Transaction-Level Modeling for Sensor Networks Using SystemC

Jeff Hiner[‡], Ashish Shenoy[‡], Roman Lysecky[†], Susan Lysecky[†], Ann Gordon Ross[§]
Department of Electrical and Computer Engineering
[†‡]University of Arizona, [§]University of Florida
[‡]{jhiner, ashenoy}@email.arizona.edu, [†]{rlysecky, slysecky}@ece.arizona.edu, , [§]ann@ece.ufl.edu

Abstract—**As sensor networks are finding widespread use across many applications, designers increasingly must not only focus on application development, but also on sensor network optimizations. Given the complexities of sensor networks and the difficulty of analyzing the long-term effects of design changes within a deployed system, simulation is often the only feasible option for evaluating such optimizations. The Arizona Transaction-Level Simulator for Sensor Networks (ATLeS-SN) is a transaction-level modeling based sensor network simulation environment emphasizing modular design for modeling various components within sensor nodes and across the sensor network. We provide an overview of our proposed ATLeS-SN simulation framework and highlight the benefits of this framework for a building monitor application and a forest fire detection and propagation tracking application.**

Keywords-Sensor networks; simulation; transaction-level modeling; SystemC; profiling

I. Introduction

Sensor networks are collections of sensor nodes typically comprised of a microcontroller, sensors and actuators, radios, and power source. Many sensor networks are inexpensive to manufacture and can be rapidly deployed. Hence, sensor networks are seeing increased use in a wide variety of applications including industrial controls monitoring, wildlife habitat monitoring, military applications, and even everyday use in home automation, garage door openers, or intrusion detection systems [18].

Given the significant differences in requirements for sensor network applications, designers must not only focus on application development, but also on sensor network optimization intended to maximize performance, throughput, lifetime, or other high-level metrics. For example, a sensor network utilized for gunshot localization would require precise timing and latency constraints along with high computational performance. On the other hand, a sensor network deployed in a remote area for wildlife monitoring may require a very long lifetime because an operator may be unable to frequently access the system for maintenance.

Unfortunately, the complexity of sensor network systems makes it difficult to assess the impact of design changes or optimizations on system-level design constraints, such as long-term power consumption. Taking empirical measurements by deploying a specifically optimized sensor network for several months and verifying power consumption is impractical. Thus, simulating a sensor network is typically the only feasible option for evaluating and optimizing a sensor network.

Simulation of wireless sensor networks has been the focus of much previous work. Network simulation tools such as REAL [9] and NS2 [13] pioneered simulation in this field in order to test routing and MAC protocols for current and emerging network standards. However, network-level simulators are not designed to simulate applications, nor are they suitable for simulating individual sensor node power consumption. As these tools were developed before sensor node hardware became commonplace, they typically cannot emulate the specific operation of individual nodes apart from the radio transceivers. In addition, users must be well versed in network architectures, protocols, and terminology to effectively utilize this class of simulators.

After standards were developed and physical hardware for sensor network became more readily available, simulators such as TOSSIM [11] and PowerTOSSIM [16] emerged to simulate both the functionality and power consumption of individual nodes – specifically focusing on TinyOS-based mote devices – by simulating the microprocessor and radio hardware with instruction-level precision. However, such simulators can only simulate specific sensor nodes and typically require significant simulation time due to the low-level at which the simulation is performed. In many cases, such timing accuracy may be unnecessary.

XRM's reactive modules simulator employed statistical methods and high-level state machines modules to simulate a sensor network [4]. While providing a fast simulation approach, statistical approaches often have insufficient capability to reproduce unique or aberrant sensor events, and it is often difficult or infeasible to produce a statistical model that captures the behavior of a real application.

CENSE combines simulation methods with hardware emulation of physical sensor nodes to provide accurate low-level simulation capabilities [10]. This combined simulation/emulation approach can provide very accurate measurements. For example, by coupling the simulation approach with power measurement hardware, an accurate analysis of the sensor network power consumption can be determined. However, such emulation approaches require specialized hardware for each emulated physical sensor node and are difficult to scale to large sensor network simulations. In addition, because the emulation hardware is specific to the nodes being tested, the model is difficult to generalize or adapt to other sensor node hardware.

While these existing simulation approaches are effective for their respective purposes, they lack the ability to model a sensor network across all levels – including node level hardware, application level software, and network level interconnection. A more holistic and modular system-level simulation is needed to effectively analyze and evaluate high-level design metrics such as power consumption or long-term fault tolerance. For example, an effective simulation tool for sensor networks should allow a designer to efficiently and accurately estimate the energy consumption of sensor nodes while abstracting network protocol details in order to increase simulation speed. While many simulators offer point solutions for evaluating specific elements of sensor nodes or sensor networks, a more general and modular framework would provide many advantages.

In this paper, we present the Arizona Transaction-Level Simulator for Sensor Networks (ATLeS-SN) – a transaction-level modeling and simulation environment for sensor networks implemented using the SystemC language. ATLeS-SN emphasizes modular design for modeling various components both within sensor nodes and across the sensor network. This flexible approach allows designers to focus on modeling, simulating, analyzing, and optimizing specific sensor network components without requiring a detailed timing accurate implementation across all levels. In Section 2, we provide an overview of transaction-level modeling

(TLM) and the SystemC language. In addition, we specifically highlight two previous sensor network simulation efforts that utilized transaction-level modeling and SystemC. In Section 3, we provide a detailed overview of our ATLeS-SN framework. In Section 4, we highlight the benefits of this framework for two applications: a building monitor application and a forest fire detection and propagation tracking application. Through these case studies, we highlight the development efficiency, simulation scalability, and ease of incorporating additional functionality afforded by ATLeS-SN. Most importantly, we highlight how ATLeS-SN enabled us to quickly analyze the forest fire detection and propagation tracking application in the early design phases to determine that the current implementation was unsuitable, thereby enabling us to quickly identify the shortcoming in order to make the necessary modifications. Finally, in Section 5, we conclude and highlight future directions.

II. TRANSACTION-LEVEL MODELING AND SYSTEMC OVERVIEW

Transaction-level modeling is a modeling technique intended to separate the specification of computation and communication while providing efficient methods for implementing the various elements at different levels of abstraction [2][5]. Figure 1 provides a basic overview of transaction-level modeling, in which a model can consist of components, channels, interfaces, ports, and connections. Within a TLM model, components correspond to the basic computational, physical, or storage elements. In the provided example, the microprocessor *(μP)*, memory *(Mem)*, and peripherals *(PE_1, PE_2, ..., PE_n)* are all components. In contrast to components, channels provide the basic means of communicating between components, such as the instruction and data bus *(Bus)* connecting a microprocessor and memory. Both components and channels can define one or more interfaces that specify a set of methods – or *transactions* – that can be utilized to interact with the respective component or channel. In addition, components can have multiple ports that specify the type of interface(s) to which a component can be connected. A port can be connected to a component or channel as long as that component or channel implements the corresponding interface. In that sense, a single port can only be connected to one component or channel.

Transaction-level modeling provides many benefits that make it a useful tool for modeling, designing, and simulating systems. The inherent modularity of TLM allows designers to model and refine the individual elements of a system with varying levels of detail and timing accuracy. On the one extreme, a functional un-timed implementation can be utilized to identify the required elements verifying functional correctness but provides little details regarding the final implementation or resulting system performance. Such a model can be extended to an approximate-timed model by incorporating timing annotations to approximate the time required to perform specific operations. At the other extreme, a cycle-accurate bit-accurate implementation defines the exact cycle-by-cycle timing where all operations are defined at the bit level often corresponding to a synthesizable.

Transaction-level modeling provides an efficient means of designing complex systems by allowing designers to start with a high-level functional model and successively refine that model until the final cycle-accurate bit-accurate implementation is reached. For example, consider a designer tasked with designing a system implementing a new image processing application. Initially, the designers may start with a functional abstraction consisting of a microprocessor connected to a memory. As the target microprocessor may not yet be known, the software can be directly model with a C/C++ implementation of the required

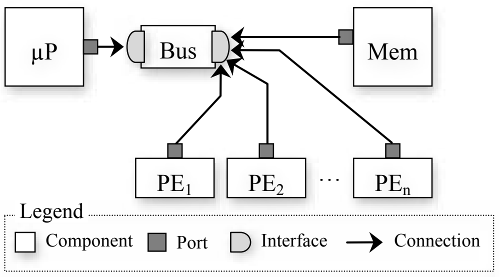


Figure 1. Transaction-level model example consisting of several components (*μP*, *Mem*, *PE_1*, *PE_2*, …, *PE_n*) connected through a communication channel (*Bus*) with two interfaces.

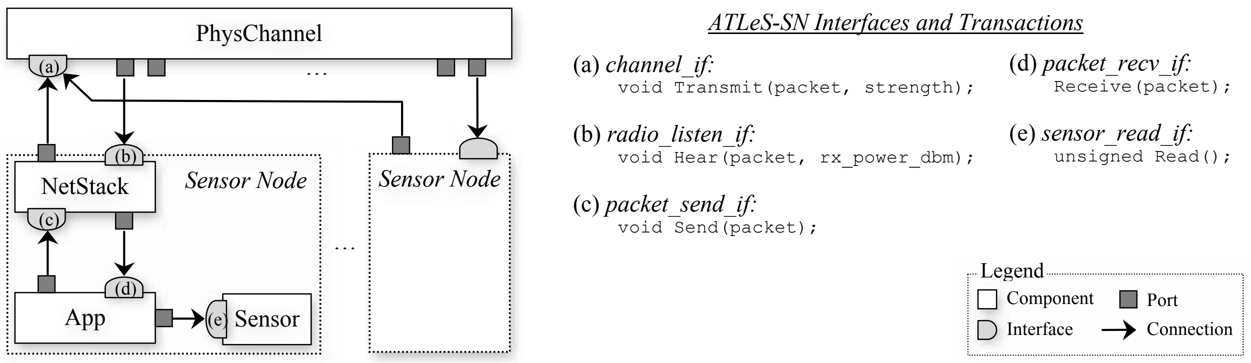
algorithm. Once the designer has selected a microprocessor, the C/C++ implementation can be replaced with an instruction-accurate instruction-set simulator. At this level, only approximate timing is available, as the instruction accurate simulation may not model the delays due to cache misses, page faults, or branch mispredictions. However, the approximate timed performance data can be utilized to evaluate various code optimizations. As this implementation is further refined, a cycle-accurate implementation may be required in which the instruction-set simulator can be replaced by a synthesizable VHDL/Verilog implementation of the target processor. At the time a system design is first envisioned, detailed timing information is often not available perhaps due to insufficient documentation or an incomplete hardware design. Despite this lack of detail, transaction-level modeling provides a useful model for simulating and evaluating early design options, thereby reducing the design space as further details and requirements are known.

Transaction-level modeling has gained in popularity over the past several years and has been increasingly supported within various design languages, including SystemC [14], SpecC [7], and among others. While SystemC was originally designed as a hardware description language comparable to VHDL or Verilog, the current SystemC standard provides robust system-level modeling capabilities that make it an excellent option for transaction-level modeling. SystemC is a modeling language built on top of C++ that provides the required modeling abstractions required to implement complex system design while still retaining the cycle-accurate bit-accurate modeling suitable for synthesizable hardware descriptions. One of the advantages of SystemC is the integration of simulation engine within the SystemC implementation itself. Thus, designers can compile and simulate a SystemC-based model using any standard C++ compiler.

A. *Transaction-Level Modeling of Sensor Networks*

Transaction-level modeling and SystemC provide an intuitive method for modeling the various elements of a sensor network. For each sensor node, the functionality and/or timing of various physical components, e.g. microprocessor, sensors, and radio, as well as software components, e.g. application, operating systems (OS), drivers, can be efficiently modeled. Given the complexity and number of elements that need to be implemented within a complete sensor network system, transaction-level modeling offers tremendous flexibility in both how the various components are modeled as well as allowing designers to focus on specific components of complex sensor network implementation.

For example, a designer may be interested in implementing and analyzing the performance of the software executing on each sensor node. The designer might choose to model the processor using a cycle-accurate instruction set simulator. This would provide very precise timing information for the sensor node. A



ATLeS-SN Interfaces and Transactions

(a) *channel_if:*
```
    void Transmit(packet, strength);
```
(b) *radio_listen_if:*
```
    void Hear(packet, rx_power_dbm);
```
(c) *packet_send_if:*
```
    void Send(packet);
```
(d) *packet_recv_if:*
```
    Receive(packet);
```
(e) *sensor_read_if:*
```
    unsigned Read();
```

Figure 2. Overview of Arizona Transaction-Level Simulator for Sensor Networks (ATLeS-SN) highlighting the underlying components, interfaces, and transactions needed to model various elements with a sensor network.

disadvantage of this approach is that a single node modeled at this level of abstraction would require long simulation times. Even with state-of-the-art high-performance workstations, simulating a small network with tens of nodes would be prohibitively time consuming. Instead, a designer can save time by implementing an approximate-timed model [1] in which the functionality of various elements is implemented at a high level of abstraction, along with timing statements that provide an approximation of the time required for the main tasks being performed. This simulation method can provide relatively accurate timing information while requiring significantly less time for simulation.

Both transaction-level modeling and SystemC have been utilized to develop simulation frameworks specifically targeting sensor networks. In [19], a SystemC-based simulation environment originally targeted at modeling System-on-Chip (SoC) systems was adapted for modeling and simulating a sensor network. Modeling efforts primarily focused on modeling each sensor node as a set of tasks – both for processing and I/O – executing on a real-time OS, with significant attention directed at modeling the various states of each task to facilitate efficient scheduling. In addition, the physical environment model tightly coupled nodes' sensors and actuators – each sensor node's radio was modeled as a sensor for receiving and actuator for transmission – with the simulated physical location and sensor data.

In [15], SpecC was utilized to model various aspects of a specific sensor network application. SpecC is a system-level design language that provides transaction-level modeling capabilities allowing designers to independently design individual components within the target application. The resulting sensor network model utilized pre-existing SpecC communication functionality to provide the basic mechanisms needed to model both handshaking and controlled access to shared radio resources. This case study demonstrated the feasibility of utilizing transaction-level modeling for sensor network applications, but the focus was not on developing a generic simulation framework that could be quickly and efficiently adapted to different applications.

III. Arizona Transaction-level simulator for Sensor Networks (ATLES-SN)

The Arizona Transaction-Level Simulator for Sensor Networks (ATLeS-SN) is a simulation framework built using SystemC and transaction-level modeling that emphasizes modular design for modeling various components both within sensor nodes and across a sensor network. This modularity, which is inherent in a transaction-level implementation, allows designers to focus on specific modeling tasks, such as application development, networking protocols, or even the physical environment in which the sensor network is deployed.

Figure 2 provides an overview of the ATLeS-SN framework, specifically identifying the components, interfaces, ports, and connections that provide the underlying foundation for modeling and simulating various sensor network applications. Each *Sensor Node* consists of an *App* component for modeling or implementing a node's functionality, a *Sensor* component for modeling the sensor (or sensors) available within each node, and a *NetStack* component for modeling the network-level communication. In addition, all nodes connect to a *PhysChannel* component that models the physical medium, e.g. wired or wireless, by which the nodes are connected.

In the ATLeS-SN framework, components can either be active or passive. A passive – or synchronous – component will block execution of the calling component until the specific transaction is complete. On the other hand, an active – or asynchronous – component is an independent component that accepts the information needed to perform a transaction without requiring the calling component to wait for the operation to complete. Instead, the result of the operation may come at any time, typically via a separate callback transaction or callback interface defined within the calling component.

As further annotated in Figure 2, each component with the ATLeS-SN framework contains one or more interfaces that define the transactions – or functions – by which the connected components can interact. The following section provides a detailed overview of the various components, their interfaces, and their connections to other components with the proposed modeling and simulation framework.

A. Sensor Node-level Modeling

The ATLeS-SN framework provides a virtual *Sensor Node* component that encapsulates the *App*, *NetStack*, and *Sensor* components within each node along with any shared data items needed by multiple components. This shared information includes the node identifier (i.e. a unique numeric identifier utilized for addressing nodes), a node's physical coordinates, and execution statistics including the number of packets sent, the number of packets received, number of packets retransmitted, etc. A designer can extend this shared data object to incorporate additional information. The sensor node's shared data object provides a designer with an area typically represented by a block of physical shared memory within the sensor node that can be directly accessed by multiple components.

1) Application-level Modeling (App): The *App* is an active component that provides the underlying foundation needed to implement the functionality for a specific sensor network application. This component is analogous to the user-level application software that will execute on the processor within a physical sensor node. However, this abstraction does not include the device drivers necessary to interface with physical sensors or network-level communication.

The *App* component defines the *packet_recv_if* interface that allows the application to receive packets from other nodes within the network. The *packet_recv_if* interface defines a single transaction `Receive` that provides the application with packets specifically being sent to a particular node. As described later, the *NetStack* component is responsible for implementing the network-level communication – which may receive additional packets that need to be re-transmitted or forwarded through the node – but the application within a sensor node will only receive those packets specifically sent to the node. This modularity and separation of concerns allows an application developer to focus on the specific functionality of the sensor node without the need to understand the corresponding low-level network protocols.

Each *App* component contains a port that will be connected to an interface for transmitting packets (typically its *NetStack* component) and one or more ports connected to *Sensor* components. Note that the simplest implementation of an *App* component would neither read sensor values nor transmit packets, but would still be required to receive packets as it must implement the `Receive` function of the *packet_recv_if* interface.

Within ATLeS-SN, the functionality of the application can be modeled at various levels of abstraction. At the cycle-accurate level, the *App* component can also be implemented as an instruction set simulator for the processor incorporated within the target sensor nodes, which executes the compiled application binary for the software application. The advantage of this approach would be the ability to accurately measure sensor node performance and energy consumption.

Alternatively, the *App* can be abstractly modeled as a C/C++ implementation within the *App* component. While this abstraction lacks the low-level details of an instruction set simulation, a designer can directly incorporate analysis and estimation methods within the *App* component. As most computational events within a sensor node are performed periodically, the energy consumption of a specific computation can be modeled as the average energy consumption [17]. Hence, an *App* component in our model includes a computation event counter that should be incremented by the application designer whenever any energy expensive computations occur. Such an energy-approximate model will provide sufficient accuracy for monitoring microprocessor energy consumption or implementing a battery model to analyze sensor node lifetime, as highlighted in Section IV.B.

2) Sensor-level Modeling (Sensor): A Sensor component provides the underlying foundation for modeling physical sensors, e.g. temperature sensor, accelerometer, connected to the sensor node. In ATLeS-SN, sensors can be modeled at various abstractions ranging from a generic sensor interface that reads sensor values from an input file to a detailed model of a specific sensor incorporating the device driver code needed to interface with the sensor within a specific physical sensor node.

Each *Sensor* component must implement the *sensor_read_if* interface. This interface consists of a single transaction `Read` that returns the sensor reading as an `unsigned` data value. Although sensor measurements may correspond to real numbers – such as measurement from an accelerometer – the raw sensor data received by a processor must be converted according to the specific sensor being utilized. As such, the *sensor_read_if* interface provides a similar abstraction in that the values read from the sensor are provided as unsigned integers and may need to be processed by the sensor node's software application.

Currently, all reads from a sensor are blocking synchronous transactions, in which the `Read` function will block until the sensor reading is available. Alternatively, a non-blocking asynchronous interface could be implemented by adding a callback interface to the *App* component to receive the sensor readings directly from the *Sensor* component asynchronously.

The *Sensor* component also incorporates a sensor read event counter that allows a designer to monitor sensor activity. Similar to the computation event counter, the sensor read event counter could be utilized to estimate the energy consumption of activating and reading values from the physical sensor. For nodes with active sensors, such as the Crossbow IRIS [3], this estimation method can provide an accurate estimate of energy consumption, as the sensors are inactive – or powered off – before and after each sensor access.

3) Network-level Modeling (NetStack): The *NetStack* is an active component that provides the foundation for modeling and implementing both the software and hardware components needed for the network stack and physical radio interface. Hence, the *NetStack* bridges the high-level software commands from the application to the physical transmission and reception of packets within the network. The implementation of network details such as Media Access Control (MAC), error correction, and retransmission are encapsulated away from the application as much as possible.

The *NetStack* component defines the *packet_send_if* interface consisting of a single `Send` transaction. The `Send` transaction is called from an *App* component to transmit packets to other nodes within the network utilizing the specific network protocol implemented by the *NetStack* component. As such, the *NetStack* is responsible for implementing a packet buffer so that multiple packets sent by the *App* are stored until they can be transmitted to the physical channel via the *NetStack's* port connected to a *PhysChannel*.

The *NetStack* also implements *radio_listen_if* that defines the `Hear` transactions. The `Hear` transaction is called whenever the current node can hear a transmission from another node and provides the received packet along with the signal strength of the transmission. The *NetStack* component is responsible for implementing any MAC protocols, as well as ensuring the *App* does not receive packets addressed to other recipients. In addition, the *NetStack* handles retransmissions, packet forwarding, fragmentation, and other network-level details supported by the platform. As the network stack typically contains some software elements, a designer can increment the computation event counter as needed to account for this software execution.

Packets transmitted within the ATLeS-SN framework are dynamically constructed objects containing a destination ID, link source identifier, handler identifier, payload, as well as any other protocol specific information needed by the *NetStack*. The destination identifier is the identifier of the node to which the packet is being transmitted. The link source identifier is the identifier of the node that last transmitted the packet. The handler identifier is an application specific identifier that typically defines the type of data carried within the packet, and how it should be handled. The *Packet* object provides a generic object that can be

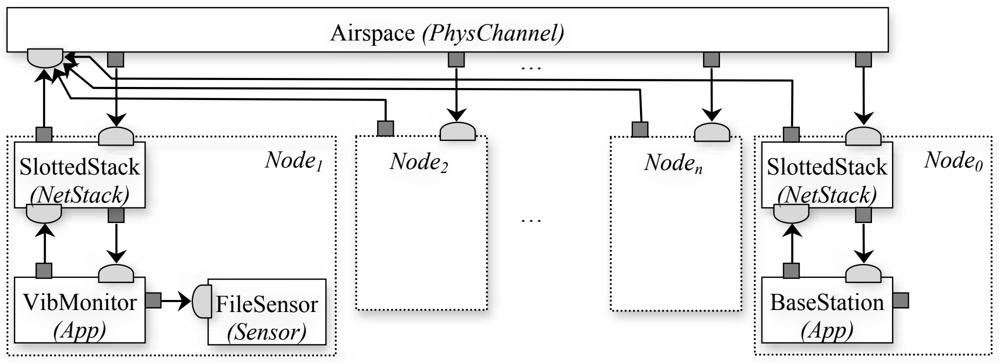


Figure 3. Building monitor application highlighting the specific implementations of the underlying components with the ATLeS-SN simulation framework.

readily extended to include additional information needed by the application via the payload and the network protocol via additional header fields.

B. Physical Channel-level Modeling (PhysChannel)

The *PhysChannel* component provides a method for modeling the physical channel through which the sensor nodes communicate by modeling the real life propagation medium over which the sensor nodes transmit packets. The *PhysChannel* defines a single *channel_if* to which all nodes connect to transmit packets. The *channel_if* interface defines the `Transmit` transaction called from a sensor node's *NetStack* that provides the packet to be transmitted along with the transmission strength at the point of origin. The *PhysChannel* also consists of one port for each sensor node within the system, where each port connects to a specific node's *NetStack* component.

The *PhysChannel* emulates inter-node communication issues such as collisions. Depending on the implementation, a collision may result in a jam resulting in no packets getting through, or the corresponding network protocol may contain a backoff or negotiation method ensuring that the transmission with the highest priority is always received.

The *PhysChannel* determines whether a packet transmission is heard by each sensor node's *NetStack*. As sensor nodes typically communicate via radios, the ATLeS-SN framework incorporates specific fields for specifying and modeling transmission and reception strengths, which can be ignored for other media. Using radio propagation models, the *PhysChannel* determines the signal strength at each receiver in dBm. If a particular node can hear a packet transmission, the *PhysChannel* will call the `Hear` function of the node's *NetStack radio_listen_if* interface. The dBm parameter allows the network stack to calculate its received signal strength for that packet transmission if needed. For example, the transmission and reception strengths may be used by nodes to determine their nearest neighbors for distributed clustering or to dynamically determine an optimal spanning tree for routing packets within the network.

A designer can implement various radio propagation models within the *PhysChannel* component. The model might include deflection or echoes from obstacles and terrain, integrate active interference objects, utilize ground reflection models [6], or use a simple equation based on distance in free space. Additionally, the *PhysChannel* model might also discard a certain percentage of packets depending on received signal strength or introduce bit errors as a function of the reception strength. In an ideal case, the *PhysChannel* implementation may simply assume that each node can hear all other nodes equally with zero packet loss.

IV. Case Studies and Experimental Results

We utilized ATLeS-SN to develop and simulate two sensor network applications: a building monitoring system, and a forest fire detection and propagation tracking system. For both systems, we primarily focus on utilizing the ATLeS-SN simulation framework to develop and test application functionality. Additionally, we further demonstrate how a designer can leverage the modularity of ATLeS-SN to incorporate additional functionality by integrating a profiling component within each node of the forest fire detection and propagation tracking application to analyze various application statistics. The resulting profiling methodology allows us to efficiently and non–intrusively monitor the sensor network during simulation as well as evaluate the potential overhead of utilizing such profiling at runtime within a deployed system to monitor the status and health of the network.

A. Building Monitor Application

The building monitor application is designed to monitor movement within a building using periodic sampling of motion/vibration sensors. As the need to monitor activity varies with the time of day, nodes may operate in a low-power mode in which the sensor and radio components are turned off.

Figure 3 presents an overview of the building monitor application implement within ATLeS-SN. The building monitor application consists of a single base station node and several vibration monitor nodes. In the base station node, the *App* component is implemented as a *BaseStation* component. However, as the base station does not directly detect movement, the base station does not require a *Sensor* component. Within the remaining sensor nodes, the *App* component is implemented as a vibration monitor, *VibMonitor*. Within these sensor nodes, an accelerometer connected to a floor plate is utilized to detect vibrations and movement within the affected room. These *Sensor* components are implemented using a generic *FileSensor* component that reads simulated data from a designer specified input file for each sensor.

One of the advantages of the modularity supported by ATLeS-SN is the ability to utilize simplistic models and methodologies within early design phases. In developing the building monitor application, we developed basic implementations for the *PhysChannel* and *NetStack* components. Rather than utilize a dynamic routing scheme for routing packets, we developed an *Airspace* component for the *PhysChannel* that creates a spanning tree during the initialization of the simulation that is then utilized by the *NetStack* components when routing packets. This initial abstraction allowed us to reduce both simulation and development time, as our current focus is not on modeling and implementing a dynamic routing protocol. However, the ATLeS-SN framework

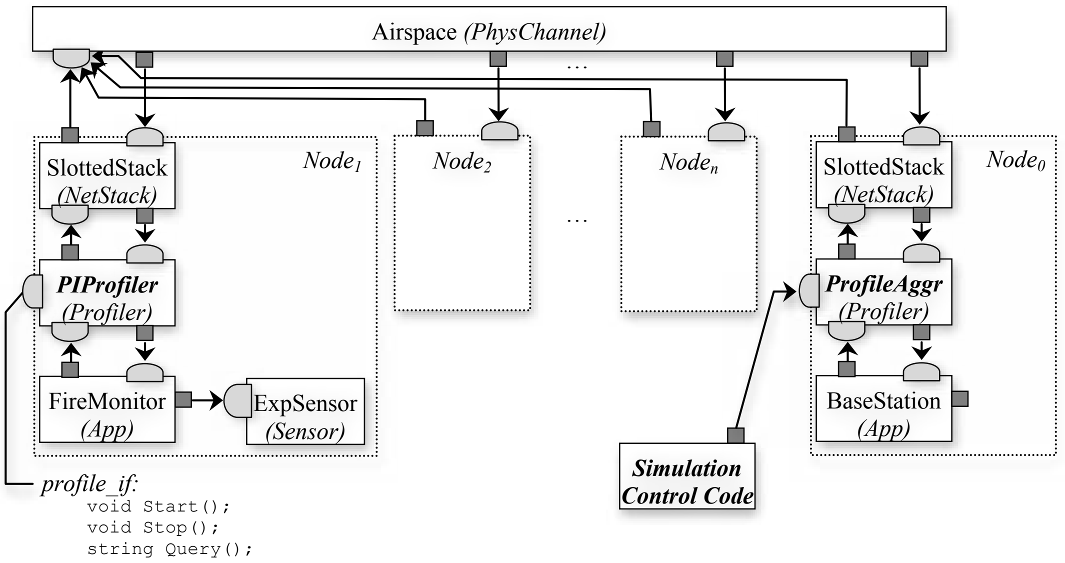


Figure 4. Forest Fire Detection and Propagation Tracking sensor network application highlighting the *Profiler* interface *(profile_if)*, the addition of a packet insertion profiler *(PIProfiler)* within each sensor node, a profile aggregator *(ProfileAggr)* incorporated within the base station node, and the *Simulation Control Code* needed to control the profiling and analyze profile results.

would allow a designer to incorporate this functionality at a later time if necessary. In addition, we developed a *SlottedStack* component for the *NetStack* within each sensor node that implements a time-multiplexed network protocol with schedule arbitration.

As sensor nodes are not equipped with GPS, each node must synchronize its time with the base station on startup. If the node cannot synchronize with the base station to determine the current time and operating mode, those nodes will still monitor activity and attempt to transmit messages whenever movement is detected. Upon receiving a synchronization request, the base station node broadcasts to all nodes the current time along with the current operating mode.

During the normal monitoring mode, if the standard deviation of the last four samples is greater than a user defined threshold, the sensor node will transmit a message to the base station indicating movement was detected along with the time at which the movement was detected. Otherwise, nodes mostly remain silent.

When operating in the low-power mode, sensor nodes turn off the radio and sensors for prolonged durations to reduce power consumption. However, each node will wake up once per hour to communicate with the base station to verify the node is still functional as well as respond to requests from the base station to change operating modes.

The base station node receives movement messages from all nodes and determines if the movement is within acceptable designer specified ranges. The acceptable movement ranges can be defined both as a maximum level of movement within specific rooms or as a maximum level of movement across multiple rooms. For example, a night watchman mode could be defined where low levels of movement by a patrolling guard within a single room is acceptable, but too much movement within one room or movement within more than one room is not. If this movement threshold is exceeded, the base station can sound an alarm, notify appropriate personnel, or take other necessary actions.

The initial application development and testing for the building monitor application with eight sensors only required two man-hours. This rapid application development was enabled by the high-level of abstraction and modularity provided by ATLeS-SN. However, we note the development time does not include the initial learning curve a designer would face when first using ATLeS-SN – although we anticipate this learning curve to be marginal.

We simulated the building monitor application using a 3.2 GHz quad-core Xeon server running Red Hat Linux 2.6. Simulating the initial eight-node system required only 15 seconds for each hour of simulated time. In addition, we evaluated the simulation speed of ATLeS-SN for sensor networks with increasing number of nodes. Figure 5 presents the simulation execution time for one hour of simulated time with 2 to 512 sensor nodes. As the number of nodes within the system increases, the simulation time increases linearly. With 512 sensor nodes, a one-hour simulation of the building monitor application required just over 21 minutes.

B. Forest Fire Detection and Propagation Tracking Application

The forest fire detection and propagation tracking application is designed to detect and track fires in remote forest areas by observing and monitoring extreme temperatures. Figure 4 presents an overview of the forest fire detection and propagation tracking application implemented within ATLeS-SN. This application consists of a single base station node and several temperature sensor nodes. In the base station node, the *App* component is implemented as a *BaseStation* component. The *App* components of the remaining sensor nodes are implemented as fire monitor components, called *FireMonitor*.

During the normal fire detection operation, the sensor nodes within the system will periodically monitor temperatures and transmit the temperature readings every five minutes to the base station. In the event that a node detects an elevated temperature for the previous two temperature samples, that node issues an alert to nearby nodes and transitions to a fire-tracking mode. Whenever a node receives an alert message from a nearby node, the node will also enter the fire-tracking mode to ensure that the fire's propagation can be efficiently tracked with reduced latency. In the

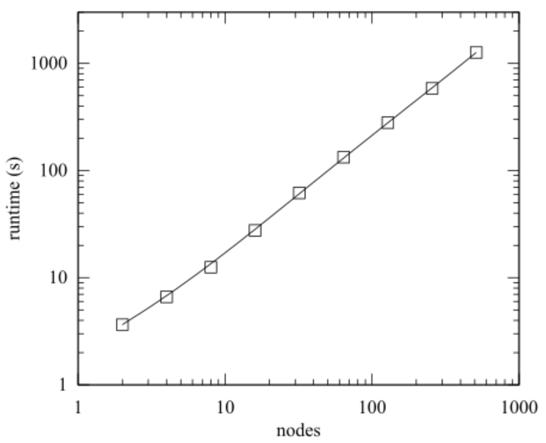


Figure 5. ATLeS-SN simulation ex ecution for one hour of simulated time using building monitor sensor network application consisting of 2 to 512 nodes.

fire-tracking mode, each node will sample and report the temperature every ten seconds.

The base station node aggregates the reported temperature sensor, displays the reported data with appropriate timestamps, and issues alerts whenever a node enters the fire-tracking mode or a sensor node suffers an abrupt node failure.

The *Sensor* components within each sensor node are implemented as *ExpSensor* components that model the increase in temperature of a given sensor node as an exponential function based on the node's physical location – or X and Y coordinates – and the current time. The *ExpSensor* component initially returns a semi-constant nominal temperature reading until a specific time dependent on the node's X and Y coordinates, after which the temperature increases exponentially. Using the fire simulation data presented in [12], we model a forest fire with linear spatial propagation, i.e. we model a forest fire starting at the origin (0,0) for which a node detects no fire until $t = 5\sqrt{x^2 + y^2}$, after which its temperature readings begins to increase. Note that the *ExpSensor* component not only models a physical temperature sensor but also models the physical environment that affects the temperature at each node. While the exponential increase in temperature due to a forest fire is not an accurate model of a real fire's actual temperatures, the spreading of the fire based on a node's physical location could be accurately modeled from historical forest fire data. Alternatively, more accurate forest fire propagation models, such as [8], could be efficiently integrated into the *Sensor* components.

For the forest fire detection and propagation tracking application, we extended the *SlottedStack* and *Airspace* components to provide a basic priority-based packet queue, so that the base station could quench packet floods and override sampling rates for specific nodes. A packet priority field was added to each *Packet*. A packet's priority is based on both the direction a packet is being transmitted and the packet's message. For instance, packets being transmitted downstream from the base node are less frequent but have a higher priority. In addition, sensor data is not as important as an alert message and therefore has a lower priority. We modified the *SlottedStack* by adding multiple packet queues, and we implemented a simple arbitration scheme to select which queue would send the next packet.

The initial application development and testing for the forest fire detection and propagation tracking application required approximately five man-hours.

We further utilized the ATLeS-SN framework to integrate a profiler within the forest fire detection and propagation tracking application in order to analyze various application statistics. Profiling and analysis of a sensor network application is typically necessary to enable designers to optimize the sensor network by adjusting node operating parameters, e.g. sampling rates or transmission power, or allow them to modify network level protocols to increase reliability or fault tolerance.

While the simulation environment provides simple methods to externally monitor sensor network activity, we directly incorporated a *Profiler* component within each sensor node that provides a dedicated profiling interface, *profile_if*, allowing a designer to profile individual nodes in isolation or to profile the ent ire system from the aggregated profile data available at the base station. Using other simulation approaches, incorporating profiler support would require direct modifications to the target application, which could potentially affect the execution behavior of the target application. With ATLeS-SN, the *Profiler* component can be easily incorporated as a shim between the *App* and *NetStack* components without requiring any modifications to the existing code.

The base *Profiler* component defines a single *profile_if* consisting of three transactions, `Start`, `Stop`, and `Query`, where the `Start` and `Stop` transactions enable and disable the profiling operation across the entire system, and `Query` returns the current profile data.

As highlighted in Figure 4, for the forest fire detection and propagation tracking application, we implemented a packet insertion profiler, *PIProfiler*, within each sensor node and a profile aggregator, *ProfileAggr*, within the base station node. When profiling is disabled, the *PIProfiler* component simply forwards packets between the *App* and *NetStack* without modification. When profiling is enabled, the *PIProfiler* will transmit additional profiling-specific packets.

The base station node's *Profiler* is implemented as a profiler aggregator that intercepts and extracts the transmitted profile packets. In addition, the base station *ProfileAggr* component transmits profiling-specific packets to all nodes when profiling is enabled or disabled by the `Start` and `Stop` transactions. These profiling-specific packets are intercepted by the *PIProfiler* within each sensor node, thereby completely separating the profiling functionality from the *App* components.

This profiling strategy offers several advantages. First, the packet insertion profiler allows designers to efficiently monitor the system within the simulation environment without affecting the sensor network application simulation and without requiring changes to any components within the original application implementation. Secondly, this profiler implementation could be later utilized within a deployed system to allow designers to monitor the health of the sensor network or optimize the sensor network operation at runtime. If profiling will be incorporated within a deployed system, a designer must carefully balance the need for accurate – and up-to-date – profiling information with the overhead of transmitting and processing profiling packets.

For the forest fire detection and propagation tracking application, we utilized the *Profiler* components to monitor common node-level operating statistics, including packets received, packets transmitted, computation events, battery voltage, and local link quality. Furthermore, we utilized this profile data to model and analyze the power consumption and battery lifetime of physical implementation of the application using Crossbow IRIS sensor nodes. We implemented the forest fire detection and propagation tracking application on an IRIS system and analyzed the power consumption of radio communication and temperature sampling. The power consumption for radio communication consists of the power consumed for transmission and reception, including packet transmissions, packet receptions, and packet retransmissions. The power consumption of temperature sampling includes interfacing with the physical temperature sensor along with processing the temperature data.

With the power consumption data and profiling information, we utilized the ATLeS-SN simulator to monitor and analyze the battery discharge of each sensor node in order to estimate the lifetime of a deployed system within the current application specification. For a nine node system, our analysis estimates that the sensor network would have a lifetime of only one week during the normal operating mode when no fire is detected and a lifetime of six hours once the alert mode is activated. The total simulation runtime required for these scenarios was less than 40 minutes.

During an actual forest fire, a six hour lifetime for a sensor node should be appropriate enough to track a fire through the area monitored by a single node – or at least until the node is destroyed in the fire. However, our analysis indicates that the current implementation and/or platform are not suitable because the nodes' lifetime during normal operation is inadequate to last even for a single fire season. Nevertheless, ATLeS-SN quickly allowed us to identify this shortfall in the early design phases in order for future modifications and optimizations to be applied.

V. Conclusions and Future Work

The Arizona Transaction-Level Simulator for Sensor Networks provides a modular framework for modeling and simulating components of a sensor network. This flexible approach allows designers to focus on modeling, simulating, analyzing, and optimizing specific sensor network components without requiring a detailed time accurate implementation across all levels while retaining the capability to holistically analyze an entire sensor network. We demonstrated the efficiency of ATLeS-SN to quickly design and test sensor network applications through two case studies, illustrating how transaction level modeling can facilitate sensor network simulation and simplify incorporation of additional functionality.

While ATLeS-SN provides a foundation upon which designers can integrate advanced functionality, our immediate future work will focus on extending the ATLeS-SN framework to provide *NetStack* and *PhysChannel* implementation using standard networking protocols available within existing sensor networks along with incorporating more advanced signal propagation models. Future work also should include developing a framework to directly support modeling the physical environment as a separate component and providing a robust interface with which *Sensor* components can interact. By modeling the environment as an independent but synchronized component, existing simulation tools for modeling physical stimuli such as fire propagation models can be efficiently incorporated into our framework. As many sensor networks include actuators such as LEDs, speakers, and motors, we plan to extend the sensor node model to include an *Actuator* component that designers can utilize to model and test various output devices for physical sensor nodes. These actuators would also interface with the physical environment model in order to allow for analyzing systems with feedback loops. Finally, in order to provide a foundation upon which other researcher can leverage the benefits of ATLeS-SN, we anticipate that a basic library of common sensor network components are needed, including but not limited to protocols, sensors, actuators, and processor models.

Acknowledgement

This research was supported by the National Science Foundation (CNS-0834102 and CNS-0834080).

References

[1] Ben Atitallah, R., S. Niar, S. Meftali, J.-L. Dekeyser. An MPSoC Performance Estimation Framework Using Transaction Level Modeling. *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 525-533, 2007.

[2] Cai, L., D. Gajski. Transaction Level Modeling: An Overview." *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 19-24, 2003.

[3] Crossbow Technology. IRIS Wireless Modules. http://www.xbow.com, 2009.

[4] Demaille, A., S. Peyronnet, and B. Sigoure. Modeling of Sensor Networks using XRM. *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pp. 271-276, 2006.

[5] Donlin, A. Transaction Level Modeling: Flows and Use Models. *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 75-80, 2004.

[6] Egli, J. J. Radio Propagation above 40 MC over Irregular Terrain. *Proceedings of the IRE*, Vol. 45, No. 10, pp. 1383-1391, 1957.

[7] Gajski, D., J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao. SpecC: Specification Language and Design Methodology, *Kluwer Academic Publishers*, 2000.

[8] Jindal, A., K. Psounis. Modeling Spatially Correlated Data in Sensor Networks. *ACM Trans.on Sensor Networks (TOSN)*, Vol. 2, No. 4, pp. 466-499, 2006.

[9] Keshav, S. REAL: A Network Simulator. Technical Report 88/472*, UC Berkeley*, 1988.

[10] Kumar, U., A. Ranjan, V. Jalan, P. Mundra, P. Ranjan. CENSE: A Prototype for Modular Sensor Network Testbed. *National Conference on Embedded Systems, Mumbai, India*, 2006.

[11] Levis, P., N. Lee, M. Welsh, D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. *International Conference on Embedded Networked Sensor Systems (SenSys)*, pp. 126-137, 2003.

[12] Lin, R., J. Reisner, J. Colman, J. Winterkamp. Studying Wildfire Behavior using FIRETEC. International Journal of Wildland Fire, Vol. 11, No. 4, pp. 233-246, 2002.

[13] The Network Simulator – NS-2. http://www.isi.edu/nsnam/ns/, 2009.

[14] Open SystemC Initiative (OSCI). SystemC Language. http://www.systemc.org/, 2009.

[15] Sachdeva, G., R. Dömer, P. Chou. System Modeling: a Case Study on a Wireless Sensor Network. Technical Report CECS-TR-05-12, *UC Irvine*, 2005.

[16] Shnayder, V., M. Hempstead, B. Chen, G. Allen, M. Welsh. Simulating the Power Consumption of Large-Scale Sensor Network Applications. *International Conference on Embedded Networked Sensor Systems (SenSys)*, pp. 188-200, 2004.

[17] Sinha, A., A. Chandrakasan. Dynamic Power Management in Wireless Sensor Networks. *IEEE Design & Test of Computers*, Vol. 18, No. 2, pp. 62-74, 2001.

[18] Szewczyk, R., A. Mainwaring, J. Polastre, J. Anderson, D. culler. An Analysis of a Large Scale Habitat Monitoring Application. *International Conference on Embedded Networked Sensor Systems (SenSys)*, pp. 214-226, 2004.

[19] Virk, K., K. Hansen, J. Madsen. System-Level Modeling of Wireless Integrated Sensor Networks. International Symposium on *System-on-Chip (SOC),* pp. 179-182, 2005.