A Comparison of Sensor Network Simulators

Michael J. Pflug

April 8, 2004

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the Degree of
MASTER OF COMPUTER SCIENCE

Approved:

Professor Boleslaw Szymanski

Thesis Advisor

Rensselaer Polytechnic Institute
Troy, New York
May 2004

Contents

1 Introduction and Historical Review **1**

2 Flooding **3**

2.1 Flooding Algorithm 3

2.2 Flooding and the OSI Model 5

3 The NS-2 Simulator **7**

3.1 Split-programming Model 7

3.2 Memory Management 9

3.3 Learnability 10

4 SENSE **11**

4.1 Component-oriented model 12

4.1.1 Makeup of a Component 13

4.1.2 Types of Components 14

4.1.3 Putting it all Together 15

4.2 Memory Management 17

4.3 Learnability 18

5 Simulations and Results **19**

5.1 NS-2 Simulations 20

5.2 SENSE Simulations 20

5.3 Results 21

6 Discussion and Conclusions **27**

7 Future Work **28**

8 Appendix A: floodingrun.tcl **33**

9 Appendix B: sim_flooding.cpp **44**

10 Appendix C: immobile.h **59**

11 Appendix D: Traffic Generation Script **62**

12 Appendix E: Topology Generation Script **69**

List of Figures

1 Flooding within the OSI model 6

2 Connected Components 12

3 Sensor constructed of various components 16

4 Packet construction in SENSE 17

5 Execution time as simulated time increases 21

6 Event processing rate as simulated time increases 22

7 Memory usage as simulated time increases 23

8 Execution time as the number of nodes increases. 24

9 Number of events as the number of nodes increases. 24

10 Event processing rate as the number of nodes increases 25

11 Number of packets sent and recieved as size of the simulation increases. 26

12 Memory usage with revised NS-2 26

13 Execution time with revised NS-2. 27

Abstract

A new sensor network simulator, SENSE, has recently been released for public use. The developers of SENSE have designed their simulator with the primary goals being extensibility, scalability, and reusability and achieved them primarily due to the component-oriented nature of the simulator. In realizing their vision, the SENSE team has created a new simulator that outperforms existing simulators in both ease of use and execution time. In this paper we take an in-depth look at SENSE and, using an optimized flooding simulation, compare it to perhaps the most popular network simulator today, NS-2.

1 Introduction and Historical Review

A sensor network consists of many wireless, possibly mobile, nodes that communicate with one other without the help of some fixed infrastructure [24]. The nodes communicate on a given radio frequency directly with neighbors, and indirectly through other nodes as necessary, passing gathered data efficiently from one side of the network to another.

Presently, sensor networks are still being developed by researchers who are at the same time constantly discovering new applications. Sensor networks are currently being used experimentally for everything from data collection in remote areas [13, 3, 23] to the monitoring of patients in hospitals [1] to helping to educate our children [20]. Future applications could include uses such as military battlefield surveillance in which sensors would be dropped from an unmanned plane high above enemy territory [14]. More mainstream uses could include toxin and smoke detection in the walls of homes where sensors would automatically alert authorities the instant abnormal levels were detected. These applications are still years away however, as almost all current work with sensor networks is only experimental, partially due to the currently high cost and short battery life of the sensors themselves.

Recently sensor networks have become more and more prominent, creating many new opportunities for research in this area. In turn, researchers have called for a simulation toolkit to aid in their investigations. Researchers turn to simulation because physical experiments are costly to set up and difficult to debug, especially for large networks of hundreds of nodes scattered over a square kilometer of terrain. Provided the right simulator however, researchers can overcome these difficulties, setting up and executing simulations in a fraction of the time for a fraction of the cost while easily collecting comprehensive experimental data.

As a first attempt at sensor network simulation additions were made to existing simulators to incorporate sensor networks into their modeling libraries. NS-2 [17] was among the first to add mobile networking capabilities [15] to its library. The original additions were sufficient for the time; allowing users to create wireless topologies, adding node movement, incorporating the 802.11 MAC protocol, new routing agents, and new methods of data collection. While these additions were an important step in mobile ad-hoc networks, NS-2 was not designed with such networks in mind causing these new extensions to be somewhat slow and cumbersome. More so, NS-2 is written in an object-oriented approach which causes ns2 modules to contain excess interdependencies leading to run-time overhead and creating a steep learning curve.

When these first additions were made to NS-2 mobile network simulation was an emerging, unknown field. As the technology being simulated has advanced, so to must we advance our simulation methods. Unsatisfied with the capabilities of NS-2, SENSE [19] was developed as strictly mobile network simulator. Moreover, SENSE was created with a focus on a component-oriented design [7, 8] with the expectation that efficiency would be gained by lowering execution speed and facilitating the end-user experience. This design allows users to figuratively remove a component from the simulation engine, modify it to their specifications, and plug it back in without having to worry about module dependencies. Overall, component-oriented design minimizes the learning curve allowing researchers to run simulations and model their own components in a fraction of the time. The component-oriented design of SENSE gives researchers a new tool, letting them run simulations faster and more efficiently, thereby facilitating new discoveries and advancements in the field.

2 Flooding

The broadcasting of packets within a network is perhaps the most fundamental way of sharing information between nodes. The flooding protocol is not only relatively simple to understand and implement, but is also used as a building block for many more advanced routing protocols (eg directed diffusion [12]). The efficiency of the flooding simulation will be transfered onto these more advanced protocols. The percolating effect of broadcasting makes the flooding simulation a good representative for comparison tests among different simulators.

2.1 Flooding Algorithm

Many different levels of optimization of the flooding algorithm have been developed by various researchers [21, 22]. The original flooding algorithm involves nodes rebroadcasting every packet they receive without regard as to whether or not the packet has previously been viewed. While this may make for easy implementation, clearly it is a very inefficient approach. In answer, several different levels of optimization have been developed. We have chosen to implement a slightly optimized version of the flooding algorithm in which a list of seen packets are stored within each node. Each node then only rebroadcasts a packet if the packet has not previously been viewed by the node. This decreases the amount of traffic on the network leading to a quicker responce time and fewer collisions during transmission.

The flooding algorithm itself is straightforward. When a node N broadcasts a packet, the packet is sent to every other node within range of its signal (also known as the node's neighborhood). Each of the receiving nodes check their records, usually in the form of a hash table, as to whether they have

Flooding: Broadcast

1: To broadcast packet P from node N
2: Send P to all neighbors

Flooding: Receiving

1: Node M upon receiving packet P
2: **if** M has not received packet P before **then**
3: Add P to table of viewed packets
4: Deliver the packet to the application
5: Wait some random interval
6: Broadcast packet P to all neighbors
7: **else**
8: Drop packet P
9: **end if**

seen the packet before. If the packet has been seen already it is dropped, otherwise the packet is marked as viewed and added to the hash table. Finally, after a random wait time to minimize collisions in the physical layer, the packet is broadcasted to all neighbors and the process begins again. The algorithm can be seen in pseudo-code below.

The flooding algorithm has one obvious drawback, the overhead of duplicate messages being pushed through the physical layer. In ideal routing conditions, each node receives the packet only once. In flooding however, it is very likely that a node will receive the same packet many times causing numerous collisions in the physical layer and forcing unnecessary calculations to be performed by the nodes. Excess collisions also occur due to nodes sending packets across the channel at the same time. Preventative methods such as the random wait interval are in place to minimize simulataneous node

activity.

Extra calculations not only slow down the node's sending and receiving of messages but also use up valuable battery life. While there are additional optimizations that minimize this overhead, we will run our simulations with the standard algorithm given above in order to keep the simulations as fundamental and dynamic as possible.

2.2 Flooding and the OSI Model

The architecture of both NS-2 and SENSE correspond very well with a simplified version of the Open System Interconnection (OSI) model. The OSI model is a standard that defines a networking framework broken up into layers. Data is passed from layer to layer down the stack with the layers often putting a wrapper on the existing packet as it is passed. When the information reaches the destination the data travels back up the stack and the headers are unwrapped until the packet's data arrives at the correct application.

The standard OSI model is composed of seven layers: application, presentation, session, transport, network, data link, and physical. In simulations however, layers are often combined for simplification, focusing on the most pertinent for the task at hand. Network simulations focus on the bottom layers of the stack so we end up generalizing the application, presentation, session, and transport layers all down into a broad application layer. This simplification is allowed because once the packet is delivered, the network no longer concerns itself with the packet, allowing individual hosts to assure the packet reaches it's destination. The lower levels of the OSI model (physical, link, and network) are kept seperate to maintain a sufficient level of detail.

The flooding algorithm, like other routing protocols, operates within the

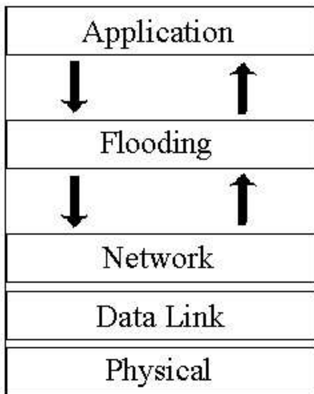

Figure 1: Flooding within the OSI model

network layer of the OSI model (Figure 1). The network layer's function is to move a packet from the source to the destination, through intermediate nodes if necessary. To send, the network layer wraps up each packet passed to it by the application layer and addresses one to each node in its neighborhood. On receiving a packet from the link layer, the network layer checks that it is the addressee, unwraps the packet and passes it up to the application layer for processing.

To represent the link layer we use the 802.11 wireless protocol as specified by IEEE standards [11]. The purpose of the link layer is to transform the physical channel into a smoothly operating, message transferring operation. This is generally accomplished by breaking up packets into manageable frames and transmitting them across the channel by passing the string of bits to the physical layer. The link layer also handles any problems that may arise during a transmission, such as corrupted frames and collisions and is responsible for rebroadcasting if necessary.

The physical layer is where the packet is actually transfered from one

node to another via a radio signal operating on some predefined frequency. It receives a string of bits from the link layer and, without regard to its meaning, transmits them sequentially. In a wired network, the physical layer would consist of the actual wires, CAT5 cable for example, but within a sensor network the physical layer is the part of the sensor used to send and receive radio signals.

3 The NS-2 Simulator

The Network Simulator (NS-2) is a discrete event simulator created for aiding network research [17]. Work began on NS-2 in 1989 out of the VINT simulator project and has since evolved into possibly the most widely used network simulator. Although originating from the University of California at Berkeley, the simulator team has taken contributions from across the world as researchers developed added new protocols and other features. Of particular interest is the Monarch project out of Carnegie Mellon University [15] which enabled researchers to simulate wireless networks for the first time. The new wireless model created new link and physical layers specific to wireless networks as well as implementing node movement and some of the more common routing protocols (DSDV, DSR, TORA, and AODV). Since these additions, additional wireless simulation protocols and functionality for sensor networks have been contributed. These additions have made NS-2 possibly the best sensor network simulator currently available.

3.1 Split-programming Model

Instead of being written in a single language, simulations in NS-2 generally consist of two pieces: a simulation script written in Tcl (Tool Command

Language) and the actual C++ kernel behind the simulation. Additionally, the Tcl script can be set up to read in a topology and traffic file. NS-2 simulations take place in a rectangular plane known as the topology. The topology file contains the initial coordinates of each node as well as destination and speed if movement is desired. Traffic files contain a description of initial communications between nodes including the time the interest will occur and the size of the interest.

The "front-end" simulation script is written in Tcl due to it's dynamic nature [2]. Being written in a scripting language, the simulation scripts can be easily refined as the research task at hand unfolds, allowing researchers to make quick modifications to parameters, objects, and traffic specifications. Not having to wait for the simulator to recompile is a necessity for most researchers, especially when working with a simulator the size of NS-2. The C++ part of NS-2 contains the code for event processing, for protocol implementation, and packet manipulation. Essentially, any relatively static, black box portion of the simulator is better represented in a compiled language due to the performance increase. One downside to this is that when a user desires to modify these portions of the simulators, the process is lengthy due to recompilation time.

As mentioned above, NS-2 is an object oriented simulator. The main drawback to working with objects is that they often become too inflexible and cannot be easily reconfigured to work for alternative uses [5, 6]. For example, consider objects A and B in some NS-2 environment, with object B allowing A access to call its methods. A and B are now essentially combined into one object [5] due to the reliance of A on the existence of B. In the simulation A and B were originally created for, this situation does not create problems, however for other situations in which a developer might want to

make use of these objects, the dependency of A on B is now buried within the original source code. In order to reuse A correctly in another simulation we need to also include the code for object B, however this dependency is not visible to a developer by looking only at A or B and could lead to confusion when designing new implementations. More importantly, A and B are linked in such a way that if a developer writes a new object, C, that is somehow better for his simulation, he cannot simply plug C in for B because this would break object A's functionality. If the developer wants to create a replacement C for B, they must make C a subclass of B, which would allow A to still function as intended. This design option can work, but limits the programmer and often creates extra overhead by needing to include object B even though a new solution, C, has been developed [5].

3.2 Memory Management

The memory requirements of NS-2 are extremely high for a network simulator. The high cost is partially due to the simple packet management system the developers chose to implement. Every time a packet is modified or passed, even if the packet has already been created by another node and just needs to be passed along, a new copy is created for the next layer or node. These copies are kept around until the end of the simulation and, as one would expect, over time they build up and greatly increase the total amount of memory taken by the program. Another reason NS-2 has high memory requirements is due to the way C++ and Tcl are integrated in the simulator. The integration causes many C++ objects to export a very large amount of state variables to Tcl [16]. These variables require large amounts of memory, adding to the problems with the packet management system.

The large memory requirements of NS-2 do not effect the execution time

of the simulation to noticeably until the capacity of physical memory is exhausted and we have to start working out of a swap file. The multiple magnitude increase in access time from RAM to disk brings the execution of the simulation down to an unbearably slow rate at which time it becomes almost unfeasible to run large scale simulations for any length of time.

3.3 Learnability

Overall, the Tcl/C++ code split in NS-2 can be productive. A clear separation is made between the simulation environment used for conducting simulations (Tcl) and simulator design and extension (C++). The original design was implemented with the idea that those researchers interested in using pre-existing tools could use the simulator as a black-box while those interested in designing new protocols and investigating a little bit deeper could modify the C++ code. This approach seems effective at first glance, and in fact works well for those who clearly fall into one of the two categories.

Remember, NS-2 was originally designed for traditional networks which have been around long enough to have established a set of common protocols (TCP, 802.11, Ethernet, etc). Researchers using NS-2 for these purposes will have a relatively easy time because almost everything they could want to simulate is already implemented. Sensor networks on the other hand are a relatively new idea with no dominant protocols as of yet, mostly due to the fact that sensor networks are most often designed with a specific purpose in mind and it is unlikely that one protocol will be able to meet the needs of so many different situations [19]. Therefore, when working with sensor networks, sooner or later the need to augment the source code due to lack of necessary protocols will arise. At this point, the line between the two types of users starts to blur and the jump from user to developer in NS-2 can be

dramatic. For the journeyman NS-2 user, struggling through the mountains of source code, can be a daunting and time consuming task. Because NS-2 is no longer a completely in-house project, there are no standards, often little or no comments within the code, and online documentation is spotty at best when it comes to contributed code. The lack of a good documentation system scares away many users that want to venture further into the source.

On top of the documentation dilemma, adding new functionality to the simulator is also complicated. Usually this will involve adding C++ code as well as updating Tcl configuration files so NS-2 can identify the new code and integrate its uses. The C++ code must be written carefully due to the object-oriented nature of the simulator. Objects need to interact with one another during a simulation, thus when integrating new functionality one must make sure that other parts of the simulator are not broken with your new addition. These factors combine for a steep learning curve when developing new NS-2 functionality. In turn, the excess time spent developing the new tools greatly increases the time it takes to complete a project and overall slows down the researchers' progress.

4 SENSE

The Sensor Network Simulator and Emulator (SENSE) [19] began development in 2003 at the Rensselaer Polytechnic Institute's Center of Pervasive Computing and Networking. The project was begun in order to provide researchers with a simulator designed specifically for sensor networks, providing an alternative to other simulators who had added this capability as an afterthought. The SENSE team hoped their simulator with its component oriented design focus would gain efficiency by lowering execution speed and

Figure 2: Connected Components

facilitating the end user experience.

SENSE has its roots in the Component Oriented Simulation Toolkit (COST) [8, 10], a C++ class library designed to aid researchers in discrete event simulation. COST consists of components, each themselves containing interfaces used to connect to other components. COST and SENSE are both based on the component-oriented model of simulation. In addition to the required components, a simulation contains a simulation engine used to synchronize all the components and drive the simulation. Each component is modeled via an event-oriented view, in other words, each component has an event-handler which performs the correct task when the corresponding event arrives at the component. SENSE has been built on top of COST by creating new components developed solely for the purpose of sensor network simulation such as a linear battery model, flooding, AODV, 802.11, and more [19]. This extension is largely made possible due to COST's component-oriented nature and because COST is heavily based on C++ template libraries, allowing for a maximal level of reusability while retaining efficiency.

4.1 Component-oriented model

From the view of the simulation programmer, a designing a simulation is nothing more than programming with time. In component-oriented software engineering, components, tied together through inports and outports (Figure

2), react to events presided over by an event scheduler. Events are split into two types, synchronous events arriving at the components inports and asynchronous events which are associated with a timer that lays between the component and the simulation engine [10]. Events are processed in a time-sequential manner, simulating the passing of time within the event scheduler. In this way, it is a natural fit to take component-oriented programming and apply it to simulation design.

4.1.1 Makeup of a Component

Using the component-oriented design, a simulation consists of a number of components, preexisting or newly developed, working independently to form the system. The major benefits from this design follow from this idea. First, each component is itself a small autonomous system working on its own, unaware of other components within the system. Secondly, it follows that components are easily reusable between systems due to the fact that there is no interdependence [6]. Any component can be removed from a simulation and plugged into another, providing the interfaces are compatible, and work without the developer worrying about hidden dependencies.

A component itself is made up of three parts, inports, outports, and possibly a timer. To send an event, the component places the event on the correct outport, received events arrive at inports and are handled by an event handler. To view this in a function-oriented way, view the inports as functions and the outports as function pointers. An event is the information needed by the function call such as the address in memory and the parameters. The returned value of the function is akin to the event acknowledgment [6]. Note that in this functionated view, one component cannot call another components inport, information must be sent through the outport. The

components are completely independent and known nothing about the world they are in. Continuing with the function association, like functions, ports have types. Each port is typed and can only handle the assigned type of event. Additionally, an inport and outport can be connected together only if they are of the same type, as if putting together a puzzle. Unlike a puzzle however, a single inport can be connected to multiple outports, and likewise an outport can be connected to by multiple inports.

So how does a component with its inports and outports differ from an object with member functions? In short, in an object-oriented paradigm, it is generally assumed that an object can directly call another objects methods. Contrarily, in a component-oriented system, all connections between components are specified with ports. This keeps all components completely autonomous and allows for maximum reuseability.

4.1.2 Types of Components

Components are further classified into three temporal types: time-independent, time-aware, and autonomous. Time-independent components, as their name implies, have no idea they are in a time governed environment. Because they have no knowledge of time, a time-independent component only generates events based on events received. A typical time-independent component is a FIFO queue in which an event enters an inport and remains in the queue until a release event enters the component via another inport. At this time an event will be generated at an outport freeing the item from the queue. Time-independent components are perhaps the most versatile of any components as they can be easily moved from a time oriented environment to a non-time driven system without requiring internal changes [10].

Time-aware components generally represent the main parts of a simula-

tion. These components cannot advance the system time themselves, but, based on events received at inports, can create and send events scheduled for a time somewhere in the future. Time-aware components are used to represent system components such as servers which would recieve an event at some time, and generate a release event at a future time based on the function the server performs. A CPU in an M/M/1 system would be represented as a time-aware component, generating events based on processing time of the event passed in.

Finally, we come to autonomous components, those components that can themselves manipulate the simulated time. Like time-aware components, autonomous components take in events at a certain time, but unlike a time-aware component, an autonomous component advances the simulated time itself rather than just scheduling a future event. An autonomous component could represent an encapsulation of an entire system in which a user's job enters the system at some time and then leaves the system, advancing the simulation clock the amount of time the job spent in the system.

4.1.3 Putting it all Together

Once one has a set of components designed, they must be combined using a simulation engine. The simulation engine synchronizes any number of components into one massive autonomous component with no inports or outports and also maintains the simulation clock. Different types of simulation engines tie together different types of components. A sequential engine links together time-independent and time-aware components into a single autonomous component. A parallel engine links together multiple systems represented by autonomous components. Many parallel engines can also be linked together to form an even larger autonomous component.

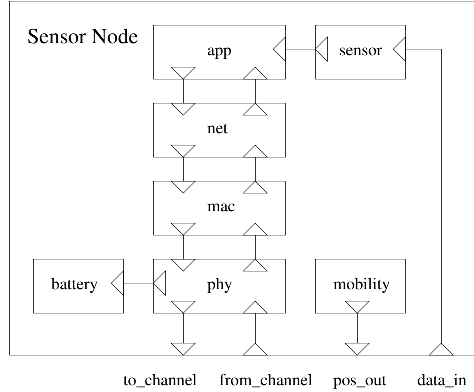

Figure 3: Sensor constructed of various components

In order for the components to communicate to the simulation engine, each component must have an interface specifically for this purpose. The type of the interface that the component uses to link to the simulation engine is known as the components simulation type. When transferring a component from one simulation to another, above all else the simulation types must match. This interface allows the simulation engine to keep track of everything that occurs within each component, allowing the engine to synchronize all the components.

Simulation time is usually implemented as a real number, increasing as the simulation progresses. Each type of component has its own methods of placing a timestamp on an event. Time-independent components do not have any way of advancing the simulation and thus simply copy timestamps from the received when a new event is triggered. A time-aware event uses incremental timestamps, setting the time for a future event to occur based on the function of the component. An autonomous component uses what are known as comparable timestamps, which are intuitively compared to determine what order the events should be processed [6].

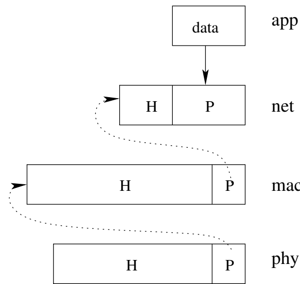

Figure 4: Packet construction in SENSE

Once all the components and a simulation engine have been defined, they are combined to complete a system. In figure 3 we see vaious components combined to form a sensor node. The sensor node then becomes component itself with its own in and outports for channel communication and data entry.

4.2 Memory Management

In order to avoid the massive memory requirements of NS-2, SENSE takes a more efficient, but more difficult to implement approach. Instead of passing copies of a packet's data blindly from layer to layer like NS-2 does, SENSE developers decided to only create the packet data once and, if the data is bigger then a pointer, pass pointers to the packet instead. As each layer adds its header to the packet, only a pointer is passed to the next layer, which adds its header and passes the pointer again (Figure 4.

Additionally, when a packet is broadcast by a node, it is almost always received by multiple nodes. If copies of the entire packet were passed, each node would keep its own copy in memory, now with this advanced system, each node only gets a pointer to the packet. When a node acquires a packet, it increases the reference counter of the packet to let the system know the packet is in use. When a node releases a packet, the counter is decremented.

If no other nodes are using the packet (i.e. the reference counter is at zero) then the packet can be deleted from memory.

Overall, the efficient packet handling of SENSE results in much cheaper memory requirements allowing the user to run large scale simulations for extended periods of time without having to wait for disk access because the simulation spilled over into virtual memory.

4.3 Learnability

SENSE was not only designed to speed up the execution time of simulations but also to decrease the steepness of the learning curve found in other simulators. The first step towards this goal was improving the level of documentation, especially online, when compared to existing options. SENSE has made available a comprehensive online documentation system detailing the purpose of each class (and it's methods) not only of SENSE but for COST as well. The developers of SENSE also put together step-by-step tutorials containing how-to's for users of all levels. New users are introduced to the process of creating a simulation. They are shown how to design a node component, then how to combine these nodes into a network, then finally how to configure and run the simulation. Intermediate users are shown how to write more advanced components such as a flooding simulation and a constant bit rate (CBR) component. More advanced users are taken deeper into the simulation, given examples of the inner workings of COST, thereby allowing the user to build their simulation from the ground up.

In addition to the excellent documentation, the component-oriented design makes it easier for the user to understand and their simulations. This design was chosen in part because a simulation can be thought of as many small pieces working together to create a result greater than the sum of its

parts. The user can concentrate on first created the individual pieces and having them all work individually before integrating them to create the simulation. This approach leads to easier debugging for the user and overall decreases the amount of time spent creating a simulation. A component-oriented approach also easily allows users to adopt existing simulations to their own research. If a simulation is close to meeting a researchers needs, they can easily modify a component and reinsert it into the simulation without having to worry about breaking dependencies. Likewise, if a previously written simulation contains a component that could be inserted into one's current work, the component can easily be slipped into the new simulation, provided the interfaces meet simulation requirements.

Overall, these steps taken by the developers created a much more user-friendly environment than is currently provided by other simulators. Users of all levels are treated to tutorials catering to their specific needs, lowering the steepness of the learning curve found in many other simulation programs. Development time is drastically reduced due to the reusability and extensibility inherent to the component-oriented model leading to a much better overall experience.

5 Simulations and Results

All simulations were conducted using a Dell Latitude D600 with an Intel 1.6 Ghz Pentium-M processor and 512MB 266MHz DDR SDRAM. All simulations were optimized flooding simulations containing various numbers of immobile nodes. The size of the topologies as well as the amount and paths of traffic also varied from simulation to simulation. TCL and C++ scripts were written to randomly generate traffic (Appendix D) and topology (Appendix

E) files, and simulators were modified to read from the same input files. All nodes are running the 802.11 IEEE MAC layer protocol [11] in ad hoc mode. Simulations were conducted to compare the two simulators execution times and memory usage under various conditions.

5.1 NS-2 Simulations

All NS-2 simulations were conducted using NS-2 version 2.26, the current version at the time of writing. The NS-2 simulation we used for comparison was originally coded by Chalermek Intanagonwiwat of USC in May 1999 and was later edited for use in [4]. The TCL script that drives the simulation is a variation of tcl/ex/floodingrun.tcl with some very minor alterations (Appendix A). The main changes made were to disable the simulator from producing the trace file. This insured that we were timing the simulations not the output of the trace file. Additionally, we changed the calender scheduler to a heap scheduler because it is less sensitive to different time increment distributions. In order to time the NS-2 simulations, code was added to the run() function in common/scheduler.cc computing the start and finish time of each simulation, then making the subtraction.

5.2 SENSE Simulations

SENSE simulations were conducted using the latest beta version of SENSE (as of March 2004) compiled with GCC version 3.2.2. For comparison with the NS-2 flooding simulation, we modified the flooding component available with SENSE written by Gang Chen found at /test/sim_flooding.cpp was used. The modifications took place in both sim_flooding.cpp (Appendix B) and mobile/immobile.h (Appendix C) and were made to faciliate comparisons between the simulators by allowing SENSE the ability to read NS-2 traffic

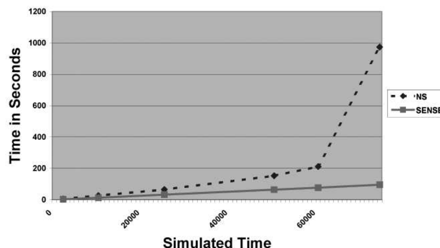

Figure 5: Execution time as simulated time increases

and topology files. This insures the simulators are working with the same data and simulating the same set of occurrences.

5.3 Results

We begin by looking at the way the two simulators handle simulations of an exceptional length. For comparison we created simulations containing thirty nodes, with the same random placement, in both NS-2 and SENSE. These simulations took place on an 800m by 800m terrain with sensors zero through eight each sending requests with a ten unit interval.

Perhaps the most important element of a simulator, other then perhaps ease of use, is the speed at which simulations are executed. There are two ways to look at the speed of a simulation, temporally and by the event processing rate. Figure 5 shows the drastic difference between how well NS-2 and SENSE handle lengthy simulations. We notice that both SENSE and NS-2 both have linear time increase with NS-2's running time roughly double that of SENSE, until the 75,000 unit simulation at which point the NS-2 simulation is exponentially higher. Another measure of the speed of a sim-

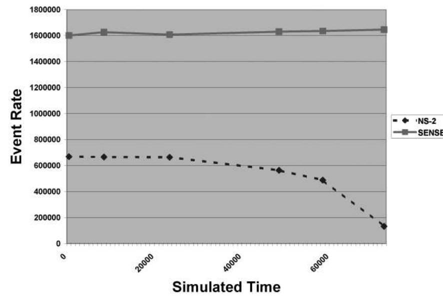

Figure 6: Event processing rate as simulated time increases

ulation is the event processing rate. This metric is the ratio of events per second during the simulation run. In Figure 6 we see that SENSE possesses a huge advantage over NS-2 in the simulation described above. The event processing rate of SENSE is not only much higher then that of NS-2 but more importantly it remains linear regardless of the length of the simulation. In contrast, the event processing rate of NS-2 is generally decreasing, particularly in the longer simulations. In order to explain the drastically different behavior of NS-2 during the longer simulations, we turn to the issue of memory consumption of the simulators. From Figure 7 the issue with the last data point of NS-2 is immediately evident. It is at this point that NS-2 fills up main memory and spills over into virtual memory. The time for swapping to disk is what accounts for the tremendous slowdown. The 60,000 unit data point for NS-2 just fits under the systems 512 MB capacity, however with system processes running in the background, there is some spillover into virtual memory there too. As a second scenario, we consider the simulators performance as we increase the number of nodes in a simulation. In this test we simulate various numbers of nodes again on an 800m

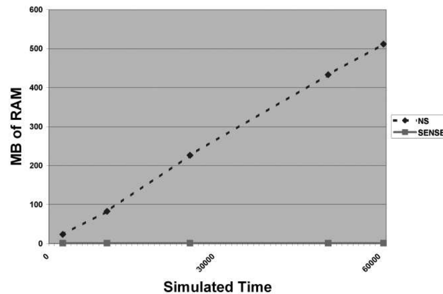

Figure 7: Memory usage as simulated time increases

by 800m terrain. Each node now is sending requests with an interval of 10 units. As more nodes get packed into the same amount of space, more traffic is generated throughout the network, plus each packet is received by more nodes creating more events. The number of events increases exponentially with the number of nodes placed in the simulation as shown in Figure 9. The increase in events intuitively causes the execution time of the simulation to increase exponentially as well (Figure 8). Looking at the event processing rate in Figure 10 we see the expected result, a slightly downward plot due to the extra computations the simulator needs to perform as the number of nodes crammed together in such a small space increases. Clearly the NS-2 simulations are lagging behind the SENSE simulations in both memory useage and execution speed. In an effort to maximize the performance of NS-2, modifications were made to the simulator. First we modified the NS-2 flooding simulation to remove all unnecessary headers from NS-2 packets. This cut down NS-2 memory useage by approximately thirty percent, but still left it lagging far behind SENSE.

The dramatic performance difference between NS-2 and SENSE can be

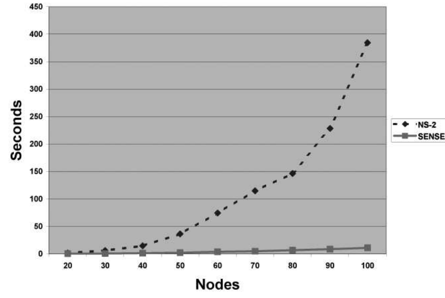

Figure 8: Execution time as the number of nodes increases.

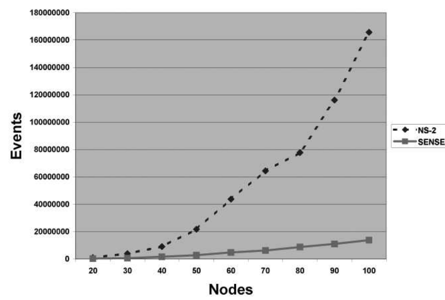

Figure 9: Number of events as the number of nodes increases.

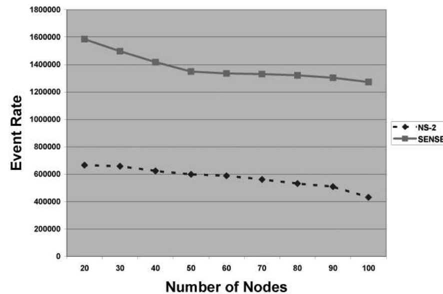

Figure 10: Event processing rate as the number of nodes increases

largely attributed to the ways they allocate and release packets. In NS-2, when a packet is about to be broadcast every neighboring node recieves a copy of said packet. This makes the number of packet allocations equal to the number of recieved packets. In SENSE on the other hand, a packet is shared by all receivers because only pointers are passed. This better packet management makes the number of packet allocations equal to the number of sent packets. The more dense a network becomes the larger difference between the number of packets sent and recieved (Figure 11). With this difference corrected we see the overall performance of NS-2 has increased substantially. First, running a sixty node simulation for various lengths of time with an interval of one packet per ten seconds with twenty percent of the nodes sending in a 1000m x 1000m area, we see that the memory consumption of NS-2 has become much more manageable (Figure 12). Note that NS-2 still uses approximately ten times the memory occupied by SENSE due to the fact that copies of each packet are passed to each node rather than pointers to a shared packet. This change is due mainly to a modification of the packet management scheme. Now, like SENSE, in NS-2 each node is not keeping

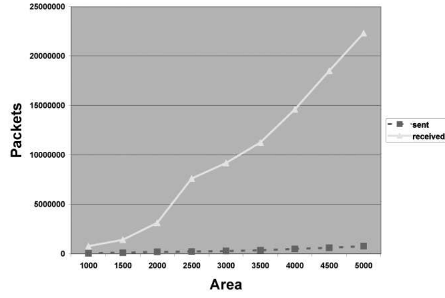

Figure 11: Number of packets sent and recieved as size of the simulation increases.

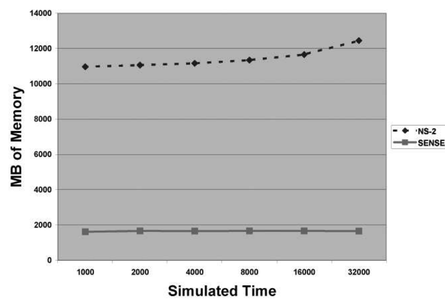

Figure 12: Memory usage with revised NS-2

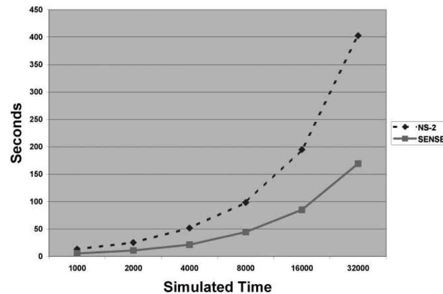

Figure 13: Execution time with revised NS-2.

track of every packet recieved in a hash table but rather is only storing the packet with the latest sequence number. We also see a drastic decrease in the execution time of NS-2 in the same simulations (Figure 13). This follows directly from the changes made to improve memory consumption.

6 Discussion and Conclusions

In this paper we have shown SENSE to be an effective and extremely efficient alternative to NS-2 for sensor network simulations. The component-oriented design model makes SENSE easier for new users to learn and gives experienced users the control and flexibility they require in a simulator. Additionally, the memory management system of SENSE makes for shorter execution times, dominating NS-2 in lengthy simulations.

Concerns with SENSE do not lay in the present but rather in the future. Will the development team become more relaxed in their documentation standards as the component repository grows larger? Will third party contributed components be tested for the same high quality as those created

internally before they are incorporated into the repository? As development on SENSE continues, it is crucial to make sure SENSE maintains it's current benefits over other simulators. Somehow, the developers must make sure that SENSE does not follow NS-2, which sacrificed ease of use for increased functionality.

NS-2, while a fine multi-purpose simulator, was not designed as a sensor network simulator and will therefore not be able to perform with the same efficiency as a specific simulator like SENSE. For many, NS-2 may still be the simulator of choice while SENSE builds its repository and becomes a stable product, however given the currently wide gap between the two simulators based on current functionality, SENSE will be the sensor network simulator of choice in the future.

7 Future Work

Now that SENSE has been released to the public, the main goal is to grow the component repository through internal and external development. Specifically, current work is being done to implement DSR and directed diffusion. As new components are created for the SENSE repository, developers must conduct thorough testing to make sure they uphold the orignal high standards of the simulator. Memory management specifically must be done correctly and kept as low as possible as errors in this area will only lead to larger problems in the future.

A visualization tool, similar to NAM in NS-2, should also be a priority. Being able to visualize their simulation gives beginning users a better grasp of what is happening internally and allows users of all levels to more easily find problems within their simulations. Determining the same information from

text base trace files is much more difficult. The inclusion of a visualization tool could also lead to a graphical interface in which a network could be constructed with minimal coding. Graphical network creation would make the simulator more appealing to low-end users.

Finally, parallelization is an issue becoming more important to today's researchers. The obvious benefits of parallelization can be applied to network simulators, the question is a matter of how. Many simulators, including NS-2 [18], have designed automated parallelization into NS-2 simulations. The developers of SENSE believe that this approach is doomed to be inefficient. Parallelization will provided as an option to users through a parallel simulation engine that will execute a set of compatible components [9].

References

[1] I. F. Akyildiz, W. Su, Y. sankarasubramaniam, and E. Cayirci, *A survey on sensor networks*, IEEE Communications Magazine, 40(8):102-114, August 2002.

[2] bajaj et al., *Improving Simulation for Network Research*, USC Computer Science Department Technical Report 99-702b, September 1999

[3] Edoardo Biagioni, Kent Bridges, *The application of remote sensor technology to assist the recoery or rare and endangered species*, Special issue on Distributed Sensor Networks for the International Journal of High Performance Computing Applications, Vol. 16 N. 3, August 2002..

[4] David Cavin, Yoav Sasson, and Andre Schiper, *On the Accuracy of MANET Simulators*, Proceedings of the Workshop on Principles of Mobile Computing (POMC '02), pages 38-43, ACM, October 2002.

[5] Gilbert Chen and Boleslaw K. Szymanski, *Lecture Notes in Computer Science, Parallel Processing, and Applied Mathematics: 4th international Conference*, A Component Model for Discrete Event Simulation, pages 580-594. Springer-Verlag, 2002.

[6] Gilbert Chen and Boleslaw K. Szymanski, *Component-Based Simulation*, 2001 European Simulation Multi-Conference, pages 68-75.

[7] Gilbert Chen and Boleslaw K. Szymanski, *Component-Based Simulation Architecture: Towards Interoperability and interchangeability*, 2001 Winter Simulation Conference, B.A Peters, J.S. Smith, D.J Medeiros, and M.W. Rohrer, eds., SCS Press, 2001, pp. 495-501.

[8] Gilbert Chen and Boleslaw K. Szymanski, *COST: A Component-Oriented Discrete Event Simulator*, Winter Simulation Conference WSC02, December 2002, vol. I, pp. 776-782.

[9] Gilbert Chen, Joel Branch, Eugene Brevdo, Lijuan Zhu, and Boleslaw K. Szymanski, *SENSE: A Sensor Network Simulator*, unpublished, www.cs.rpi.edu/tr/04-03.pdf.

[10] Component Oriented Simulation Toolkit, *http://www.cs.rpi.edu/ cheng3/cost/*

[11] IEEE 802.11, 1999 ed., *http://standards.ieee.org/getieee802/802.11.html*

[12] Chalermek Intanagonwiwat, Ramesh Govindan, Deorah Estrin, John Heidemann, Fabio Silva, *Directed Diffusion for Wireless Sensor Networking*, IEEE/ACM Transactions on Networking, Vol. 11, No. 1, Febuary 2003

[13] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson, *Wireless sensor networks for habitat monitoring*, ACM INternational Workshop on Wireless Sensor Networks and Applications, Atlanta, GA, September 2002.

[14] Rajeswari Malladi and Dharma P. Agrawal, *Current and Future Applications of Mobile and Wireless Networks*, Communications of the ACM, October 2002, Vol. 45, No. 10.

[15] Monarch Project, *http://www.monarch.cs.rice.edu*

[16] David M. Nicol, *Scalability of Network Simulators Revisited*, In Procedings of Communications Networks and Distributed Systems Modeling

and Simulation Conferences (CNDS) part of Western Multi-Confernce (WMC), 2003.

[17] THE NETWORK SIMULATOR - NS2, *http://www.isi.edu/nsnam/ns*

[18] PDNS - PARALLEL/DISTRIBUTED NS, *http://www.cc.gatech.edu/computing/compass/pdns/index.html*

[19] SENSE: SENSOR NETWORK SIMULATOR AND EMULATOR, *http://www.cs.rpi.edu/ cheng3/sense/*

[20] MANI B. SRIVASTAVA, RICHARD R. MUNTZ, AND MIODRAG POTKONJAK, *Smart kindergarten: sensor-based wireless networks for smart developmental problem solving environments*, Mobile Computing and Networking, pages 151-165, 2001.

[21] YU-CHEE TSENG AND SZE-YAO NI AND YUH-SHYAN CHEN AND JANG-PING-SHEU, *The broadcast storm problem in a mobile ad hoc network*, Wireless Networks, vol. 8, no. 2/3, 2002. Issn. 1022-0038, pp. 153-167.

[22] BRAD WILLIAMS AND TRACY CAMP, *Comparison of broadcasting rechniques for mobile ad hoc networks* Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking and computing, 2002, 194-205.

[23] NING XU, *A Survey of Sensor Network Applications*, www.enl.usc.edu/ ningxu/papers/survey.pdf

[24] YONGGUANG ZHANG, WEI LI, *An Integrated Environment for Testing Mobile Ad-Hoc Networks*, MOBIHOC '02, ACM, June 9-11 2002.

8 Appendix A: floodingrun.tcl

```
#
#Note that all trace file output has been commented out to minimize simulation time
#
# ======================================================================
# Define options
# ======================================================================
set opt(chan) Channel/WirelessChannel
set opt(prop) Propagation/TwoRayGround
set opt(netif) Phy/WirelessPhy
set opt(mac) Mac/802_11
set opt(ifq) Queue/DropTail/PriQueue
set opt(ll) LL
set opt(ant)            Antenna/OmniAntenna
set opt(god)            off
set opt(x) 800 ;# X dimension of the topography
set opt(y) 800     ;# Y dimension of the topography
#set opt(traf) "../test/sk-30-3-3-1-1-6-64.tcl"      ;# traffic file
set opt(traf) "./traffic.dat"      ;# traffic file
#set opt(topo) "../test/myfile"      ;# topology file
set opt(topo) "./topography.dat"      ;# topology file
set opt(onoff)          ""      ;#node on-off
set opt(ifqlen) 50 ;# max packet in ifq
set opt(nn) 30 ;# number of nodes
set opt(seed) 0.0
set opt(stop) 100000 ;# simulation time
set opt(prestop)        99999       ;# time to prepare to stop
```

```
set opt(tr) "wireless.tr" ;# trace file
set opt(nam)            "wireless.nam"  ;# nam file
set opt(engmodel)       EnergyModel     ;
set opt(txPower)        0.660;
set opt(rxPower)        0.395;
set opt(idlePower)      0.035;
set opt(initeng)        10000.0         ;# Initial energy in Joules
set opt(logeng)         "off"           ;# log energy every 1 seconds
set opt(lm)             "off"           ;# log movement
set opt(adhocRouting)   FLOODING
set opt(duplicate)      "enable-duplicate"

# ========================================================================

LL set mindelay_ 50us
LL set delay_ 25us
LL set bandwidth_ 0 ;# not used

Queue/DropTail/PriQueue set Prefer_Routing_Protocols    1

# unity gain, omni-directional antennas
# set up the antennas to be centered in the node and 1.5 meters above it
Antenna/OmniAntenna set X_ 0
Antenna/OmniAntenna set Y_ 0
Antenna/OmniAntenna set Z_ 1.5
Antenna/OmniAntenna set Gt_ 1.0
Antenna/OmniAntenna set Gr_ 1.0
```

```
# Initialize the SharedMedia interface with parameters to make
# it work like the 914MHz Lucent WaveLAN DSSS radio interface
Phy/WirelessPhy set CPThresh_ 10.0
Phy/WirelessPhy set CSThresh_ 1.559e-11
Phy/WirelessPhy set RXThresh_ 3.652e-10
Phy/WirelessPhy set Rb_ 2*1e6
Phy/WirelessPhy set Pt_ 0.2818
Phy/WirelessPhy set freq_ 914e+6
Phy/WirelessPhy set L_ 1.0

# ========================================================================

proc usage { argv0 }  {
    puts "Usage: $argv0"
    puts "\t\t\[-topo topology file\] \[-traf traffic file\]"
    puts "\t\t\[-x max x\] \[-y max y\] \[-seed seed\]"
    puts "\t\t\[-nam nam file\] \[-tr trace file\] \[-logeng on or off\]"
    puts "\t\t\[-stop time to stop\] \[-prestop time to prepare to stop\]"
    puts "\t\t\[-initeng initial energy\] \[-engmodel energy model\]"
    puts "\t\t\[-chan channel model\] \[-prop propagation model\]"
    puts "\t\t\[-netif network interface\] \[-mac mac layer\]"
    puts "\t\t\[-ifq interface queue\] \[-ll link layer\] \[-ant antena\]"
    puts "\t\t\[-ifqlen interface queue length\] \[-nn number of nodes\]"
}

proc getopt {argc argv} {
```

```
global opt
lappend optlist cp nn seed sc stop tr x y

for {set i 0} {$i < $argc} {incr i} {
                set arg [lindex $argv $i]
                if {[string range $arg 0 0] != "-"} continue

                set name [string range $arg 1 end]
                set opt($name) [lindex $argv [expr $i+1]]
}
}

proc finish {} {
    global ns_ nf opt god_ node_
    $ns_ terminate-all-agents
    $god_ dump_num_send
    $ns_ flush-trace
    if [info exists tracefd] {
        close $tracefd
#       exec rm -f $opt(tr)
    }
    if [info exists nf] {
        close $nf
#       exec rm -f $opt(nam)
#       exec nam $opt(nam) &
#       exec gzip $opt(nam)
    }
```

```
    exit 0
}

#comment out trace file to reduce simulation time
#proc cmu-trace { ttype atype node } {
# global ns_ tracefd
#
# if { $tracefd == "" } {
# return ""
# }
# set T [new CMUTrace/$ttype $atype]
# $T target [$ns_ set nullAgent_]
# $T attach $tracefd
#        $T set src_ [$node id]
#
#        $T node $node
#
# return $T
#}

# ======================================================================
# Main Program
# ======================================================================
getopt $argc $argv

# do the get opt again incase the routing protocol file added some more
```

```
# options to look for
getopt $argc $argv
if {$opt(seed) > 0} {
        puts "Seeding Random number generator with $opt(seed)\n"
        ns-random $opt(seed)
}

# Initialize Global Variables
set ns_ [new Simulator]
$ns_ use-scheduler Heap

#$ns_ set-scheduler Heap  wont run with this in it??? why?
# define color index
$ns_ color 0 red
$ns_ color 1 blue
$ns_ color 2 chocolate
$ns_ color 3 yellow
$ns_ color 4 green
$ns_ color 5 tan
$ns_ color 6 gold
$ns_ color 7 black
#set chan [new $opt(chan)]
#set prop [new $opt(prop)]
set topo [new Topography]

if { $opt(nam) != "" } {
    set nf [open $opt(nam) w]
```

```
    $ns_ namtrace-all-wireless $nf $opt(x) $opt(y)
}
if { $opt(tr) != ""} {
    set tracefd [open $opt(tr) w]
    $ns_ use-newtrace
    $ns_ trace-all $tracefd
}

$topo load_flatgrid $opt(x) $opt(y)
#$prop topography $topo

# Create God
set god_ [create-god $opt(nn)]
$god_ $opt(god)
$god_ allow_to_stop
$god_ num_data_types 1

# log the mobile nodes movements if desired
if { $opt(lm) == "on" } {
    log-movement
}

#global node setting

$ns_ node-config -adhocRouting $opt(adhocRouting) \
 -llType $opt(ll) \
 -macType $opt(mac) \
```

```
 -ifqType $opt(ifq) \
 -ifqLen $opt(ifqlen) \
 -antType $opt(ant) \
 -propType $opt(prop) \
 -phyType $opt(netif) \
 -channelType $opt(chan) \
 -topoInstance $topo \
 -agentTrace OFF \
                -routerTrace OFF \
                -macTrace OFF \
 -energyModel $opt(engmodel) \
 -initialEnergy $opt(initeng) \
 -txPower  $opt(txPower) \
 -rxPower  $opt(rxPower) \
 -idlePower  $opt(idlePower)

#comment out trace file to speed up simulation
#$ns_ node-config -adhocRouting $opt(adhocRouting) \
#  -llType $opt(ll) \
#  -macType $opt(mac) \
#  -ifqType $opt(ifq) \
#  -ifqLen $opt(ifqlen) \
#  -antType $opt(ant) \
#  -propType $opt(prop) \
#  -phyType $opt(netif) \
#  -channelType $opt(chan) \
#  -topoInstance $topo \
```

```
#  -agentTrace ON \
#                 -routerTrace ON \
#                 -macTrace ON \
#  -energyModel $opt(engmodel) \
#  -initialEnergy $opt(initeng) \
#  -txPower  $opt(txPower) \
#  -rxPower  $opt(rxPower) \
#  -idlePower  $opt(idlePower)

#  Create the specified number of nodes [$opt(nn)] and "attach" them
#  to the channel.
for {set i 0} {$i < $opt(nn) } {incr i} {
        set node_($i) [$ns_ node $i]
        $node_($i) color black
        $node_($i) random-motion 0 ;# disable random motion
        $god_ new_node $node_($i)
}
if { $opt(topo) == "" } {
        puts "*** NOTE: no topology file specified."
        set opt(topo) "none"
} else {
        puts "Loading topology file..."
        source $opt(topo)
        puts "Load complete..."
}
for {set i 0} {$i < $opt(nn)} {incr i} {
    $node_($i) namattach $nf
```

```
   $ns_ initial_node_pos $node_($i) 20
}
if { $opt(onoff) == "" } {
        puts "*** NOTE: no node-on-off file specified."
        set opt(onoff) "none"
} else {
        puts "Loading node on-off file..."
        source $opt(onoff)
        puts "Load complete..."
}

# log energy if desired
if { $opt(logeng) == "on" } {
    log-energy
}

# Source the traffic scripts
if { $opt(traf) == "" } {
        puts "*** NOTE: no traffic file specified."
        set opt(traf) "none"
} else {
        puts "Loading traffic file..."
        source $opt(traf)
}

# Tell all the nodes when the simulation ends
$ns_ at $opt(prestop) "$ns_ prepare-to-stop"
```

```
#$ns_ at $opt(stop).00000001    "finish"
#$ns_ at $opt(stop).00001  terminate-all-agents
for {set i 0} {$i < $opt(nn) } {incr i} {
    $ns_ at $opt(stop).00000002 "$node_($i) reset";
}

# tell nam the simulation stop time
$ns_ at  $opt(stop).00000003 "$ns_ nam-end-wireless $opt(stop)"
$ns_ at $opt(stop).00000004 "puts \"NS EXITING...\" ; $ns_ halt"

puts $tracefd "M 0.0 nn $opt(nn) x $opt(x) y $opt(y)"
puts $tracefd "M 0.0 topo $opt(topo) traf $opt(traf) seed $opt(seed)"
puts $tracefd "M 0.0 prop $opt(prop) ant $opt(ant)"

puts "Starting Simulation..."
$ns_ run
```

9 Appendix B: sim_flooding.cpp

```
#define PRIORITY_QUEUE HeapQueue

/*******************************************************************
 * This header file is required
 *******************************************************************/
#include "../common/sense.h"
#include <fstream>
#include <iostream>
#include <string>

/*******************************************************************
 * The following head files are necessary only if the corresponding
 * components are needed by the sensor node component
 *******************************************************************/
#include "../app/cbr.h"
#include "../mobile/immobile.h"
#include "../network/flooding_routing.h"
#include "../queue/fifo_ack.h"
#include "../mac/mac802_11.h"
#include "../mac/simple_mac.h"
#include "../phy/transceiver.h"
#include "../phy/wireless_channel.h"
#include "../energy/battery.h"
using namespace std;

/*******************************************************************
```

```
 * The default mac component is @MAC802_11@.  Another choice is
 * @Simple_MAC@.
 ******************************************************************/
#ifndef mac_component_t
#define mac_component_t MAC802_11
#endif

/******************************************************************
 * Now we can begin to define the sensor node component. First we
 * instantiate every subcomponent. The most complicate part is to
 * determine the template parameter type for each subcomponent.  We
 * usually work in a top-down manner, starting from the application
 * layer. Normally the application layer component does not have any
 * template parameter, so the template parameter type of the network
 * layer component depends on the packet type of the application
 * layer, and the mac layer depends on the network layer.  For
 * convenience, we usually use @typedef@ to avoid writing very long
 * type names.
 *
 * @phy_inf_t@ is the standard physical layer interface that is used
 * to transmit packets between the mac and physical layers. It includes
 * the packet, the error flag, and the transmission/receive power.
 * The data inport and outport of the sensor node component are of this
 * interface type.
 *
 * This picture shows the internal structure of a sensor node.
 *
```

```
 * @<center><img src=sim_flooding_node.gif></center>@
 *******************************************************************/

class SensorNode : public TypeII
{
public:

    CBR app;
    Immobile mob;
    FloodingRouting <CBR::packet_t> net;
    typedef FloodingRouting <CBR::packet_t>::packet_t net_packet_t;
    mac_component_t <net_packet_t*> mac;
    typedef mac_component_t<net_packet_t*>::packet_t mac_packet_t;
    typedef phy_inf_t<mac_packet_t*> mac_txinfo_t;
    DuplexTransceiver < mac_packet_t > phy;
    SimpleBattery battery;

/*******************************************************************
 * We may put a queue between the network and mac layers, to prevent
 * packets from being dropped
 *******************************************************************/

#ifdef MAC_QUEUE
    FIFO_ACK< net_inf_t<net_packet_t*> > queue;
#endif

/*******************************************************************
```

```
 * Components parameters are defined here.
 *****************************************************************/
    double MaxX, MaxY;        // the size of the geographical area
    ether_addr_t MyEtherAddr; // the ethernet address of this node

/*****************************************************************
 * These three functions must be present for all SENSE components.
 * @Start()@ will be called when the simulation is started, @Stop()@
 * when the simulation is stopped, @Setup()@ when the simulation is
 * being set up.
 *****************************************************************/
    void Start();
    void Stop();
    void Setup(TypeII*,const char*);

/*****************************************************************
 * Inports and Outports are declared here.
 *****************************************************************/
    Outport <void, const mac_txinfo_t > to_channel_packet;
    Inport <void, const mac_txinfo_t > from_channel;
    Outport <void, const coordinate_t > to_channel_pos;
};

void SensorNode::Start()
{
}
```

```
void SensorNode::Stop()
{
}

void SensorNode::Setup(TypeII*c, const char* name)
{
/*******************************************************************
 * The first thing to do in the @Setup@ function is to assign values
 * the every parameter of every subcomponent.
 *******************************************************************/
    battery.InitialEnergy=1e6;
    app.MyEtherAddr=MyEtherAddr;
    app.DumpPackets=true;
    mob.MaxX=MaxX;
    mob.MaxY=MaxY;
    net.MyEtherAddr=MyEtherAddr;
    net.ForwardDelay=.1;
    net.DumpPackets=true;
    mac.MyEtherAddr=MyEtherAddr;
    mac.DumpPackets=true;
    mac.Promiscuity=false;

    phy.TXConsumed=1.6;         // these three parameters determine
    phy.RXConsumed=1.2;         // the power consumption in transmit,
    phy.IdleConsumed=1.15;      // receive, and idle modes.
    phy.TXPower=0.280;          // transmitted signal strength
    phy.TXGain=1.0;             // transmission antenna gain
```

```
    phy.RXGain=1.0;            // receive antenna gain
    phy.Frequency=9.14e8;      // frequency of the signal
    phy.RXThresh=3.652e-10;    // weakest signal received
    phy.CSThresh=1.559e-11;    // weakest signal detected

/********************************************************************
 * @Setup()@ functions of subcomponent are called here.
 ********************************************************************/

    app.Setup(this,"app");
    mob.Setup(this,"mob");
    net.Setup(this,"net");
    mac.Setup(this,"mac");
    phy.Setup(this,"phy");
    battery.Setup(this,"battery");
#ifdef MAC_QUEUE
    queue.Setup(this,"queue");
#endif
    to_channel_packet.Setup(this,"to_channel_packet");
    to_channel_pos.Setup(this,"to_channel_pos");
    from_channel.Setup(this,"from_channel");


/********************************************************************
 * Now we can connect pairs of outports and inports.
 ********************************************************************/
```

```
    Connect(app.to_transport, net.from_transport);
    Connect(net.to_transport, app.from_transport);

#ifdef MAC_QUEUE
    Connect(net.to_mac, queue.in);
    Connect(queue.ack,net.from_mac_ack);
    Connect(mac.to_network_ack,queue.next);
    Connect(queue.out,mac.from_network);
#else
    Connect(net.to_mac, mac.from_network);
    Connect(mac.to_network_ack, net.from_mac_ack);
#endif

    Connect(mac.to_network_data, net.from_mac_data);
    Connect(mac.to_phy, phy.from_mac);
    Connect(phy.to_mac, mac.from_phy);
    Connect(phy.to_channel, to_channel_packet);
    Connect(from_channel, phy.from_channel);
    Connect(phy.to_battery, battery.in);
    Connect(mob.pos_out, to_channel_pos);

/********************************************************************
 * The transciever has an inport which can turn it off.  In this
 * example this inport is not connected to any other outport. Without
 * this function call a warning would be prompted for each node
 * component.
 ********************************************************************/
```

```
    phy.power_switch.SetConnected();

/******************************************************************
 * This function must be called last.
 ******************************************************************/

    TypeII::Setup(c,name);

}

/******************************************************************
 * Now we are ready to model the sensor network using the sensor node
 * component just created. Here is the structure of the network.
 *
 * @<center><img src=sim_flooding_net.gif></center>@
 ******************************************************************/

class SimFlooding : public CostSimEng
{
public:

/******************************************************************
 * Simulation parameters here.
 ******************************************************************/

    double MaxX, MaxY;           // the size of the geographical area
    int NumNodes;                // number of sensor nodes
```

```
    int NumSourceNodes;          // number of sensor nodes acting as sources
    int NumConnections;          // number of connections per source
    int PacketSize;              // size of packets generated in the application la
    int Interval;                // interval between consecutive packet creation

/******************************************************************
 * Sensor nodes are declared as a component array. Another component
 * is the wireless channel component.
 ******************************************************************/

    TypeIIArray < SensorNode > nodes;
    WirelessChannel < SensorNode::mac_packet_t> channel;

/******************************************************************
 * These three functions as required.
 ******************************************************************/
    void Start();
    void Stop();
    void Setup(const char*);
};

void SimFlooding :: Start()
{
}

/******************************************************************
 * This function collects some statistics.
```

```
 *****************************************************************/

void SimFlooding :: Stop()
{
    int i,sent=0,recv=0;
    for(i=0;i<NumNodes;i++)
    {
        sent+=nodes[i].app.SentPackets;
        recv+=nodes[i].app.RecvPackets;
    }
    printf("total packets sent: %d, received: %d\n",sent,recv);
}
void SimFlooding :: Setup(const char*name)
{
    int i,j;

/*****************************************************************
 * A component array must be instantiated by @SetSize()@ before each
 * individual component can be referenced.
 *****************************************************************/
    nodes.SetSize(NumNodes);

/*****************************************************************
 * Initialize componenter parameters.
 *****************************************************************/

    for(i=0;i<NumNodes;i++)
```

```
    {
nodes[i].MaxX=MaxX;
nodes[i].MaxY=MaxY;
nodes[i].MyEtherAddr=i;
    }
    nodes.Setup(this,"station");

    channel.NumNodes=NumNodes;
    channel.DumpPackets=true;
    channel.CSThresh=nodes[0].phy.CSThresh;
    channel.RXThresh=nodes[0].phy.RXThresh;
    channel.PropagationModel=channel.FreeSpace;

    channel.Setup(this,"channel");

/*******************************************************************
 * Make the connections.  Now read in from NS-2 file (mpflug)
 *******************************************************************/
    for(i=0;i<NumNodes;i++)
    {
Connect(nodes[i].to_channel_packet,channel.from_phy[i]);
Connect(nodes[i].to_channel_pos,channel.from_sensor[i]);
Connect(channel.to_phy[i],nodes[i].from_channel);
    }

    int src, dst, size;
    double iv;
```

```
    ifstream inFile;
    //open the traffic file for reading
    inFile.open("traffic.dat");
    string word;
    inFile >> word;
    while(word != "num_connect:")
inFile >> word;
    inFile >> NumSourceNodes >> word >> Interval;
    cout<<"Number of sorce nodes is "<<NumSourceNodes<<endl;
    NumConnections = 1;
    for(i = 0; i < NumSourceNodes; i++)
    {
            //get source
            inFile >> word;
            while(word != "node")
            inFile >> word;
            inFile >> src;
for(j=0;j<NumConnections;j++)
{
        //get packet size
            inFile >> word;
            while(word != "packetSize_")
            inFile >> word;
            inFile >> PacketSize;
    //get interval
            inFile >> word;
            while(word != "interval_")
```

```
            inFile >> word;
            inFile >> Interval;
            //get destination
            inFile >> word;
            while(word != "node")
            inFile >> word;
            inFile >> dst;
    //cout<<"source is "<<src<<" pack size is "<<PacketSize<<" interval is "<<Inter
    size=Random(PacketSize)+PacketSize/2;
    iv=Random(Interval)+Interval/2;

    nodes[src].app.Connections.push_back(
make_triple(ether_addr_t(dst),size,iv));
}
    }
    inFile.close();
/*******************************************************************
 * This function must be called last.
 *******************************************************************/
    CostSimEng::Setup(name);
}

/*******************************************************************
 * Here is how to run the simulation.
 *******************************************************************/

int main(int argc, char* argv[])
```

```
{
    SimFlooding sim;

/*******************************************************************
 * These three system parameters are predefined. @StopTime@ determines
 * the total simulation time.  @Seed@ is the initial seed of the
 * random number generator. @InfoLevel@ determines the detail level of
 * output information.
 *******************************************************************/
    sim.StopTime=75000;
    sim.Seed=1234;
    sim.InfoLevel=2;

/*******************************************************************
 * Application specific system parameters.
 ********************************************************************/

    sim.MaxX=800;
    sim.MaxY=800;
    sim.NumNodes=30;
    /*sim.MaxX=5000;
    sim.MaxY=5000;
    sim.NumNodes=100;*/
    sim.NumConnections=2;
    sim.PacketSize=64;
    sim.Interval=10;
```

```
    if(argc>=2)sim.StopTime=atof(argv[1]);
    if(argc>=3)sim.NumNodes=atoi(argv[2]);
    if(argc>=4)sim.MaxX=sim.MaxY=atof(argv[3]);

    sim.NumSourceNodes=sim.NumNodes/10;

/*******************************************************************
 * @Setup()@ must be before @Run()@.
 *******************************************************************/

    sim.Setup("flooding sim");
    sim.Run();
    return 0;
}
```

10 Appendix C: immobile.h

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;
#ifndef immobile_h
#define immobile_h

class Immobile : public TypeII
{
public:

    Outport <void, const coordinate_t> pos_out;
    double MaxX, MaxY;
    void Start();
    void Stop();
    void Setup(TypeII*, const char*);

};

//global array of nodes
double coords[100];
int flag = 1;
int nodeNumber=1;
int arrayPos=0;
```

```
void Immobile::Start()
{
    double x = coords[arrayPos];
    arrayPos++;
    double y = coords[arrayPos];
    arrayPos++;
    //the code to assign positon
    coordinate_t pos=coordinate_t(x,y);
    pos_out.Write(pos,0.0);
    Printf((true,"moves to %f, %f\n",pos.x,pos.y));
}
void Immobile::Stop()
{
}
void Immobile::Setup(TypeII*c, const char* name)
{
   //read in the global array of coordinates from the file
   if(flag == 1){
      ifstream infile;
      infile.open("topography.dat");
      string word;
      int numNodes, place;
      place = 0;
      double myX, myY;
      infile >> word;
      while(word != "nodes:")
          infile >> word;
```

```
      infile >> numNodes;
       for(int i = 1; i <= numNodes; i++)
       {
            //get x coord
            infile >> word;
            while(word != "X_")
               infile >> word;
            infile >> myX;
            //get y coord
            infile >> word;
            while(word != "Y_")
               infile >> word;
            infile >> myY;
          coords[place] = myX; place++;
          coords[place] = myY; place++;
        }
       place = 0;
       infile.close();
       flag = 0;
    }
    pos_out.Setup(this,"pos_out");
    TypeII::Setup(c,name);
}


#endif /* immobile_h*/
```

11 Appendix D: Traffic Generation Script

```
#This code is a modified version of the traffic source generator developed by USC
# for use with CMU's mobile code.
#
# ======================================================================
# Default Script Options
# ======================================================================
set opt(nn) 0 ;# Number of Nodes
set opt(seed) 0.0
set opt(mc) 0
set opt(pktsize) 64
set opt(rate) 0
set opt(interval) 1 ;# inverse of rate
set opt(type)           ""

# ======================================================================

proc usage {} {
    global argv0
    puts "\nusage: $argv0 \[-type cbr|tcp\] \[-nn nodes\] \[-seed seed\] \[-mc conn
}

proc getopt {argc argv} {
global opt
lappend optlist nn seed mc rate type
for {set i 0} {$i < $argc} {incr i} {
set arg [lindex $argv $i]
```

```
if {[string range $arg 0 0] != "-"} continue
set name [string range $arg 1 end]
set opt($name) [lindex $argv [expr $i+1]]
        }
}

proc create-cbr-connection { src dst } {
global rng cbr_cnt opt
set stime [$rng uniform 0.0 180.0]
puts "#\n# $src connecting to $dst at time $stime\n#"
##puts "set cbr_($cbr_cnt) \[\$ns_ create-connection \
##CBR \$node_($src) CBR \$node_($dst) 0\]";
puts "set udp_($cbr_cnt) \[new Agent/UDP\]"
puts "\$ns_ attach-agent \$node_($src) \$udp_($cbr_cnt)"
puts "set null_($cbr_cnt) \[new Agent/Null\]"
puts "\$ns_ attach-agent \$node_($dst) \$null_($cbr_cnt)"
puts "set cbr_($cbr_cnt) \[new Application/Traffic/CBR\]"
puts "\$cbr_($cbr_cnt) set packetSize_ $opt(pktsize)"
puts "\$cbr_($cbr_cnt) set interval_ $opt(interval)"
puts "\$cbr_($cbr_cnt) set random_ 1"
puts "\$cbr_($cbr_cnt) set maxpkts_ 10000"
puts "\$cbr_($cbr_cnt) attach-agent \$udp_($cbr_cnt)"
puts "\$ns_ connect \$udp_($cbr_cnt) \$null_($cbr_cnt)"
puts "\$ns_ at $stime \"\$cbr_($cbr_cnt) start\""
incr cbr_cnt
}
```

```
proc create-src-connection { src snk } {
global rng cbr_cnt opt
set stime [$rng uniform 0.0 180.0]

##source
puts "#\n# node $src is ready to send data type 0 at time $stime\n#"
puts "set src_($cbr_cnt) \[new Agent/Diff_Sink\]"
puts "\$ns_ attach-agent \$node_($src) \$src_($cbr_cnt)"
puts "\$ns_ put-in-list \$src_($cbr_cnt)"
puts "\$src_($cbr_cnt) set packetSize_ $opt(pktsize)"
puts "\$src_($cbr_cnt) set interval_ $opt(interval)"
puts "\$src_($cbr_cnt) data-type 0"
puts "\$src_($cbr_cnt) set fid_ 0"
puts "\$src_($cbr_cnt) \$opt(duplicate)"
puts "\$node_($src) color red"
puts "\$ns_ at $stime \"\$src_($cbr_cnt) ready\""

##sink
puts "#\n# node $snk expresses interest type 0 at time $stime\n#"
puts "set sk_($cbr_cnt) \[new Agent/Diff_Sink\]"
puts "\$ns_ attach-agent \$node_($snk) \$sk_($cbr_cnt)"
puts "\$ns_ put-in-list \$sk_($cbr_cnt)"
puts "\$sk_($cbr_cnt) set packetSize_ $opt(pktsize)"
puts "\$sk_($cbr_cnt) set interval_ $opt(interval)"
puts "\$sk_($cbr_cnt) data-type 0"
puts "\$sk_($cbr_cnt) set fid_ 1"
puts "\$sk_($cbr_cnt) \$opt(duplicate)"
```

```
puts "\$node_($src) color blue"
puts "\$ns_ at $stime \"\$sk_($cbr_cnt) announce\""

incr cbr_cnt
}

proc create-tcp-connection { src dst } {
global rng cbr_cnt opt
set stime [$rng uniform 0.0 180.0]
puts "#\n# $src connecting to $dst at time $stime\n#"
puts "set tcp_($cbr_cnt) \[\$ns_ create-connection \
TCP \$node_($src) TCPSink \$node_($dst) 0\]";
puts "\$tcp_($cbr_cnt) set window_ 32"
puts "\$tcp_($cbr_cnt) set packetSize_ $opt(pktsize)"
puts "set ftp_($cbr_cnt) \[\$tcp_($cbr_cnt) attach-source FTP\]"
puts "\$ns_ at $stime \"\$ftp_($cbr_cnt) start\""
incr cbr_cnt
}

# =======================================================================

getopt $argc $argv

  if { $opt(type) == "" } {
    usage
    exit
  }
```

```
  elseif { $opt(type) == "cbr" } {
    if { $opt(nn) == 0 || $opt(seed) == 0.0 || $opt(mc) == 0 || $opt(rate) == 0 } {
        usage
        exit
    }
    set opt(interval) [expr 1 / $opt(rate)]
    if { $opt(interval) <= 0.0 } {
        puts "\ninvalid sending rate $opt(rate)\n"
        exit
    }
}

puts "#\n# nodes: $opt(nn) num_connect: $opt(mc) interval: $opt(interval) seed: $op
set rng [new RNG]
$rng seed $opt(seed)
set u [new RandomVariable/Uniform]
$u set min_ 0
$u set max_ 100
$u use-rng $rng
set cbr_cnt 0
set src_cnt 0

for {set i 0} {$i < $opt(nn) } {incr i} {
set x [$u value]
if {$x < 50} {continue;}
incr src_cnt
set dst [expr ($i+1) % [expr $opt(nn) + 1] ]
```

```
#if { $dst == 0 } {
   #set dst [expr $dst + 1]
#}
if { $opt(type) == "cbr" } {
   create-cbr-connection $i $dst
}
if { $opt(type) == "src" } {
   create-src-connection $i $dst
}
if { $opt(type) == "tcp" } {
   create-tcp-connection $i $dst
}
if { $cbr_cnt == $opt(mc) } {
   break
}

if {$x < 75} {continue;}
set dst [expr ($i+2) % [expr $opt(nn) + 1] ]
#if { $dst == 0 } {
   #set dst [expr $dst + 1]
#}

if { $opt(type) == "cbr" } {
       create-cbr-connection $i $dst
}
if { $opt(type) == "src" } {
   create-src-connection $i $dst
```

```
}
if { $opt(type) == "tcp" } {
   create-tcp-connection $i $dst
}

if { $cbr_cnt == $opt(mc) } {
   break
}
}

puts "#\n#Total sources/connections: $src_cnt/$cbr_cnt\n#"
```

12 Appendix E: Topology Generation Script

```
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <cstdlib>
using namespace std;
double random_range(int lowest_number, int highest_number);

int main(){
   ofstream outfile;
   int numNodes, i, j, area;
   double random, random2, random3, something, startTime;
   outfile.open("myfile");
   outfile<<setprecision(10);
   cout<<"Enter number of nodes:"<<endl;
   cin >> numNodes;
   cout<<"Enter area"<<endl;
   cin >> area;
   startTime = .126;
   outfile <<"#"<<endl;
   outfile <<"# "<<"nodes: "<<numNodes<<endl;
   outfile <<"#"<<endl;

   //Coordinates
   for(i = 0; i<numNodes; i++){
      random  = random_range(0, area);
```

```
      random2 = random_range(0, area);
      outfile <<"$node_("<<i<<") set X_ "<<random<<endl;
      outfile <<"$node_("<<i<<") set Y_ "<<random2<<endl;
      outfile <<"$node_("<<i<<") set Z_ "<<0.0<<endl;
   }

   //God specs
   for(i = 0; i<numNodes; i++){
      for(j = i+1; j<numNodes; j++){
        something = 4;
        outfile <<"$god_ set-dist "<<i<<" "<<j<<" "<<something<<endl;
      }
   }

   //Destination and Start times
   for(i = 0; i<numNodes; i++){
      random  = random_range(0,area);
      random2 = random_range(0,area);
      random3 = random_range(0,1);
      outfile<<"$ns_ at "<<startTime<<" \"$node_("<<i<<") setdest "<<random<<" "<<r
   }
   return 0;
}

//random number generation
double random_range(int lowest_number, int highest_number)
{
```

```
    if(lowest_number > highest_number){
        swap(lowest_number, highest_number);
    }
    int range = highest_number - lowest_number + 1;
    return lowest_number + double(range * double(rand()/(RAND_MAX + 1.0)));
}
```