interrogation facilities discussed previously.

To provide transparency, all the functionalities of virtual hardware are coded into high level access functions provided as part of the AVR glibc library. Since the virtual registers are located in reserved I/O space and read/write operations performed on them has no effect on a real device, instrumented programs can also be run directly on the real hardware with only slight performance loss. In this way, a program with debugging instrumentation compiled into it can be moved between the simulator and the actual hardware without change, thereby speeding the debug and test cycle.

III. DISTRIBUTED SIMULATION

One of the primary motivations for the development of our system is the ability to simulate "large" ensembles of sensors so that potential problems of scale can be studied. While previous work [43], [16], [25], [2] has addressed the issue of scalability using different approaches (cf. Section V for a review of related work), our goal is to support binary transparency with respect to the applications and operating system (similar to Avrora [43]) in a way that maximizes the size of the ensemble that can be simulated.

There are two measures of scalability DiSenS attempts to maximize. The first, analogous to the standard notion of speedup used to characterize parallel programs, is to maximize the ratio of wall-clock time that elapses for a complete sensor network simulation on a single processor to that for the same simulation running on multiple processors. This ratio characterizes the benefit of parallelism in terms of reduced execution time for a given simulation.

In addition, we also consider speedup (or more probably slowdown) in terms of the clock periods of the sensor network devices under study. By calculating the number of device clock cycles that have been simulated in a given wall-clock period, we can compute the speedup or slowdown of the simulation relative to the clock cycles that the real device experiences in real time.

Notice that these two notions of speedup are related but distinct. For example, it is possible for our system to achieve excellent speedup using the first measure (the parallel time is much faster than the sequential time) but poor speedup or even large slowdown using the second measure (devices are simulated only a small fraction of their real time speeds). While we have designed DiSenS to attempt to optimize both measures, we focus on the latter measure – the relationship to real time device speed – in this work as we believe it is the more challenging of the two.

Clearly, the degree to which these measures can be optimized depends on both the structure, constituent devices, and topology of the ensemble simulation and the characteristics of the computational resources. For the latter, we believe a distributed memory cluster computing environment has the largest potential. However, the typically close coupling of simulation systems makes distributed implementation challenging.

A. Background and Approach

Our approach is to simulate ensembles of sensor devices by executing individual cycle-accurate device simulations which communicate via simulated radios. Notice that this approach is distinct from an event-driven methodology in that we do not decompose the collection of simulations into explicit events that must then be time ordered. Rather, we use individual device emulations and a simulated radio communication environment as a virtual deployment of a complete sensor network, and run the same operating system and applications on the virtual sensor network as if they were running on an actual deployment[1]. To coordinate between individual device emulations, when radio communication occurs, the two communicating sensor devices must be synchronized with respect to their relative internal clocks.

[1]Our style of simulation might more properly be termed an "emulation" as a way of emphasizing the distinction between our approach and an event-driven one. Because the radio environment is purely simulated, however, we have chosen to term our approach as a "simulation" since we believe that term to be more general.

Previous work that takes a similar approach includes ATEMU [27] and Avrora [43]. ATEMU [27] is a cycle-accurate sensor network simulator. It maintains a global clock and emulates one instruction a time for each simulated device. In this way, the sensor nodes are automatically synchronized and no extra facility is necessary to maintain the correct order of radio events. However, ATEMU is limited to a single process and can not scale to larger systems. Avrora [43] extends the simulation to a multi-threaded shared memory system. It scales on multi-processor machines. In Avrora, each device is simulated in a separate thread. Avrora loosens ATEMU's cycle-to-cycle synchronization requirement by extending the synchronization period to the length of a byte transfer time – 3072 ATmega128L cycles – since packets are always transmitted in byte unit. A thread barrier is used to achieve its lock-step style synchronization, which stops all the threads periodically to ensure every radio byte will be correctly received during the correct time period.

In a clustering computing environment, relatively large and variable network latencies make direct extensions of these two approaches difficult. Cluster network latency is measured in milli-seconds while a desktop PC can easily emulate one device instruction in the 0.1 micro-second range. If lock-step global synchronization is used, the simulated clock speed will be determined by the all-to-all network communication latency.

B. Synchronizing Ensemble Emulations

Our approach to synchronizing multiple device emulations relies on an abstraction of the radio communication protocol. To illuminate the nature of this abstraction, we begin by discussing sensor network radio behavior in some detail.

Currently two types of radio chips are emulated in our device emulator, the CC1000 chip and CC2420 Zigbee radio chip [7], both manufactured by Chipcon. The CC1000 is the radio chip used by the Mica2 senor mote [21] and the CC2420 is used in the more recent MicaZ [22] platform. The CC1000 is a rather simple radio chip. It has two working modes: transmitting and receiving (ignoring power saving features of the chip for the moment). In transmitting mode, data bits are pumped in from the SPI line, modulated, and emitted through the antenna. In receiving mode, the radio signal is amplified, demodulated and converted into digital bits which will be assembled into radio packets by software protocol stack. The mode transition is controlled by the software. CC1000 also has a receive signal strength indication (RSSI) measurement function. This analog value of the signal strength is output via a chip pin, and converted into digital value by the ADC module of the microcontroller. The RSSI value is used by the software MAC layer to perform collision detection.

CC2420 is a more advanced radio chip that implements the low level function of Zigbee (IEEE 802.15.4) standard. The major difference between CC1000 and CC2420 from the simulation point of view is that CC2420 performs the packet assembly in the chip and has a much faster transmission speed. CC2420 has a similar signal sampling function and also measures RSSI value. However CC2420 uses a pin called CCA (clear channel access) to indicate whether the radio channel is clear based on a preset threshold. This provides a simpler interface for MAC layer collision detection.

The typical radio activity paradigm of TinyOS sensor applications can be described as follows. Normally, the radio stays in receiving mode (it may be turned off for power saving). When a preamble of a packet is recognized, the complete packet payload is to be assembled and uploaded to the application. When a packet needs to be sent, the MAC layer checks the channel using RSSI value or CCA value. If the channel is busy, it backs off for a random period of time and tries again. Otherwise, the radio chip is switched into transmitting mode and a complete packet is sent out.

Thus, packet receiving and signal sampling are very similar operations: they both read a value from the channel. The only difference is the length of time they use to access the channel. As a result, radio communication behavior can be abstracted into two operations: *read_channel* and *write_channel*. The *read_channel* represents the packet receiving and the signal sampling. The *write_channel* represents the packet transmitting.

As discussed previously, global clock is not feasible in a distributed environment since every clock access needs to traverse the network thereby incurring a large overhead. Instead, we use a peer-to-peer design in which each sensor node maintains its own local clock, clocks are synchronized before message rendezvous, and each node is otherwise simulated independently.

When a communication between nodes occurs, the causal relationship that exists between sender and receiver is rectified at the receiver so that packets are received in order, and that local clock values roughly correspond to arrival timings. We formalize this synchronization problem in abstract terms and then discuss our proposed solution.

We first define the simulation:

Definition 1 (Simulation): If we define a radio node N_i as a tuple $(clock_i, read_channel, write_channel)$, where $clock_i$ is the internal clock of node N_i, $read_channel$ and $write_channel$ are the only two operations performed on a shared resource, C, representing the channel, we can define a *simulation* S as a set: $(N_0, N_1, ..., N_k, C)$.
We have to distinguish the *simulation time* and *simulated time*. The former is the wall clock time in real world that is used to measure the simulation. The latter is the virtual clock time in simulated world that is shared by simulated Motes. Then we define the correctness of a simulation:

Definition 2 (Correctness): A *simulation* is correct if the following relationship is ensured: $\forall$ *simulated time* period $[vt_{i_1}, vt_{i_2}]$ (corresponding *simulation time* period $[rt_{i_1}, rt_{i_2}]$), at which node N_i is scheduled to $write_channel(C)$, and its neighbor node N_j is to $read_channel(C)$ during $[vt_{j_1}, vt_{j_2}]$ (*simulation time* $[rt_{j_1}, rt_{j_2}]$); if $[vt_{i_1}, vt_{i_2}] \cap [vt_{j_1}, vt_{j_2}] \neq \emptyset$, $[rt_{i_1}, rt_{i_2}] \cap [rt_{j_1}, rt_{j_2}] \neq \emptyset$.
Intuitively, a *correct* simulation requires any receiver to receive any data that it is meant to receive according to the causality

in simulated time space. In our simulation structure, given that sent data is transferred in byte unit and buffered at the receiver side, correct simulation can be achieved if each receiving node delays the delivery of each message byte until the local clock on the receiver is past the local clock on the sender. Conservatively,

Property 1 (Safe Receive): if whenever a node N_i invokes operation $read_channel$, it waits until synchronized with its neighbors, which means $\forall k$, if N_k and N_i are neighbors, $clock_k >= clock_i$, the simulation S is *correct*.
Note that we have to be conservative by waiting all the neighbors since we can not predict which neighbor will transmit at the time when we receive. We term this property the *safeness property.*

C. Distributed Synchronization Protocol

Based on the safeness property we design the complete synchronization protocol for distributed simulation. We first introduce a primitive, *wait_on_sync*.

Definition 3 (wait_on_sync): *wait_on_sync* is a primitive operation. If it is called by a node N_i, it waits until $\forall k$, N_k is a neighbor of N_i, $clock_k >= clock_i$.
wait_on_sync has to be called every time the radio channel is accessed (receiving or sampling).

Since *wait_on_sync* relies on the clock information of neighboring nodes, each node has to be informed of its neighbors' local clock value. We use a simple clock update protocol in which each node broadcasts its local clock time periodically. The length of update interval does not affect correctness but does have effect on performance. There are two special requirements on when to send updates. First, clock updates can not be sent during the transmission of a byte. This is because if it is sent, a neighbor waiting on a receive will believe it is time to proceed (if it does not wait for others) and will miss a partial byte. Updates, then, can only be sent between bytes during a transmission. Second, before a node starts to wait by calling *wait_on_sync*, it must first send an update. Without notifying its neighbors of it intention to wait, a node's silent wait will cause a deadlock if some other nodes are going to wait for it.

In summary, any receiver *wait_on_sync*s to block and wait for neighbors' clock updates before it receives a message or samples the radio medium. Before blocking, however, it must reliably inform its neighbors of its local clock value to prevent deadlock.

Using the above synchronization protocol, we implement our distributed simulation system. Given a set of nodes, we first partition them into groups. Each group is simulated on one machine and each node is simulated in one thread. In each group, a clock table is maintained to keep the updated clock time for all local nodes and their neighbor nodes. Whenever an clock update is sent, it first updates the local neighbors and then multicasts to the remote neighbors if it has. Our synchronization protocol treats the local and remote synchronization in the same way. The following pseudo code demonstrates the synchronization algorithm of a sensor node.

```
do_for_every_byte_transfer_time() {
    switch (mode) {
    case RECEIVING:
        send my clock update;
        wait_on_sync();
        retrieve data byte from packet buffer;
        break;
    case TRANSMITTING:
        send my clock update and data byte;
        break;
    default:
        send my clock update;
        break;
    }
}
```

The above code doesn't show the algorithm for signal sampling operation, which is the same as receiving (the "RECEIVING" section in *switch* statement). The code shows that we send at least one clock update for every byte transfer time regardless of radio modes. For transmission, data byte is "piggy-backed" on the clock update messages to reduce the message traffic. Notice that there is no constraint for senders. Senders send data bytes at any time they want. The sent data bytes are buffered at receivers' side. And it is receiver's responsibility to ensure the correct reception of radio packets. Notice also that there is a great deal of overhead in this protocol. If this overhead cannot be amortized or ameliorated by the performance of the network interconnect within the cluster, the overall performance of the ensemble simulation will be low. Our results seem to indicate that these issues are addressable, however.

Here is the code for *wait_on_sync*:

```
wait_on_sync(nodei) {
    for (all nodej as a neighbor of nodei) {
        if (nodej's time < nodei's time) {
            put nodei on nodej's waiting list;
        }
    }
    if (nodei waits on any node)
        wait();
}
```

The following code shows what happens when a clock update is received, regardless locally or remotely.

```
update_clock(nodei, clock) {
    nodei.clock = clock;
    for (all nodej waiting on nodei) {
        if (clock >= nodej's time) {
            decrease nodej's waiting count;
            if (its waiting count is 0)
                wake up nodej;
        }
    }
}
```

D. Node Partitioning for Parallel Execution

As indicated, the major potential source of overhead comes from the network synchronization necessary to keep the various emulations synchronized. To get maximal performance, we must reduce the remote synchronization as much as possible.

Thus partitioning the sensor nodes into groups plays an important role in the making of an efficient simulation.

The amount of remote network synchronization is determined by the number of remote neighborhood links between sensor nodes. Local updates to neighbors co-located on the same machine are relatively inexpensive (because they can use a shared data structure in memory) compared to remote clock update synchronization. As such our nodes partition algorithm has two goals. First, we need to evenly distribute the node workload among groups if we are running simulation on a homogeneous system like a dedicated cluster. This need for load balancing is because any slow host will become the bottleneck of the whole simulation due to the implicit dependency among nodes. Second, we want to minimize the number of links among remote neighbors that are assigned to processors that can only communicate via network messages.

We find that we can actually convert this optimization problem into a "classical" graph partition problem that is well studied in parallel computing area [34], [37], [28]. Formally, the partition problem is as follows. Given a weighted, undirected graph $G = (V, E)$, the k-way graph partition problem is to split the vertices of V into k disjoint subsets such that each subset has roughly equal amount of vertex weight while the sum of the weights of the edges whose incident vertices belong to different subsets (an edge cut) [34].

In our case, given a node map, which specifies the node coordinates in a 2D or 3D space, and the maximal transmission range of a typical sensor node, we can build up a graph called potential neighboring graph (PNG). Each vertex of the graph is a node. Each edge represents that the connected two nodes have the potential to communicate. Then the node partition problem is exactly a graph partition problem with both edge and vertex weights to be unitary.

The graph partition problem is known to be NP-complete in general. A large body of research explores heuristic algorithms. There are geometric algorithms, like recursive inertial bisection that uses coordinate axes to partition the graph; combinatorial algorithms, like K-L algorithm that optimizes an partition locally; spectral methods, which transform the discrete optimization into a continuous one using linear algebra; and multilevel algorithms featuring a coarsening-refining process. In our simulator, we use a general graph partitioning package for parallel computing from Sandia National Lab, called Chaco [10], which combines these techniques based on graph topology and vertex and edge weights. We use Chaco without modification and plan to report on its effectiveness in a future effort. Anecdotally, however, we are quite pleased with the quality of the partitions it generates for the simulations we have investigated.

E. Scalability Analysis

Before looking at the experimental results generated by our implementation, we attempt to describe the potential scalability of the system analytically. The simulation performance is determined by the pure device simulation speed and the synchronization overhead. Ultimately, the computational and memory cost of emulating individual devices will dominate performance, but the machine and memory speeds of the cluster hosts are so much more powerful than the devices simulated on them, it is the network synchronization that poses the greatest impediment to scalability.

We define the following property that describes the scalability of our algorithm in the ideal case.

Property 2 (Scalability): Given fixed map density d and node density D and node number N_h on each host, when the number of hosts H increases, and thus the simulated nodes N increase, the communication cost for each host is constant if the partition of nodes to hosts is optimal.
Here the map density d is defined as the ratio between the sum of areas of node range circles (the circle centered at the node with maximal radio range as radius) and the area of the map (maximal area that the nodes occupy). It is a good indication of nodes' average number of neighbors.

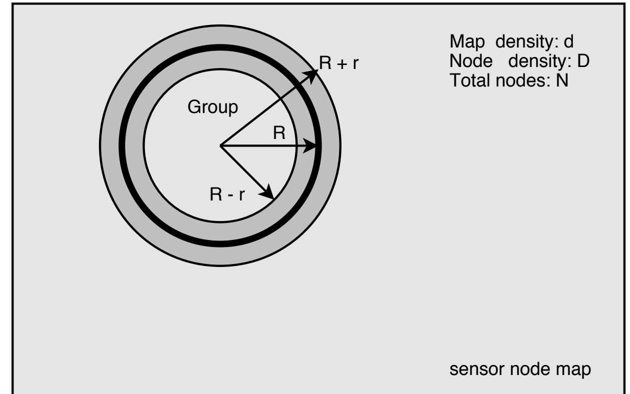

Fig. 1. Illustration of Property 2

The property is illustrated in Figure 1. The circle with radius R represents a group. We can use circle is because, assuming an optimal partition, the group should have minimal contact with others and a circle is a good estimation of its boundary. Since the map density d is fixed and the number of nodes per group is also fixed, the area of a group and thus radius R is constant. Moreover the ring area corresponding to the area between circles having radius $R - r$ and radius $R + r$ both having the same center is the area which nodes may have cross-group edges, if r defined to be the maximal radio transmission range. Then the number of cross-group edges for a group can be calculated as follows:

$$Num_{edge} \approx Area_{ring} * Density_{node} * Density_{map} = 4\pi RrdD \tag{1}$$

Since R, r, d and D are all constant with respect to the *total number of nodes*, the number of cross group edges is fixed. Thus the communication cost of each *host* is fixed. Although Property 2 corresponds to an ideal upper bound on communication overhead, it predicts that scalability will be affected most by the number of nodes assigned to each processor rather than the total number of nodes simulated or the total number of processors employed. Our experimental

results described in the next section seem to reflect this outcome.

IV. Evaluation

In this section, we examine the fidelity and scalability of DiSenS. As a measure of fidelity, we compare cycle counts generated by our simulator to those observed from individual real devices (using an oscilloscope to maximize measurement accuracy). This cycle count comparison is for full-device emulation (CPU and memory, flash storage, radio, etc.) using a set of benchmarks designed to exercise all sensor hardware subsystems.

We also investigate the transparency and completeness of our system by booting unmodified TinyOS images on the simulator and executing popular large sensor network applications: TinyDB, Surge and Deluge [12]. Finally we examine the scalability of DiSenS using a single benchmark (employed previously in the literature for such studies) and compare the results to those generated by previous efforts.

As part of these evaluations, we show how different models can be incorporated into simulations for the purposes of experimentation. We do not make an evaluation of these model plug-ins in terms of their accuracy or their effects on simulation fidelity, using them only to demonstrate flexibility. We do assert, however, that the underlying timing accuracy achieved by our system at scale constitutes an important capability necessary for the development and evaluation of such constituent models.

A. Experimental Framework

The results presented in the following exposition have been generated using two different machine clusters to which we have access at UCSB. CLUSTER1 is a 16 host [2] dual-processor 3.2GHz Intel Xeon cluster that uses switched gigabit Ethernet as its communication interconnect. CLUSTER2 is a larger, departmental cluster composed of 64 dual-processor 2.6GHz Intel Xeon hosts, again interconnected via a gigabit Ethernet switch. Both systems are used in dedicated mode to remove the effects of network or host contention by other executing applications.

For all the scalability experiments, we use TinyOS application *CntToRfm* as the the comparison benchmark. *CntToRfm* periodically sends out radio packets and makes the radio channel busy. Note that although it does not actually receive packets, the radio chip is still in receiving mode when it is not transmitting so it does in effect exercise all radio activities. For this reason, *CntToRfm* has been used as the touchstone in previous scalability studies [16], [43]; we follow suit.

For most of the experiments, we use Mica2 as the target sensor device. At the end of this section, however, we briefly discuss scalability results for MicaZ to show how the effect of radio transfer speed on simulation performance.

[2]The term "node" is rather unfortunately common to both the sensor network and cluster computing communities occasionally leading to confusion when discussing sensor network emulation on clusters. In the remainder of this paper, we will use the term "host" to refer to a node in a cluster, and the term "node" to refer to a sensor network device.

B. Cycle-Accuracy

We use four benchmarks to test the cycle-accuracy, exercising important components on the Mote device. *cpu* benchmark runs CPU intensive computations. *flash_read* performs small reads from the on-board flash chip. *flash_write* writes to the on-board flash. *radio* exercises the CC1000 radio chip and transfers a small amount of data.

The execution time on real device is measured using an oscilloscope, Agilent 54621A (accurate up to 10 nanoseconds). Each benchmark starts by writing an "1" to a pin in I/O port C and ends by writing a "0" to the pin. The pin is connected to the oscilloscope probe. The oscilloscope measures the pulse width. The measured time is then converted into cycle numbers using a division with ATmega128L's clock speed (7372800 Hz). The numbers are compared with our simulation result.

To eliminate the effects of noise and communication channel instability on cycle timings we attempt to implement the ideal radio model (clean channel, no contention) for real devices by putting the antennas of two Motes in direct contact. That is, to compare simulation results and real timings when two Motes communicate, the simulator uses the ideal radio model, and the corresponding Mote configuration has the antennae in physical contact.

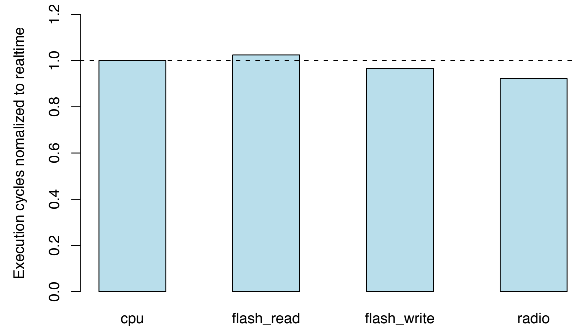

Fig. 2. Normalized average cycles for benchmarks.

Figure 2 gives the average of 20 measurements as the ratio of simulated execution cycles to cycles measured from the actual devices (a ratio of 1.0 would indicate perfect accuracy). That is, we normalize the data using actual measured cycle counts. For CPU emulation, the simulator closely approximates empirical measurement. Flash and radio emulation have slightly larger errors, but the size of these errors is of limited statistical significance. In [45] we provide a more complete statistical analysis of this comparison which we omit from this work due to space considerations. Instead, by way of summary, we note that in general the simulation error is relatively small.

C. Transparency and Completeness

To test the transparency and completeness of our simulator, we install TinyOS version 1.1.14 and then load and execute the version of TinyDB, Surge and Deluge that comes with the

release. TinyDB [18] is a sensor query engine providing a SQL language interface to the sensor network. Surge is a multi-hop routing algorithm for sensor network. Its protocol builds a dynamic spanning tree rooted from a single node. Deluge [8] is an epidemic data dissemination program. It is used to reprogram a sensor network using radio transferred program images. In each case, we use the binary image produced by the TinyOS cross-compilation system without modification. That is, the same binary for the operating system and each application loads and runs on the native Mica2 devices.

We test TinyDB and Surge using a 4x4 sensor grid. These two applications exhibit typical sensor network workflow by sequentially sensing, logging and transferring data. In addition, each application can use the serial interface of one of the Motes to forward program events to a powered device or workstation for visualization. Either the visualizer opens the serial port on the workstation and reads the events directly, or the events are read by a proxy (the serial forwarder) and forwarded across an attached network via the Internet protocols. Our emulation includes this serial-forwarding capability. A serial device is exposed by the emulator that can be read in the same way as the real serial forwarder. Figure 3 and Figure 4 show the screenshots of their GUI interfaces while the respective applications execute on the simulator. In developing and testing the system, we have found this level of software compatibility to be a valuable debugging aid.

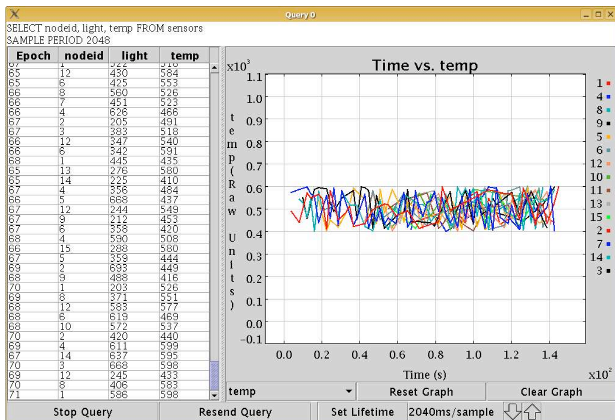

Fig. 3. Screenshot of TinyDB in execution, on 4x4 grid.

We also test Deluge using sensor grids from size 2x2 to 7x7. In each experiment, the sensors are arranged in a rectangular grid with 50 meter spacing. We take the maximal radio range for the Mica2 to be 60 meters so this test emulates a sparse sensor network deployment. Deluge heavily uses the internal flash and external flash memories of each Mote. It also exercises the self-programming function in the microcontroller. Again, the full-system emulator is transparent enough to support all of the communication and storage activities exercised by Deluge as well as the microcontroller functionality necessary to allow Deluge to perform a complete operating system installation and reboot.

In Figure 5, we show the dissemination time for the grids using both a lossless ideal radio model and also a model that

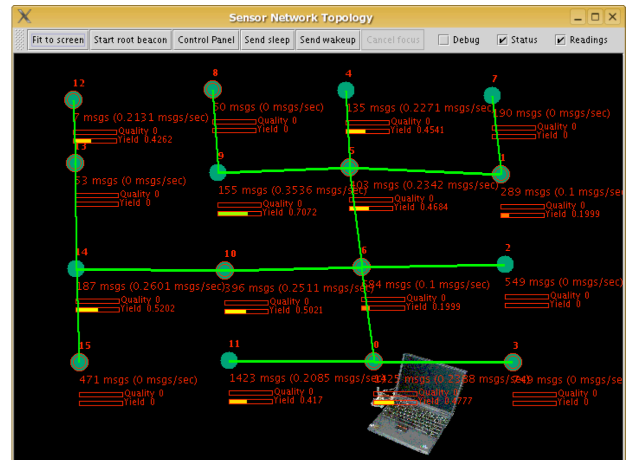

Fig. 4. Screenshot of Surge in execution, on 4x4 grid.

incorporates channel loss, and we compare these two curves to data reported in [8]. To model channel loss, we conducted a repeated radio survey using a pair of Mica2 motes on the UCSB sports fields at various times over a 3 month period. We varied the time of day, weather conditions, and Motes used for the survey and recorded communication performance as a function of distance and geographic orientation. Off-line, we analyze this data to produce a loss probability for each packet that is parameterized by the orientation of the sending and receiving Motes and their intervening distance. When the emulator sends a packet between Motes, it uses this probability to decide whether the packet should be dropped in transit or delivered successfully. This methodology is inspired by not identical to the technique used to generate the TOSSIM [16] radio model.

As expected, the introduction of channel loss dramatically affects the simulated completion time. We are, at present, attempting to quantify the error produced by this method of survey-based simulation for the geographic area we have surveyed. In this study, however, using radio packet traces as the basis for channel loss is intended to demonstrate the flexibility of the simulation approach. Indeed, as indicated, because we have verified the accuracy of the individual device emulations, we are now using the simulator to explore new, potentially higher-fidelity radio models that incorporate analysis of survey measurements.

D. Scalability

For each scalability experiment, we vary two experimental parameters independently: the number of sensor *nodes* simulated on each *host* in a cluster, and the number of *hosts* used for each experiment. Thus, for example, the value in row 2, column 4 shows the results for two nodes per host and four hosts. For each node-count-host-count pair, we run *CntToRfm* for 60 seconds and record the average simulated clock speed. Except where noted explicitly, all the experiments are run on CLUSTER1.

1) Best Case: One Dimensional Topology: Our first experiment simulates a one dimensional topology. All the nodes are

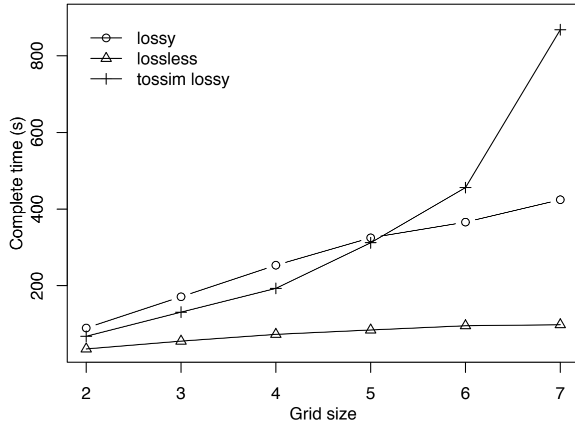

Fig. 5. Dissemination complete time for Deluge. X -axis is the size of grid. Y -axis is complete time. *lossless* is for the ideal radio model and *lossy* is for a lossy model based on radio survey data. The TOSSIM data *tossim lossy* is taken from figure 11 on page 11 in [8].

TABLE II

SIMULATED CLOCK SPEED FOR 1-D TOPOLOGY. EACH ROW HAS FIXED NUMBER OF NODES PER HOST AND EACH COLUMN HAS FIXED NUMBER OF HOSTS. ALL VALUE IS NORMALIZED TO REAL TIME CLOCK SPEED.

Nodes number	Hosts number				
	1	2	4	8	16
1	9.28	2.26	1.96	1.72	1.67
2	6.68	2.12	1.82	1.68	1.68
4	2.18	1.83	1.70	1.68	1.67
8	1.20	1.21	1.18	1.16	1.15
16	0.78	0.61	0.60	0.60	0.60
32	0.35	0.36	0.31	0.31	0.31
64	0.18	0.15	0.17	0.15	0.14
128	0.09	0.09	0.09	0.08	0.08

laid on a straight line, 50 meters apart (again assuming the maximal radio range is 60 meters). This gives us the minimal cross group edges (given an optimal partition). It constitutes the best possible case for the distributed simulation and as such provides a rough upper bound on the performance.

Table II presents the results. Each cell of the table shows the ratio of the simulated average clock speed to the real time clock speed, of 7372800 cycles per second. To compute the average simulated clock speed, the simulator records the number of clock cycles each Mote executed during the 60 second execution run. The sum of the cycles is divided by the number of Motes, and that number is divided by 60. Thus each cell depicts the average slowdown or speedup factor relative to native execution speed.

From the table, the best performance is a speedup of **9.28** times real time speed when simulating one node on one host (the upper lefthand corner in the table). Notice that as expected, simulating more nodes on a single host (column 1) yields a slower rate of decay in the speedup factor than does simulating one node on each of a successively larger number of hosts (row 1). When two nodes are co-located in the same host, the speedup factor drops to **6.68** whereas two nodes each located on a separate host generates a speedup factor of only **2.26**.

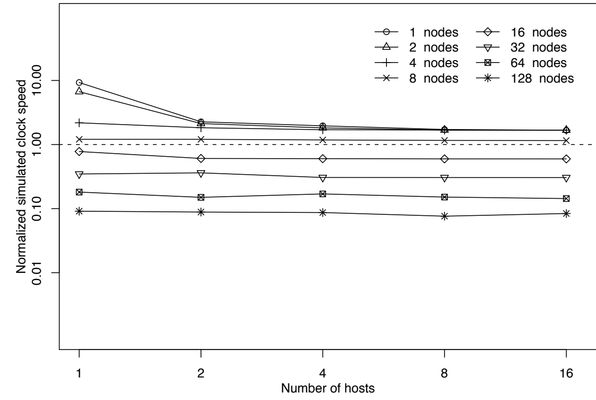

Fig. 6. Scalability of 1-D topology. X -axis is number of hosts and Y -axis is clock speed. Each curve represents the performance with a fixed number of nodes per host. Dashed line shows real time speed.

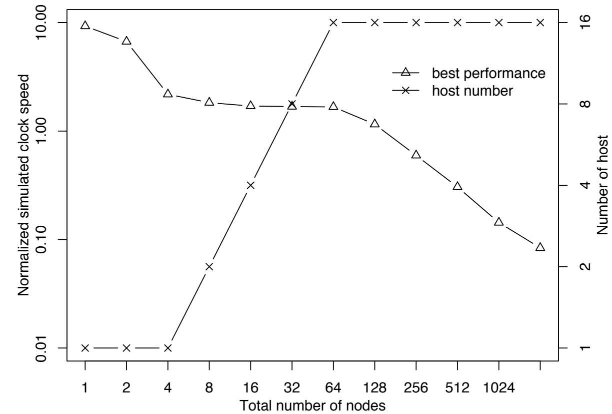

Fig. 7. Gold curves for 1-D topology. X -axis is total number of nodes simulated. The left Y -axis is normalized performance and the right one is number of hosts. The decreasing curve is the fastest speed curve. The increasing curve gives the corresponding host number at each point.

What is perhaps the most remarkable, however, is the similarity between the values for 2 through 16 hosts. While we expected a substantial fall off in speedup in moving from one host to two hosts, we expected that fall off to continue as the number of hosts increases. Indeed, starting with 8 nodes per host (the fourth row in the table) the speedup factors are remarkably similar regardless of host count. Further, the tipping point with respect to speedup and slowdown (the point where the ratio falls below 1.0) is between 8 and 16 nodes per host for *all* host counts.

Figure 6 shows this relationship graphically using a log-log scale. The speedup drops for small node counts from one host to two, but for the other data points, the number of nodes per host (and not the number of hosts) is the determining factor up to 16 hosts. This relationship is predicted by the analysis of Theorem 2 presented in the previous section but none the less, we found the degree to which it holds somewhat surprising.

By way of comparison to previous work, in this best case scenario **2048** nodes can be simulated at nearly a tenth of the real time speed using 16 hosts (lower righthand corner of

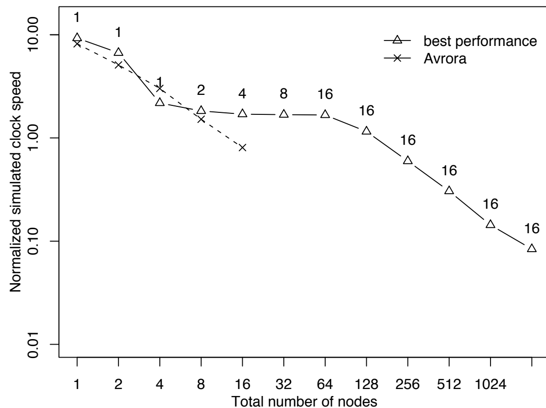

Fig. 8. Best performance curve comparison. X-axis is total number of nodes simulated. The Y-axis is normalized performance. Compare our best performance (1-D topology) with Avrora's performance. The annotated number is the corresponding number of hosts.

Table II), which is almost 8 times better than results reported for TOSSIM [16]. Also, nearly 160 nodes can be simulated in real time speed using 16 hosts, and improvement of almost a factor of 5 over previous TOSSIM results.

In Figure 7, we plot the best performance of simulating 1, a total of 2, 4, ..., and 2048 nodes respectively. The units of the y-axis on the lefthand side of the graph are for the ratio shown in Table II. For each point, we also plot the corresponding "host number" at which the best performance is achieved (the host count is shown on the y-axis at the righthand side of the graph). We call the two curves "gold curves" since they show the number of hosts necessary to obtain the fastest simulation of a specific number of nodes. Note that the fall off in the best performance curve occurs when the number of hosts reaches 16 (the maximum number in CLUSTER1) and the total node count is increased beyond 64. Thus, in this best case example, scalability is limited by host availability through 2048 simulated nodes.

We compare Avrora [43]'s best performance curve with our "gold curve" in Figure 8. We run Avrora on a single host from CLUSTER1 (using both processors on that host) for up to 16 nodes (the implementation of Avrora we ported to our machine did not execute correctly with more than 16 nodes). Recall that Avrora is not designed to use distributed memory parallelism and message passing but it can take advantage of multiple processors in a single hosts that share memory. Despite the extra overhead we have in our system that is necessary to take advantage of multiple hosts, the performance comparison is favorable to our work. For up to 8 hosts, our system and Avrora achieve similar speedup factors. For the 8 node comparison, however, we require 2 hosts, using both processors on each host (the small integers next to each triangular graph feature in Figure 8 indicate how many hosts our system requires to achieve the corresponding speedup factor) where Avrora is using only one. Beyond 8 nodes, however, our methodology, using successively larger host counts, achieves a considerable scalability improvement over Avrora.

TABLE III
SIMULATED CLOCK SPEED FOR 2-D TOPOLOGY. EACH ROW HAS FIXED NUMBER OF NODES PER HOST AND EACH COLUMN HAS FIXED NUMBER OF HOSTS. ALL VALUE IS NORMALIZED TO REAL TIME CLOCK SPEED.

Nodes number	Hosts number				
	1	2	4	8	16
1	9.14	2.52	1.83	1.66	1.64
2	6.65	2.12	1.58	1.38	1.18
4	2.09	1.49	1.27	1.12	1.10
8	1.25	1.07	1.01	0.96	0.92
16	0.82	0.63	0.62	0.59	0.57
32	0.32	0.38	0.31	0.30	0.30
64	0.16	0.17	0.16	0.15	0.15
128	0.10	0.08	0.07	0.07	0.07

2) Common Case: Two Dimensional Topology: A two dimensional topology is more realistic for sensor network applications. Using the same configurations, we perform the experiments on a two dimensional node map. The nodes are again 50 meters apart and fill a grid whose shape is as close to a square as possible. Table III provides the results. The performance for a 2-D space is somewhat worse than for the 1-D case when the number of nodes per host is below 32. However comparing Tables II and III for node-per-host counts above 32 shows surprising similarity. Again, as the number of simulated nodes increases, the number of available hosts becomes the scalability limiting factor – not the node count. In the 2-D case, however, performance equivocates between 16 and 32 nodes per host corresponding to a slowdown factor of between 0.6 and 0.3. That is, while it is possible for our system to achieve scalable 2-D simulation of the benchmark, it is not possible to do so and to run in faster-than-real time.

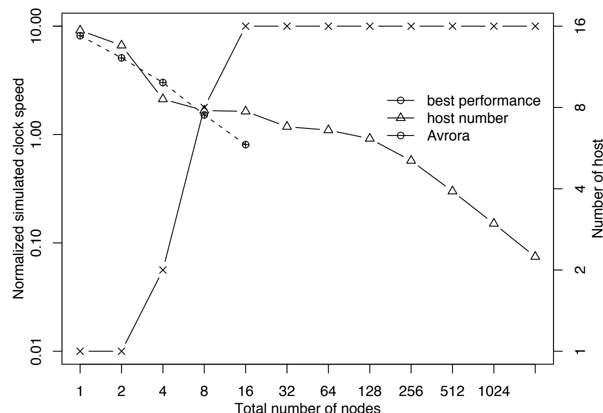

Fig. 9. Gold curves for 2-D topology. X-axis is total number of nodes simulated. The left Y-axis is normalized performance and the right one is number of hosts. The decreasing curve is the fastest speed curve. The increasing curve gives the corresponding host number at each point. The dashed curve is Avrora's speed curve.

We present the "gold curves" in Figure 9 but combine the node count and Avrora comparison curves onto a single graph. Again, our system performs similarly to Avrora (this time on the 2-D problem) but in this case, it requires more hosts to achieve the same results. For example, the simulator requires 8 hosts to duplicate Avrora's 8 node performance (using a

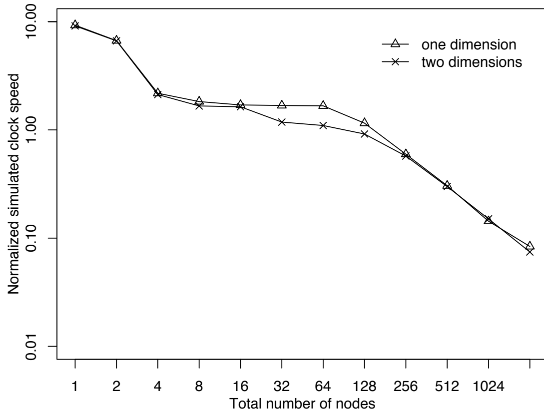

Fig. 10. Best performance comparison of 1-D and 2-D topology. X -axis is total number of nodes simulated. The Y -axis is normalized performance.

single host). Surprisingly, however, the 2-D gold curve and the 1-D gold curve have similar shape. Figure 10 shows both on the same graph (note the log-log scale). Between 32 and 128 simulated nodes there is a reduction in speedup factor for the 2-D case, but apart from that region, the curves track almost exactly.

TABLE IV

SIMULATED CLOCK SPEED FOR "ALL-TO-ALL" COMPLETE GRAPH. EACH ROW HAS FIXED NUMBER OF NODES PER HOST AND EACH COLUMN HAS FIXED NUMBER OF HOSTS. ALL VALUE IS NORMALIZED TO REAL TIME CLOCK SPEED.

Nodes number	Hosts number				
	1	2	4	8	16
1	9.28	2.36	1.66	1.60	1.36
2	6.68	1.41	1.07	0.81	0.66
4	2.04	0.94	0.75	0.62	0.42
8	1.22	0.65	0.54	0.43	0.29
16	0.62	0.44	0.32	0.23	0.14
32	0.29	0.20	0.14	0.08	0.04
64	0.12	0.08	0.04	0.02	0.01
128	0.05	0.02	0.01	0.002	0.0008

3) Worst Case: All-to-all Network: The previous scalability results we have presented rely on the limited neighborhood relationship imposed by radio range. For the worst case, we simulate an "all-to-all" complete graph configuration in which each simulated node must consider all of the other nodes to be in radio range making communication overhead maximal. Table IV and Figure 11 shows the speedup factors and scalability curves respectively. In this worst case, communication overhead increases as the square of the node density. For small node-per-host and host counts, the speedup factors are similar to the 1-D and 2-D grid cases, but as both are increased the speedup factor is continually reduced.

4) Larger Scale Experiment: To test our simulator in a larger scale, we perform the 2-D experiment on CLUSTER2, a 64-node cluster. Table V presents the results. Comparing Table V to Table III (which used CLUSTER1 for the same configuration) CLUSTER2 achieves lower speedup factors for the test cases they have in common (columns 1 through 5).

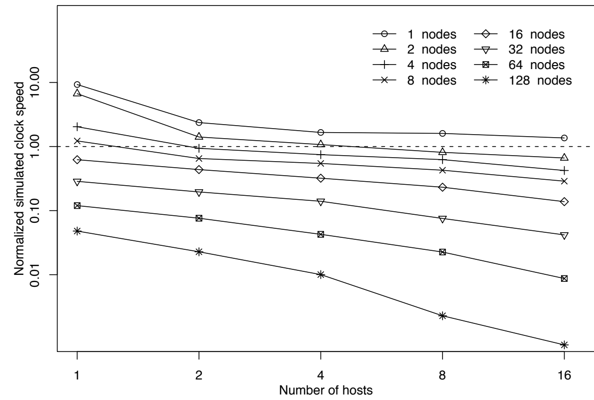

Fig. 11. Scalability of "all-to-all" complete graph. X -axis is number of hosts and Y -axis is clock speed. Each curve represents the performance with a fixed number of nodes per host. Dashed line shows real time speed.

TABLE V

SIMULATED CLOCK SPEED FOR SIMULATION OF 2-D GRID OF MICA2 MOTES ON CLUSTER2. EACH ROW HAS FIXED NUMBER OF NODES PER HOST AND EACH COLUMN HAS FIXED NUMBER OF HOSTS. ALL VALUES ARE NORMALIZED TO REAL TIME CLOCK SPEED.

Nodes number	Hosts number						
	1	2	4	8	16	32	64
1	7.21	0.85	0.70	0.55	0.45	0.41	0.41
2	3.33	0.55	0.50	0.44	0.38	0.34	0.32
4	2.51	0.55	0.48	0.42	0.39	0.35	0.34
8	1.37	0.51	0.44	0.39	0.36	0.39	0.30
16	0.74	0.47	0.39	0.37	0.37	0.36	0.33
32	0.37	0.32	0.29	0.29	0.27	0.28	0.23
64	0.17	0.16	0.15	0.15	0.13	0.16	0.12
128	0.08	0.07	0.07	0.07	0.07	0.08	0.07

Both a slower processor speed in CLUSTER2 and, somewhat curiously, higher network latency (even though CLUSTER1 and CLUSTER2 both use gigabit Ethernet as an interconnect) contribute to this lower performance. However, as the size of the sensor network being simulated scales, the overhead is once again amortized. For example, using 64 hosts of CLUSTER2 and 128 nodes/host our system can simulate **8192** nodes in total with a slowdown factor of 0.07 representing an almost **32** fold improvement over previously reported TOSSIM results.

TABLE VI

SIMULATED CLOCK SPEED FOR SIMULATED 2-D GRID OF MICAZ MOTES. EACH ROW HAS FIXED NUMBER OF NODES PER HOST AND EACH COLUMN HAS FIXED NUMBER OF HOSTS. ALL VALUES ARE NORMALIZED TO REAL TIME CLOCK SPEED.

Nodes number	Hosts number				
	1	2	4	8	16
1	8.67	0.18	0.13	0.13	0.12
2	1.53	0.11	0.09	0.07	0.06
4	0.89	0.10	0.07	0.06	0.06
8	0.35	0.08	0.06	0.05	0.04
16	0.18	0.07	0.05	0.04	0.04
32	0.10	0.06	0.05	0.05	0.04
64	0.05	0.04	0.04	0.04	0.04

5) Faster Radio: MicaZ: Another important consideration in measuring the scalability of our system is radio communication speed. In the previous subsections, our experiments are for the Mica2 platform. The newer MicaZ Mote uses the CC2420 radio chip which is almost 13 times faster than CC1000 used in the Mica2. A faster radio implies a slower simulation speed due to the communication overhead since for the CC2420, the simulator must wait and update 13 times more frequently. Table VI shows the 2-D experimental results for simulations using the MicaZ Mote. As expected, the speedup factors are considerably lower than for the Mica2 indicating that the simulator cannot achieve faster-than-real time performance. The scalability of the system is similar to that for Mica2 in that once the nodes per host is larger than 32 the number of available hosts becomes the scale-limiting factor.

V. Related Work

There have been numerous previously successful efforts to build sensor network simulation systems. Of these, Avrora [43] is the most similar to our work. Avrora is a full-system sensor network simulator supporting cycle-accurate emulation of the Mica2 Mote platform. Avrora uses a multithreaded structure in which each sensor node is simulated in a separate thread. A lock-step style synchronization scheme that is coordinated with the communication model is used to ensure the correctness of radio simulation. Avrora has a "Wait for Neighbors" operation that is similar to our *wait_on_sync* primitive, but it is only used to implement correct signal sampling operation. It also scales to multiple processors using shared memory. DiSenS provides similar cycle-accurate hardware emulation capability and some extra hardware support such as the Zigbee radio emulation. By employing cluster resources that do not share memory and carefully partitioning simulations between cluster machines, DiSenS achieves significantly greater scalability than has been reported for Avrora.

ATEMU [27] is another full-system sensor network simulator. It focuses on the detailed bit-level hardware simulation. It employs a very simple synchronization mechanism by executing one instruction a time for each sensor node. It is so simple that no extra facility and protocol is required to ensure correct radio simulation because nodes are already synchronized cycle-by-cycle using shared memory data structures. However, it can only utilize one process and thus does not scale to parallel computational resources.

Other simulators, include TOSSIM [16], SensorSim [26], GTSNetS [25], OLIMPO [2] and Shawn [14], explore the tradeoff between accuracy and performance by using discrete event simulation to elide the complexity associated with cycle-accurate hardware emulation. In addition, these systems do not achieve the transparency of DiSenS in that application and operating code must either be translated to, or compiled for, their respect discrete-event environments.

TOSSIM is a popular event-driven simulator which models not only the wireless network but also the application behavior. TOSSIM is light weight and can simulate thousands of sensor nodes on one host. None-the-less, we demonstrate how DiSenS is able to achieve improved performance levels by leveraging distributed cluster resources while achieving transparency and cycle accuracy. SensorSim is a sensor network simulator based on NS-2 [24] which is a discrete event wireless network simulator. It does not model application itself as TOSSIM does achieving even less transparency. A sensor network simulator based on GTNetS [32] claims to be able to simulate a sensor network at a scale of hundreds of thousands of nodes. This scale exceeds what we have been able to test using the resources at our disposal, but to achieve this level of scalability, the operating system and application codes must be represented in a high-level, abstract way. Thus it is not possible to use this system to directly compare executions of sensor network software in simulation and on real hardware, as it is using DiSenS.

EmStar [9] is an environment that uses libraries and user-space device drivers to emulate an embedded environment for deploying advanced sensor devices, like the Stargate [39]. It is used, primarily, to simulate the behavior of embedded applications but does not attempt to provide cycle-accurate fidelity. Scalability is also not the primary focus of the EmStar effort.

There are also a large body of literature on full-system simulation [33], [47], [46], [19], [20], [3], [29], [5]. But these efforts focus on the simulation of one architecture or device or the study of individual hardware subsystems. The focus of our work is complementary in that we implement ensemble simulations of multiple devices. To achieve the individual device fidelity and simulation performance we report, we are indebted to these efforts for many of the optimization techniques we have employed.

VI. Conclusion

DiSenS is a complete sensor network simulation framework providing high levels fidelity, extensibility and scalability. It addresses the conflict between accuracy and performance. Given enough computational resources, researchers do not have to trade simulation quality for simulation efficiency.

DiSenS also provides a complete and transparent simulation framework, including a cycle-accurate device emulator and replaceable plugin models. Users of DiSenS are able to employ customized models to explicitly control simulation quality. Internally, DiSenS uses a peer-to-peer simulation design for distributed clock synchronization. Individual node simulation threads are glued together by a simple and efficient synchronization protocol, which makes the complete simulation scalable to a large size of distributed computation resources. Using commodity cluster hardware, DiSenS can simulate one node approximately **9** times faster than real time speed, **160** nodes in real time speed using **16** dual-processor machines and **8192** nodes at nearly tenth of real time speed, which is **32** times of that reported previously [16].

We are actively improving DiSenS to make it a useful tool for sensor network research. A big challenge is to look for a dynamic node partition algorithm so that non-dedicated, heterogeneous distributed systems can be used for simulation.

We are also interested to expand the hardware support list so that the simulation can cover more platforms and can be used more widely.

REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 2002.

[2] J. Barbancho, F. Molina, C. Len, J. Ropero, and A. Barbancho. OLIMPO, An Ad-Hoc Wireless Sensor Network Simulator for Optimal SCADA-Applications. *Communication Systems and Networks (CSN 2004)*, 450, Sept. 2004.

[3] R. C. Bedichek. Efficient Memory Simulation in SimICS. *ACM SIGMETRICS*, 1995.

[4] L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. A discrete-time battery model for high-level power estimation. *In Proceedings of Design, Automation and Test in Europe*, 2000.

[5] The Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net`.

[6] A. Cerpa, J. L. Wong, L. Kuang, M. Potkonjak, and D. Estrin. Statistical Model of Lossy Links in Wireless Sensor Networks. *In the ACM/IEEE Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, Apr. 2005. Los Angeles, California.

[7] RF Receivers from Chipcon. `http://www.chipcon.com/index.cfm?kat_id=2`.

[8] A. Chlipala, J. W. Hui, and G. Tolle. Deluge: Dissemination Protocols for Network Reprogramming at Scale. *Fall 2003 UC Berkeley class project paper*, 2003.

[9] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks. *USENIX Technical Conference*, 2004.

[10] B. Hendrickson and R. Leland. The Chaco User's Guide: Version 2.0. Technical Report SAND94–2692, Sandia National Lab, 1994.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.

[12] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. *The 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, 2004.

[13] iPAQ devices. `http://welcome.hp.com/country/us/en/prodserv/handheld.html`.

[14] A. Kroeller, D. Pfisterer, C. Buschmann, S. P. Fekete, and S. Fischer. Shawn: A new approach to simulating wireless sensor networks. *eprint arXiv:cs/0502003*, Feb. 2005.

[15] O. Landsiedel, K. Wehrle, and S. Gtz. Accurate Prediction of Power Consumption in Sensor Networks. *In Proceedings of The Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, May 2005. Sydney, Australia.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. *ACM Conference on Embedded Networked Sensor Systems*, Nov. 2003.

[17] D. Linden and T. B. Reddy. *Handbook of Batteries(3rd edition)*. McGraw-Hill, 2002.

[18] S. R. Madden, M. J. Franklin, J. M. Hellerstein, , and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. *In Proceedings of SIGMOD 2003*, June 2003.

[19] P. Magnusson and B. Werner. Efficient Memory Simulation in SimICS. *Simulation Symposium*, 1995.

[20] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrm, and B. Werner. SimICS/sun4m: A Virtual Workstation. *USENIX Technical Conference*, 1998.

[21] Mica2 sensor board. `http://www.xbow.com/`.

[22] MicaZ sensor board. `http://www.xbow.com/`.

[23] Mote hardware platform. `http://www.tinyos.net/scoop/special/hardware`.

[24] NS-2 network simulator. `http://www.isi.edu/nsnam/ns/`.

[25] E. Ould-Ahmed-Vall, G. F. Riley, B. S. Heck, and D. Reddy. Simulation of Large-Scale Sensor Networks Using GTSNetS. *In Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05)*, 2005.

[26] S. Park, A. Savvides, , and M. B. Srivastava. SensorSim: a simulation framework for sensor networks. *ACM International workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, 2000.

[27] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J. S. Baras. ATEMU: A Fine-grained Sensor Network Simulator. *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004.

[28] A. Pothen. Graph partitioning algorithms with applications to scientific computing. *Parallel Numerical Algorithms*, pages 323–368, 1997. Kluwer.

[29] QEMU: A Generic and Open Source Processor Emulator. `http://fabrice.bellard.free.fr/qemu/`.

[30] Wireless Propagation Bibliography. `http://w3.antd.nist.gov/wctg/manet/wirelesspropagation_bibliog.html`.

[31] D. Rakhmatov and S. Vrudhula. Time-to-failure estimation for batteries in portable electronic systems. *In Proceedings of the International Symposium on Low Power Electronics and Design*, Aug. 2001.

[32] G. F. Riley. Large-scale network simulations with GTNetS. *In Proceedings of the 2003 Winter Simulation Conference*, 2003.

[33] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, winter:34–43, 1995.

[34] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. *Draft to be included in CRPC Parallel Computing Handbook, Morgan Kaufmann*, Sept. 2000.

[35] V. Shnayder, M. Hempstead, B. rong Chen, and M. Welsh. PowerTOSSIM: Efficient Power Simulation for TinyOS Applications. *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004. Baltimore, MD.

[36] V. Shnayder, M. Hempstead, B. rong Chen, G. Werner-Allen, and M. Welsh. Simulating the Power Consumption of Large-Scale Sensor Network Applications. *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004. Baltimore, MD.

[37] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[38] Simulavr: A simulator for the Amtel AVR processor family. `http://www.nongnu.org/simulavr`.

[39] Stargate: a platform X project. `http://platformx.sourceforge.net/`.

[40] S. Sundresh, W. Kim, and G. Agha. SENS: A Sensor, Environment and Network Simulator. *The IEEE 37th Annual Simulation Symposium*, 2004.

[41] K. C. Syracuse and W. Clark. A statistical approach to domain performance modeling for oxyhalide primary lithium batteries. *In Proceedings of Annual Battery Conference on Applications and Advances*, Jan. 1997.

[42] Moteiv Corporation. Telos Sensor Network Module. `http://www.moteiv.com/`.

[43] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. *The Fourth International Symposium on Information Processing in Sensor Networks*, Apr. 2005.

[44] F. A. Tobagi and L. Kleinrock. Packet switching in radio channels: Part II-The hidden terminal problem in carrier sense multiple-access and the busy-tone solution. *IEEE Transactions on Communications*, COM-23:1417–1433, 1975.

[45] Y. Wen, S. Gurun, N. Chohan, R. Wolski, and C. Krintz. Toward Full-System, Cycle-Accurate Simulation of Sensor Networks. Technical Report CS2005-12, University of California, Santa Barbara, 2005.

[46] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):68–79, May 1996.

[47] Intel XScale XDB Simulator 2.0. `http://www.intel.com/design/pca/prodbref/250424.htm`.

[48] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. *In Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys'03)*, 2003.

[49] G. Zhou, T. He, S. Krishnamurthy, and J. A. Stankovic. Impact of radio irregularity on wireless sensor networks. *In Proceedings of the 2nd international conference on Mobile systems, applications, and services (MobiSYS'04)*, 2004.