Physical Implementation and Evaluation of Ad Hoc Network Routing Protocols using Unmodified Simulation Models

Amit Kumar Saha Khoa Anh To Santashil PalChaudhuri Shu Du David B. Johnson

Rice University
Departments of Computer Science and Electrical and Computer Engineering
6100 Main Street
Houston, TX 77005-1892 USA
{ amsaha, takhoa, santa, dushu, dbj } @cs.rice.edu

ABSTRACT

Simulation and physical implementation are both valuable tools in evaluating ad hoc network routing protocols, but neither alone is sufficient. In this paper, we present the design and implementation of a new system that allows existing simulation models of ad hoc network routing protocols to be used — without modification — *to create a physical implementation of the same protocol. We have evaluated the simplicity and portability of our approach across multiple protocols and multiple operating systems through example implementations in our architecture of the DSR and AODV routing protocols in FreeBSD and Linux using the existing, unmodified* ns-2 *simulation models. We also illustrate the ability of the resulting protocol implementations to handle real, demanding applications by presenting a demonstration of this DSR implementation transmitting real-time video over a multihop mobile ad hoc network including mobile robots being remotely operated based on the transmitted video stream.*

Categories and Subject Descriptors

C.5.0 [**Computer Systems Organization**]: Computer System Implementation—*Miscellaneous*

General Terms

Design

Keywords

Implementation, ad hoc networks, routing protocols, *ns-2*, DSR, AODV, FreeBSD, Linux

1. INTRODUCTION

Ad hoc networking is currently a very active area of research, yet evaluating the many proposed protocols for ad hoc networks remains difficult. In an ad hoc network, wireless nodes cooperate to form a network, forwarding packets for each other to allow nodes not within direct wireless transmission range of each other to communicate. The behavior of the system can be quite dynamic due to factors such as node movement and variations in radio propagation conditions, creating frequent changes in network topology, differing concentrations in traffic load on the network, and other challenges to the operation of the network protocols.

The most common method of evaluation, network simulation, has many advantages. For example, simulation allows repeatable experiments, for comparing one protocol or protocol version to another under identical workloads. Also, it is generally easier than full physical implementation, since it avoids the need for moving the nodes under test and can evaluate systems for which the necessary hardware is not available. However, simulation may fail to capture the precise behavior of the real system, as it is difficult to accurately model the complexities of real radio propagation, realistic node mobility, and application data traffic workload.

On the other hand, physical protocol implementation allows the real system itself to be measured and can help to validate simulations, but protocol evaluations using physical implementation are generally much more difficult than simulation evaluations. For example, physical implementation must deal with real packet formats and application programming interfaces, whereas such factors can be simplified and abstracted in simulation. In addition, evaluations using physical implementation are generally much more time- and equipment-intensive than simulations, due to the use of real hardware and real mobility and the exposure of the experiments (and the experimenters) to the real environment in which this mobility takes place.

Simulation and physical implementation are each valuable as techniques in evaluating ad hoc network protocols, and any complete evaluation should include both. However, this approach normally requires two separate implementations of the protocol, one for the simulation model and one for the physical system, resulting extra effort in coding, debugging, validation, and maintenance.

To address this conflict between simulation and physical implementation, in this paper, we present the design and implementation of a new system that allows *existing* simulation models of ad hoc network routing protocols to be used — *without modification* — to create a physical implementation of the same protocol. The protocol on each node runs entirely in a single user-level process, and our system uses standard interfaces to transmit and receive packets from the kernel, simplifying protocol debugging and making the

SIGCOMM ASIA WORKSHOP, April 12–14, 2005, Beijing, China.

system highly portable between different host operating systems. Although the system is based on unmodified *simulation* code, the resulting physical implementation is entirely *real*, not simulated, running on real hardware, with real mobility, real packets, and real wireless network interfaces. Our current system is based on the *ns-2* network protocol simulator, but the techniques used in our system should also be readily portable to other network simulation environments.

We have evaluated the simplicity and portability of our approach across multiple protocols and multiple operating systems through example implementations in our architecture of the Dynamic Source Routing protocol (DSR) [10] and the Ad hoc On-Demand Vector Routing protocol (AODV) [19] on FreeBSD and Linux using the existing, unmodified *ns-2* simulation models. The user-level code is identical between our implementations on FreeBSD and Linux, and the small amount of new operating system kernel support code required by our system is identical for both protocols. We also illustrate the ability of the resulting protocol implementations to handle real, demanding applications by presenting a demonstration of our DSR implementation transmitting real-time video over a multihop mobile ad hoc network including mobile robots being remotely operated based on the transmitted video stream. All video and robot control messages were transmitted over the ad hoc network running our DSR implementation.

The rest of this paper is organized as follows. In Section 2, we describe previous work in this field and how it differs from our work. Following that, in Section 3, we describe our system architecture, and in Section 4, we describe how our architecture is portable across multiple routing protocols and operating systems. We evaluated our architecture performance in Section 5. We discuss several issues with our architecture in Section 6, and we conclude in Section 7.

2. RELATED WORK

Simulation models of many ad hoc network routing protocols have been created in simulators such as *ns-2* [7], GloMoSim [28], OPNET [18], and QualNet [24], and several of these protocols have also been implemented in physical environments. We concentrate here on the different approaches that have been attempted in merging simulation and physical implementation efforts.

One approach in this area, for creating a new physical implementation of some protocol, is to base the new implementation on the existing code of a simulation model of the same protocol. For example, Royer and Perkins documented their efforts in using an existing *ns-2* simulation model of the AODV routing protocol as the basis for a new physical implementation of the protocol on Linux [23]; like our work, their implementation of the routing protocol runs in a single user-level process with interfaces to the kernel. They report that certain simplifications had to be made to the existing AODV simulation model. Their implementation also is not directly portable between different operating systems and supports only the AODV protocol. For example, they suggest a new FreeBSD virtual device driver to replace a Linux-only kernel interface used by the user-level process for kernel routing table updates; their Linux kernel modifications also interact closely with the kernel routing table data structures, which are different in different operating systems.

Another approach merging simulation and physical implementation efforts is the use of an existing physical implementation of some protocol as the basis for a new simulation model of the same protocol. For example, AODV-UU [15] is a physical implementation of AODV, which can also be executed within *ns-2* as a simulation model of AODV. However, AODV-UU supports only the AODV protocol and does not attempt to be portable to operating systems other than Linux for which it was designed.

More general projects in this area, providing frameworks that support arbitrary ad hoc network protocols rather than a single specific protocol, include the Rooftop C++ Protocol Toolkit (CPT) [22], the *nsclick* simulation environment [17], and the work of Allard et al. [1]. In CPT, protocols must be written within the proprietary CPT environment, which provides its own simulator, plus platform wrapper functions and device drivers for physical (and embedded) implementations. In *nsclick*, protocols must be written using the research Click Modular Router framework [13]; simulation may then be done using *ns-2*, although *nsclick* replaces much of the operation of *ns-2* to support this simulation; the resulting simulation does not, for example, support much of the tracing otherwise available within *ns-2*. The *nsclick* system also currently only supports CBR (UDP) traffic and does not support features such as link-layer link breakage detection. Allard et al. creates a new C++ framework for ad hoc network routing protocol implementation and also provides a new, integrated simulator for simple testing protocols implemented in this environment.

TOSSIM [14] provides a high fidelity simulation for TinyOS and mote hardware, such that TinyOS applications can be run in this simulation framework. The basic difference with our architecture is that TOSSIM was designed from scratch with the objective of easy portability of TinyOS applications from the simulation environment to the actual mote hardware, whereas our framework is more generic and provides for deploying any simulated ad hoc network routing protocol in *ns-2*. Moreover, unlike our architecture, TOSSIM does not model CPU time, thereby leading to a case in which code that runs in simulation will not run in a real mote due to non-handling of interrupts.

EmStar [8] is a software environment for developing and deploying wireless sensor network applications on Linux-based hardware platforms like iPAQs. However, as with TOSSIM, EmStar was designed from scratch to support easy migration from simulation to implementation.

In contrast to each of these previous projects, our work allows use of *existing*, *unmodified* protocol simulation models to create new physical protocol implementations. Unlike network *emulation* systems [12, 25, 26], our resulting physical protocol implementations are entirely real, not simulated. Whereas emulation systems simulate some aspects of the network behavior, for example to make a collection of stationary, wired nodes perform as if they were mobile and wireless, the physical protocol implementations produced from simulation code in our architecture run entirely on real hardware, with real packets and real wireless network interfaces; when executed, no aspect of the network and protocol performance is simulated, making the implementation suitable for detailed, realistic protocol testing and performance evaluation or for even for possible production application.

Our system supports protocol models from the widely used *ns-2* network simulator, rather than requiring use of new implementation environments, and thus retain all the

benefits of *ns-2* simulation, such as rapid prototyping and a widespread user community. Existing protocol modules can easily be used to create new physical implementations, and new protocols or modifications to existing protocols can easily be coded and tested in both simulation and physical implementation. Additionally, our design is portable to other protocol simulation systems.

3. SYSTEM ARCHITECTURE

Our architecture for creating physical protocol implementations from existing, unmodified simulation models consists of two parts: a single user-level process and a small amount of operating system kernel-level support. The user-level process executes the actual protocol implementation using the existing code for the simulation model for the protocol; we created an environment in this process in which this protocol simulation model can run unmodified, acting as it would if run inside the original simulator, but operating on real packets. In order for this module to interface with the real network, we introduce a small amount of kernel support that acts as a conduit inside the kernel to connect the simulation model in the user process to the physical network. The user process uses only standard network API (Application Programming Interface) calls to interface with this kernel support.

The packet flow through a node implementing our architecture is illustrated in Figure 1, for several different packet scenarios. A packet to be forwarded by the node is received at the operating system kernel *device driver* of the network interface hardware, and is then passed to the user level *routing protocol simulation model* via the *packet format converter*; the routing protocol then passes the packet back to the operating system kernel via the converter, and the kernel finally passes the packet to the device driver for transmission. For reception of a packet destined to an application running on this node, the routing protocol simulation model, after processing the packet, passes the packet back to kernel, which transfers it to the user application through the standard IP input function.

Although the description of our architecture is in terms of the *ns-2* network simulator, we discuss in Section 3.2.6 how this architecture can easily be applied to other network protocol discrete event simulators as well, such as GloMoSim [28], OPNET [18], and QualNet [24].

3.1 Kernel-Level Support

In order for the network to interact with the user-level protocol module in the simulation environment, we used a BSD Unix raw IP socket [27] for passing IP packets between the physical network and user-level protocol engine. Raw sockets pass the entire packet, including the IP header, in and out of the kernel, allowing the routing protocol to access and modify fields in the IP header as necessary. In addition, raw sockets provide queuing of packets between the kernel and user level. Finally, use of raw sockets improves portability of the resulting architecture, since they are a standard facility provided by the common Berkeley sockets interface available on many different operating systems including BSD Unix versions (FreeBSD, NetBSD, OpenBSD), Linux, Mac OS X, and Microsoft Windows [27, 20]; other interfaces such as netgraph [5] and System V Streams [2] are less widely available in different systems.

Changes in the kernel to support our system are small and exist mainly in IP input and output processing routines in order to interface with the raw socket inside the kernel.

For packets originated by an application running on an ad hoc network node, the packet is intercepted at the kernel IP output routine and passed back to IP input, where it is then passed up to the user-level routing agent through the raw IP socket. The routing protocol may then, for example, add a protocol-specific header to the packet or modify existing packet header fields. The packet is then passed back to the kernel through the raw IP socket, to be transmitted. This pass through the routing protocol simulation model is illustrated in Figure 1.

In order to determine the address of the next-hop node to which to forward a packet, some ad hoc network routing protocols do not use the traditional IP routing table. To allow the user-level routing protocol itself to determine the next-hop for a packet, the packet format converter may pass this information to the kernel by appending the IP address of the next-hop to the packet. If next-hop information is present, this value is used by the kernel's IP output handling rather than using the existing kernel IP routing table mechanism to determine the next-hop address. Appending this information to the packet instead of passing it to the kernel separately allows us to continue to use the widely available standard Berkeley sockets interface, as discussed above, and simplifies the implementation since the packet and the next-hop address are passed together to the kernel. The appended next-hop address is removed by the kernel before the packet is transmitted over the network.

This mechanism can be used by DSR to indicate the next hop of the source route for a packet, without the need for the kernel to know the format of the source route header. Other protocols such as AODV could use the kernel's routing table mechanism, but this creates problems for a user-level routing protocol implementation, since the protocol cannot correctly manage the contents of the kernel routing table by keeping track of the last time that each table entry was used [23]. By instead utilizing this new mechanism to allow the user-level routing process to completely manage the routing decisions, these problems can be avoided.

Also, many mobile ad hoc network routing protocols can take advantage of received signal strength information, for example to determine when the currently used route is about to break. Based on this information, the protocol can initiate a search for a new route to the destination while the current route is still active. This optimization, known as preemptive Route Maintenance [9], reduces or even eliminates latency in searching for a new route when the current route breaks. To support this or other uses of receive signal strength for a received packet, we modified the wireless network interface driver to append the received signal strength value to each incoming packet. This information is passed up with the entire packet to the user-level routing process, where the protocol can extract the signal strength information and determine appropriate actions.

Finally, in order to a detect broken link to the next hop, many mobile ad hoc network routing protocols take advantage of link-layer acknowledgments that already exist at the MAC layer. To support this, the device driver is modified to pass the link-layer transmission status to the user-level process. We discuss in Section 3.2.5 how the user-level process utilizes this information.

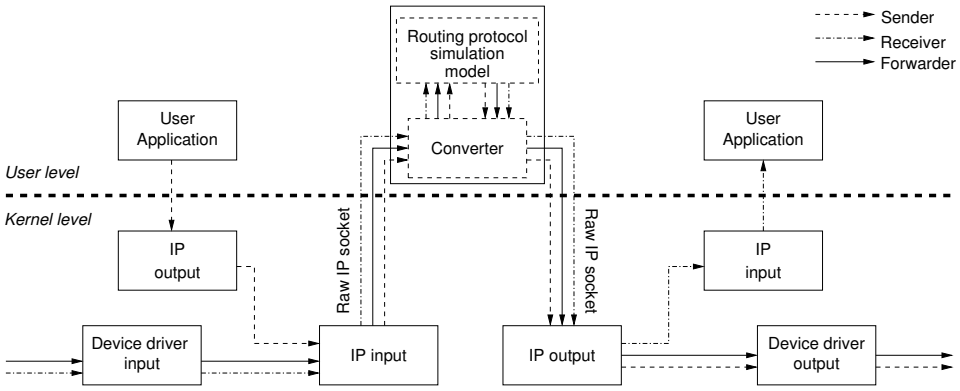

Figure 1: Flow of a Packet through a Node in our Architecture in Different Scenarios

3.2 User-Level Support

In our system, the entire routing protocol executes in a single user-level process, running the unmodified *ns-2* protocol simulation model to route real network packets.

3.2.1 Event Scheduler

In a discrete event simulator such as *ns-2* [7], the simulator maintains a queue of pending events to simulate, and maintains a global variable giving the current *virtual time* within the simulation. The event scheduler repeats a loop in which it finds in the event queue the event that should occur at the earliest scheduled time, removes that event from the queue, advances the global virtual time to the scheduled execution time for that event, and simulates the event. The time *between* event execution times is not simulated; rather, the global virtual time immediately advances to the time at which the next event is to occur.

We maintain that basic behavior, but we change the event scheduler to instead execute each event only once the *real time* (on the node itself) reaches the event's scheduled execution time. The rest of the event scheduler, including the event queue data structure and the interface to it, are not changed in any way; the simulation code still maintains its own queue of pending events to be executed, as if it were running in a standard simulation environment.

A similar type of real-time event scheduler is also used by network emulation systems [6, 12], as discussed in Section 2. However, in our system, the event queue contains only events that should occur at this node itself, whereas in network emulation, the simulation of all nodes occurs together in a single simulation, and the event queue is a common data structure holding the pending events of all nodes in the simulated system. Also, whereas the real-time event queue execution in network emulation systems may degrade emulation accuracy if the execution falls behind real time, no similar problem exists in our system, since our event queue schedules only software events that are part of the routing protocol execution; these events do not have strict real-time requirements, and the operation of the event queue simply replaces the timer event queue normally used by network protocol implementations inside the operating system [27].

3.2.2 Interaction with the MAC Layer

For wireless networks, *ns-2* uses the Monarch Project wireless extensions that provide a detailed simulation of the physical, link, and routing layers of the network [3]; other wireless simulators provide similar detailed lower layers, in order to accurately model the complex behaviors of these layers in real systems. In *ns-2*, when a packet is being sent by a mobile node, the routing layer schedules the packet to the link layer, which then schedules the packet to the Medium Access Control (MAC) layer and finally transmits the packet using the simulated physical layer. Similarly, when a packet is received at a mobile node from the simulated physical layer, the MAC layer schedules the received packet to the link layer, which then schedules that packet to the routing layer.

In our system, we do not use the simulated link layer, MAC layer, or physical layer, since these functions are provided by the real system in the operating system and in the real hardware. We explain the need for packet format conversion in Section 3.2.4. Our converter also exports those programming interfaces that the simulator expects, for example so that the routing layer can still make calls to those interfaces without knowing that it is running in our physical implementation environment rather than inside the actual simulator.

3.2.3 Reception of Packets

In addition to the basic event processing loop described above in Section 3.2.1, adapted from the existing event scheduler behavior of *ns-2*, we needed to handle the reception of packets from *outside* the simulation environment. Each node in the physical implementation runs its own copy of the simulation model of the ad hoc network protocol, and packets sent by one node to another are sent over the real network as real packets, rather than being handled internally as a normal *ns-2* event.

To integrate the reception of new packets from outside the simulation environment, we allow the receipt of such a new external packet to terminate the event scheduler's wait for the real time of the next scheduled event execution time. Specifically, the event scheduler loop blocks itself with the operating system until *either* the next scheduled event execution time arrives *or* an external packet arrives at this node that must be handled by the protocol.

When the kernel receives a packet that must be handled by the simulated protocol, the kernel uses a raw IP socket to send the packet to the packet format converter, which then sends the packet to the user-level protocol module. The handling of a received packet within the simulation code

can potentially generate other events that are inserted into the scheduler's event queue in the same way as other simulated events (in fact, they are generated by simulation code operating in the same way as if it were running inside the normal simulator).

3.2.4 Conversion between Packet Formats

Most simulators, including *ns-2*, use an abstract, internal packet format that is different from the native packet format, for ease accessing different packet headers and packet header fields in writing the simulation code for a protocol. For the simulator to work transparently in a physical implementation with real packets and with the existing, unmodified protocol simulation model code, an extra software layer must convert between abstract and native packet formats. On receiving an external packet from the kernel, this converter changes the packet from native format into the simulation abstract packet format; on transmitting a packet outside the node's simulation environment, this converter changes the packet from the simulation abstract packet format into the native packet format.

3.2.5 Transmission of Packets

When the user level protocol module needs to transmit a packet, the packet is received by the converter, which then converts the format of the packet from *ns-2* packet format to native format (dependent on the host byte order). The converted packet is then passed to the operating system kernel using the raw IP socket (the processing of this packet by the kernel is described in Section 3.1). Thus, the routing layer simulation code never needs to know the native packet format and is oblivious of how the lower layers handle packets that the routing layer sends or receives.

In addition, many ad hoc network routing protocols utilize link-layer acknowledgements (e.g., as in IEEE 802.11) to detect whether or not a transmitted packet is received by the intended next-hop node. For example, DSR uses this link-layer feedback for its on-demand Route Maintenance function [10]. In the real hardware and operating system device driver, this feedback is signaled from the hardware by a *packet transmission-complete* interrupt that occurs asynchronously after the packet has been transmitted. In our system, we pass the important information from this interrupt to the user-level protocol process, in a way that is compatible with the handling of this feedback by the simulated routing protocol.

In particular, *ns-2* passes to the simulated MAC layer a pointer to the internal *ns-2* packet data structure. If the packet cannot be successfully delivered to the next-hop node (as indicated by the link-layer feedback), this pointer is still available for the routing layer to use to access the original packet.

We maintain this same behavior, replacing the lower layers as present in *ns-2* with the real operating system and hardware. When transmitting a packet, the packet format converter passes the *ns-2* packet pointer into the kernel along with the packet through the raw IP socket, by appending the pointer value to the end of the native packet format. Inside the kernel, this *ns-2* packet pointer is removed from the packet and saved inside the kernel as an opaque value (the kernel does not use the pointer *as* a pointer). The attached packet pointer is not transmitted with the packet when sending the packet over the wireless network interface.

When the packet transmission-complete interrupt is received by the kernel, the kernel constructs an ACK (acknowledgement) or NACK (negative acknowledgement) packet to convey the success or failure status of this packet back to the simulated environment. The kernel looks up the saved (opaque) *ns-2* packet pointer that corresponds to the packet, appends that packet pointer value to the ACK or NACK packet, and sends it to the simulation environment through the raw IP packet in the same way as for other received packets.

Once the ACK or NACK packet reaches the converter in the simulation environment, the conversion routine calls an *ns-2* function that takes the appropriate action on the original packet (indicated by the appended *ns-2* packet pointer). For an ACK, the simulation code deletes the *ns-2* packet as normal; for a NACK, the simulation code has a reference to the packet, and the packet can be processed exactly as a failed packet is processed in unmodified *ns-2*.

3.2.6 Application to Other Network Simulators

A number of different discrete event simulators exist and have been used for simulating and evaluating ad hoc network routing protocols. Among the more frequently used are *ns-2* [7], GloMoSim [28], OPNET [18], and QualNet [24]. In Section 3.2, we described the implementation of our architecture for the *ns-2* simulator; our architecture can be used with other discrete event simulators as well, using the same techniques.

In particular, any discrete event simulator has an event scheduler loop similar to that discussed above, and the same mechanism we have used for *ns-2* can be used to modify that loop to support our system. The simulator already has its own data structures for maintaining the event queue, and its own procedures for adding events to the queue, removing events from the queue, and finding the next event to simulate from the queue. None of this needs to be modified in any way to apply our architecture to such a simulator.

Furthermore, network protocol simulators generally all follow a layered structure based on the standard 7-layer OSI network reference model and on the protocol layering in real operating systems. This makes it possible to replace their abstracted link-layer, MAC, and physical layers with the real operating system and hardware, through our interface to the kernel. If abstract packet formats are used in the simulator, as in *ns-2*, the same type of packet format converter can be used.

In general, since we have not disturbed the basic data structures and mechanisms used by the simulator, no changes are necessary within the source code for the protocol simulation model; the protocol model can be executed in a user-level process as we have done with *ns-2* without the simulation code being aware it is not running in the normal simulator. Although some simulators can execute simulation code on parallel machines to speed up simulation execution time, such simulators can also run in single-threaded mode in a single process. This is the only requirement for adapting the simulator and thus its protocol models to our architecture.

4. ARCHITECTURE PORTABILITY

To demonstrate the simplicity, portability, and effectiveness of our architecture, we present in this section an example implementation of two popular ad hoc network routing protocols, DSR and AODV, on two different operating sys-

tems, FreeBSD and Linux. We used the DSR and AODV models from *ns-2.26*, although, as mentioned in Section 3.2.6, our architecture can also be applied to other network protocol simulators.

The small amount of new kernel support required by our system is protocol-independent and hence is unaware of the actual protocol that is being implemented. Similarly, the routing protocol implementation is independent of the underlying operating system and hence is unaware of the operating system that the machine is running. For example, in our example protocol implementations described here, the user-level code for DSR and for AODV is identical between our implementations on FreeBSD and Linux, and the new kernel support code for FreeBSD and for Linux is identical for the two different routing protocols.

4.1 Portability across Multiple Protocols

In this section, we demonstrate the simplicity of supporting multiple ad hoc network routing protocols in our architecture by describing our example implementations of two popular protocols, DSR and AODV, and by discussing how this support extends to other routing protocols simulated in *ns-2*.

4.1.1 Example DSR Implementation

DSR [10] is an on-demand source routing protocol, with each packet containing a source route header. The DSR protocol consists of two mechanisms, Route Discovery and Route Maintenance, both of which operate entirely on demand. To perform a Route Discovery for a destination node D, a source node S broadcasts a ROUTE REQUEST packet that gets flooded through the network in controlled manner. This request is answered by a ROUTE REPLY from either D or some other node that knows a route to D. To reduce frequency and propagation of ROUTE REQUESTs, each node aggressively caches source routes that the node learns or overhears. Route Maintenance detects when some link over which a data packet is being transmitted has broken, and then returns a ROUTE ERROR to S. Upon receiving a ROUTE ERROR, S can use any other route to D that it has in its Route Cache, or S can initiate a new Route Discovery for D.

Support for a new protocol in our architecture requires only the addition of a new protocol-specific packet format conversion. Kernel support is both protocol-independent and simulator-independent, and the protocol module itself already exists in *ns-2*. We describe below DSR-specific considerations that the DSR conversion module needs to support.

The DSR source routing header follows the IP header, indicated by the value in the protocol number field in the IP header. To transmit a DATA packet, the converter needs to insert a DSR header into the packet, with the transport protocol header and its data becoming the payload following the DSR header. The converter also changes the packet's IP protocol field from the original transport protocol number to DSR, copying the original IP protocol value into the DSR header. Before the DATA packet leaves the DSR network or is delivered to the application at the final destination, it needs to be sent up to the conversion module. The conversion module removes the DSR header and reconstructs the original IP packet with the original transport protocol number as its IP protocol. Similarly, DSR control packets also have their own DSR header following the IP header. However, since DSR control packets are generated and freed by the DSR routing module, there is no non-DSR packet to modify. The converter only has to convert packets from *ns-2* format to native packet format.

4.1.2 Example AODV Implementation

AODV [19] forms hop-by-hop routes rather than source routes. When a source S needs a route to a destination D, S broadcasts a ROUTE REQUEST to its neighbors, containing its last known sequence number for D. The request is flooded throughout the network until it reaches a node that has a route to D. In this process, each forwarding node creates a *reverse route* back to S. Upon reaching a node with a route to D, the node replies back to S with a ROUTE REPLY containing the number of hops that D is from itself and the most recent sequence number for D known to the replying node. When a node forwards this reply, it creates a *forward route* to D by remembering the next-hop node towards D.

As with DSR, support for AODV in our architecture requires only addition of an AODV-specific packet format conversion module. However, AODV does not use its own protocol header; all AODV control packets are sent as UDP packets, and AODV DATA packets use only the standard IP header. Thus, the only requirement for the AODV conversion module is to directly convert between *ns-2* and native packet formats. The protocol-independent support in the kernel passes all packets arriving up to the conversion module and the routing code, even for packets addressed to this machine. This also allows the routing protocol to extract useful information from the packet about the network, either for protocol operation (such as for managing the routing table) or for logging or statistics within the protocol simulation code. After the AODV protocol module processes the packet and determines next hop information from its routing table, next hop information is passed from the protocol module to the converter to be sent to the kernel.

4.1.3 Support for Other ns-2 Routing Protocols

To be used with our architecture for creating physical ad hoc network routing protocol implementations, the *ns-2* simulation code for the protocol must meet a few simple requirements; these requirements are met by virtually all existing *ns-2* ad hoc network routing protocol models that we are familiar with and are easily met by new *ns-2* models.

For reception of unicast or broadcast packets, *ns-2* requires that the routing protocol module implements the *recv()* function, called by the MAC layer in *ns-2*. Our converter invokes the same function to pass packets to the protocol module.

In order to handle link layer transmission failures, *ns-2* requires the routing protocol simulation code to provide a callback function. For example, in the standard distribution of *ns-2*, DSR implements the function *XmitFailureCallback()* and AODV implements the function *aodv_rt_failed_callback()*. This callback function is required only for those protocols that respond to link layer transmission failures. If such a callback is provided, then the converter invokes the callback as needed.

Some routing protocols operate the network interface in promiscuous mode to overhear information contained in packets for other nodes. For such routing protocols, *ns-2* requires the routing protocol simulation code to implement

the *tap()* function. If this function is provided, the converter invokes it to pass promiscuously received packets to the protocol module.

Any *ns-2* ad hoc network routing protocol simulation model meeting the requirements above can be used *unmodified* in our architecture. Other routing protocols that utilize external information, such as a geographic routing protocol in which each node needs to know its own GPS coordinates, can also be used with minor modifications; for example, some mechanism must be added to make the current coordinates from a hardware GPS receiver available inside the protocol simulation code.

The only type of *ns-2* ad hoc network routing protocol simulation model that cannot be used with our architecture (without larger modifications) is one that relies on global knowledge present inside a simulation environment but not available in the real world. For example, if a node using the geographic routing protocol above also "magically" knows inside the simulation the current GPS coordinates of other nodes, by simply using the simulator's global knowledge of all nodes, then it will require modifications to adapt the simulation code to our architecture. However, relying on such global knowledge inside the simulation does not provide a complete simulation of the protocol and does not allow for fully accurate performance evaluation using this simulation. In this case, an additional protocol mechanism must be added to exchange the GPS coordinate values for other nodes using actual packets, and such an additional mechanism would be needed for a physical implementation of the protocol produced in any other way too.

4.2 Portability across Multiple Operating Systems

Support for different ad hoc network routing protocols in our architecture is OS-independent. User-level protocol code interfaces with the kernel only through the standard Berkeley socket programming interface, common to many operating systems, including FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, and Microsoft Windows. All OS-dependent features reside in a small amount of kernel modifications, and porting the implementation between different operating systems requires only changes to this code. We describe in this section the simplicity of supporting these kernel modifications in different operating systems, thus demonstrating the ability to port the architecture across operating systems.

In the FreeBSD kernel network protocol stack implementation, incoming IP packets from the link layer are processed in the *ip_input* function. A small modification in this function passes all IP packets, regardless of their destinations, through the raw socket to the user-level protocol module. For outgoing packets, after the protocol module sends each processed packet to the kernel with next-hop information, a modification in the outgoing IP function, *ip_output*, fills next-hop information with the value passed from the user-level routing protocol. For an outgoing packet that originates from a local application, the packet is intercepted at *ip_output* and redirected through *ip_input* to the raw socket, where it is passed up to the user-level protocol module for routing decision. In the Cisco 350 wireless LAN device driver (used in our implementation), where each outgoing Ethernet frame that encapsulates the packet is about to be transmitted by *an_start*, we associate the Ethernet frame identifier (ID) with the packet pointer (Section 3.2.5). When a transmission-complete interrupt occurs in the device driver, the *an_txeof* function is called with the ID of the Ethernet frame that has finished transmission. The frame ID is converted to its corresponding *ns-2* packet pointer, and a status notification for this packet is passed up through the raw IP socket to the user-level protocol process.

In the Linux kernel implementation, we made similar modifications, although in different function calls due to the differences in the Linux kernel. Incoming IP packet processing logic in Linux is divided into multiple functions. Specifically, a modification in *ip_route_input* treats all IP packets, regardless of their IP destinations, as if they were destined for the local node, so that they will be passed to the local user-level protocol implementation. The packet is then redirected to the raw socket interface in *ip_local_deliver_finish*. Similarly, outgoing IP processing logic in Linux is also divided into multiple functions. A modification in *ip_route_output* assigns a next-hop address for an outgoing IP packet. For an outgoing packet that originates from a local application, the packet is intercepted and passed up to the user-level protocol module for routing decision at the end of the outgoing IP processing logic, in *ip_finish_output*. Modifications in the Cisco 350 wireless LAN device driver in Linux happen in the same logical place as in FreeBSD. Here, *airo_interrupt* and *airo_do_xmit* correspond to functions *an_txeof* and *an_start*, respectively, in FreeBSD.

Our choice of FreeBSD and Linux to illustrate operating system portability is due to the fact that they are popular operating systems with freely available kernel sources, and not for any similarities between their codes. In fact, the FreeBSD and Linux kernel networking codes evolved from entirely different code bases. FreeBSD networking code evolved from the original Berkeley Extensions. The Linux networking stack, on the other hand, was intentionally separated from BSD code due to copyright issues with the BSD stack at the time. The Linux networking stack was originally developed, lead by Ross Biro, in 1992 [4]. The Linux networking stack, however, does share general similarities with FreeBSD, due to the common protocol layering defined by the standard protocols implemented.

5. SYSTEM DEMONSTRATION

We have tested our four example protocol implementations, for DSR and AODV on FreeBSD and Linux, for correct operation, and have verified that the resulting implementations are interoperable across the two operating systems. That is, our DSR implementation on FreeBSD functions correctly when operating in a network together with our DSR implementation on Linux, and likewise for our two implementations of AODV. In this section, we demonstrate that using our architecture with its user-level protocol implementation the resulting protocol implementations can support real-time applications with realistic traffic loads.

In order to validate the usability of our architecture for building physical implementations of ad hoc network protocols, and to demonstrate the resulting implementation of a protocol, we constructed a test network of mobile and stationary nodes in our department building. Our test network consisted of two mobile robots and four stationary ad hoc network nodes, with the robots remotely controlled based on real-time live video from each robot transmitted over the ad hoc network, using standard Microsoft Windows

NetMeeting video [16]. All video and robot control messages were transmitted over the ad hoc network with our protocol implementation. We show here the operation of our implementation of DSR on FreeBSD and omit for brevity demonstrations for configurations using AODV or Linux.

We now describe the design and operation of the different components of this network.

5.1 Wireless Nodes

Our test network included six IBM ThinkPad X31 laptops each running FreeBSD 5.1-RELEASE, modified as described in Section 3.1. Each of these laptops used a Cisco Aironet 350 IEEE 802.11b (11 Mbps) wireless LAN card as the wireless interface; we disabled the built-in IBM wireless LAN interface in each laptop and used the Cisco cards instead, since these cards allow the transmit power level to be modified. Of the six laptops, four were stationary, and two were mobile. By moving the mobile nodes changing multihop routes were created through a varying sequence of the stationary wireless nodes and through the other mobile node.

To create a multihop ad hoc network of more than a few hops within the limited physical space of our building, we reduced the transmit power level of the wireless network interfaces to 20 mW rather than the default 100 mW (reducing the transmission power level by a factor of 2 generally reduces the maximum transmission distance by at least a factor of 4) [21]. With this reduced transmit power level, our network created multihop routes of up to 5 hops in length. We validated during our demonstration that the traffic was using multiple hops for substantial parts of the demonstration period.

Each mobile node in our network was implemented as a robot, which we could control by software commands over the ad hoc network. We used the Koala robot manufactured by K-Team [11]. Each robot carried two laptops, one running Windows NetMeeting (with an attached camera) on Microsoft Windows XP Professional for traffic generation, and one running FreeBSD as the gateway to the ad hoc network.

5.2 Data Traffic Generation

We decided to send live video from each robot over the ad hoc network to a centralized control location. By watching the video from a robot there, it would be possible to remotely "drive" the robot by sending movement commands back to the robot over the ad hoc network. In addition to exercising and demonstrating the network, this approach also avoided the need to otherwise program intelligent control directly into the robot for autonomous motion. By using Windows NetMeeting for the video, we also demonstrate compatibility of our implementation with standard, unmodified IP-based applications, as we do not have the source code for either Windows or NetMeeting.

NetMeeting sends all video data packets using UDP. However, when a call is first placed, it uses TCP to setup a connection. Hence, we had to support both UDP and TCP data over our ad hoc network.

5.3 Demonstration Evaluation

The use of video and remote control of the robots created an engaging demonstration of our system's capabilities. In particular, in driving a robot, the user watches the video display closely to avoid driving the robot into any obstruction. If the video stops or is not clear, or if movement commands to the robot are not executed quickly (visible in the video display), the user immediately notices. Throughout the demonstration, the video display and robot control applications — and thus the ad hoc network and the protocol implementation using our system — worked very well.

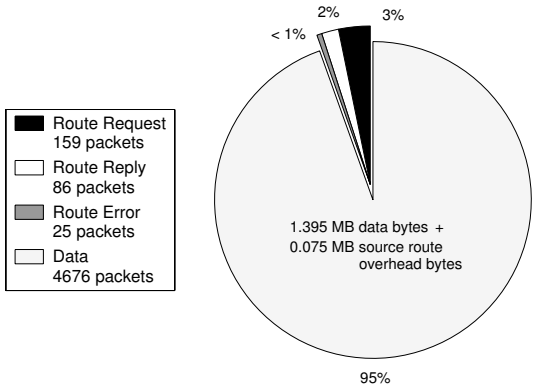

Figure 2: Packet Type and Overhead Distribution

We collected measurements during one run of our demonstration network in order to evaluate its performance. For simplicity, in this run, we used only a single robot, with the live NetMeeting video and remote robot control both being sent over the ad hoc network. During this run, the robot was remotely driven around the perimeter of the floor of our building and back to its starting position over a period of 13 minutes (780 seconds).

Figure 2 shows a summary of the types of packets transmitted during the demonstration run and the number of bytes of network overhead caused by each. Network overhead includes all ROUTE REQUEST, ROUTE REPLY, and ROUTE ERROR packets, as well as the DSR source route header in each data packet. In this figure, each transmission of an overhead packet (whether from the originator of the overhead packet or from a forwarding intermediate node) is counted separately.

Among ROUTE REQUEST, ROUTE REPLY, and ROUTE ERROR packets, the number of ROUTE REQUESTs is the greatest, since these packets are flooded through the network. The number of ROUTE REPLYs is greater than ROUTE ERRORs, since a Route Discovery is initiated from a single ROUTE ERROR, but this may result in the return of more than one ROUTE REPLY.

Figure 3 shows the Packet Delivery Ratio (PDR) for the entire run of the demonstration. The PDR is defined as the total fraction of application-level data packets originated that are actually received at the intended destination node. The horizontal dashed line shows the overall PDR for the entire demonstration run, and the solid line shows the PDR separately for each 10-second interval. There is a sharp dip in the PDR at around time 300 seconds, about half way through the demonstration run. At this time, the mobile robot was the farthest from the rest of the network and thus was experiencing temporary wireless signal fading. This behavior occurs in our physical implementation but is not modeled accurately in most purely simulation evaluations of ad hoc network protocols, since it depends on more realistic physical layer radio modeling than is usually done.

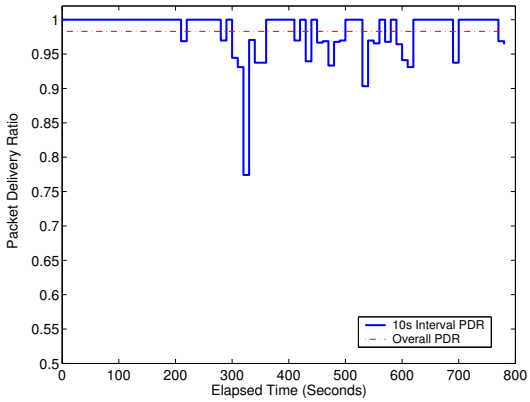

Figure 3: Packet Delivery Ratio

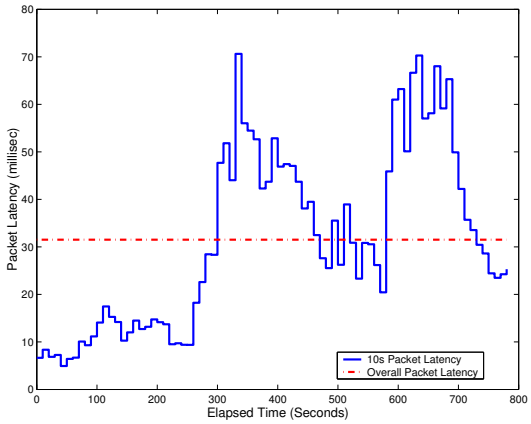

Figure 4: Packet Delivery Latency

Finally, Figure 4 shows the Packet Delivery Latency during this demonstration run. Packet Delivery Latency is measured only for application packets, and is defined as the time between originating a packet at the source node and receiving it at the destination node. The horizontal dashed line shows the overall Packet Delivery latency for the entire demonstration run, and the solid line shows it separately for each 10-second interval. As with the PDR, the Packet Delivery Latency is worse (increases) at around time 300 seconds when the robot was farthest away. This created longer routes for packets to travel from the robot to the destination node. Additionally, the weak signal strength at this location can cause additional RTS/CTS retransmissions due to dropped packets at the IEEE 802.11 MAC level, adding to Packet Delivery Latency.

6. DISCUSSION

6.1 Alternative Approach

A simpler approach to our proposed architecture is to remove the conversion module (Section 3.2.4). In this approach, the entire simulation packet is packaged as data inside an IP packet without being converted to its native format. Next-hop information is still passed down to the kernel and processed as specified in Section 3.1. When this IP packet reaches the next-hop node and arrives at the user process, the simulation packet is immediately available for the simulation protocol module to process. By removing the conversion module, new simulation protocol module can quickly be implemented since there is no protocol-specific implementation in either user or kernel space, aside from a small effort to retrieve next hop information from the simulation module to the kernel.

However, by not converting the simulation-specific format to native packet format, this approach prevents interoperability with other implementations of the same protocol. More importantly, this approach will significantly affect protocol behaviors due to the fact that sizes of different native packet types are replaced by sizes of simulation packets. In many simulations, for instance *ns-2*, all packets are represented by a common structure with different flags for different packet types. This common structure includes fields for all packet types, making the structure much larger than each native packet. Even in simulators where there are different structures for each packet type, they are often represented in formats such as a *class* or a *struct* with integer and array fields that are not nearly as compact as native packet formats. The effects of incorrect packet size on the protocol behaviors are especially magnified for wireless network protocols where contentions for the medium and allocated medium access greatly depend on packet sizes.

6.2 Applicability of the Architecture

Our architecture provides an effective way to validate experimental protocols on real physical networks. Since physical behaviors of wireless signals such as signal fading, multipaths, and delays are very difficult to model in simulations, our architecture is most beneficial for testing wireless network protocols. By employing a realistic wireless medium, the architecture can uncover issues in the protocol design related to the dynamic medium that cannot be accurately modeled in simulations.

For this particular implementation of the architecture, we focused on supporting routing protocols and implemented kernel modifications for packet interceptions at the network layer (IP layer). Thus, non-routing protocols cannot be used with this specific implementation of our architecture. However, support for protocol modules at higher layers of the networking stack (e.g., transport protocols) could be provided in a similar manner.

7. CONCLUSION

Typically network simulation and physical implementation of ad hoc network routing protocols are orthogonal to each other. Our architecture allows the protocol code to be written just once, and used in the simulation environment as well as in the physical implementation. Although the system is based on unmodified *simulation* code, the resulting physical implementation is entirely *real*, not simulated, running on real hardware, with real mobility, real packets, and real wireless network interfaces. This design saves implementation effort for the physical implementation, avoids introducing new bugs in the implementation, and eases later maintenance of the code. New protocol features and options can also be tested and evaluated first in simulation, and then moved without modification into the physical environment.

In this paper, we have described the architecture and created example implementations of DSR and AODV on FreeBSD and Linux from their existing *ns-2* models. The user-level code is *identical* between our implementations on

FreeBSD and Linux, and the small amount of new operating system kernel support code required by our system is *identical* for both protocols. We have also shown the feasibility of our approach by presenting a demonstration of our DSR implementation transmitting real-time video over a multi-hop mobile ad hoc network including mobile robots being remotely operated based on the transmitted video stream. All video and robot control messages were transmitted over the ad hoc network running our DSR implementation.

Throughout the paper, we have demonstrated the simplicity and portability of our architecture. With this architecture, experimental changes to the protocol can be quickly implemented at the user level. Additionally, a single change can be made in the simulation module that can be tested in both the simulator and validated in the real physical environment. By sharing code with simulation modules, the architecture retains many useful debugging and logging features that are common in simulation code. We plan to publicly release the source code for our system to allow other ad hoc network researchers to easily experiment with a variety of protocols in real physical implementations.

8. REFERENCES

[1] J. Allard, P. Gonin, M. Singh, and G. G. Richard. A User Level Framework for Ad Hoc Routing. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN 2002)*, pages 13–19, November 2002.

[2] AT&T. *Unix System V Streams Programmer's Guide*. Prentice-Hall, 1989.

[3] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta G. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pages 85–97, October 1998.

[4] Terry Dawson and Alessandro Rubini. A brief history of Linux Networking Kernel Development. Available at `http://www.sgmltools.org/HOWTO/NET-3-HOWTO/t151.html`.

[5] Julian Elischer. The Netgraph Networking System. Available at `http://www.elischer.org/netgraph/`.

[6] Kevin Fall. Network Emulation in the Vint/NS Simulator. In *Proceedings of the Fourth IEEE Symposium on Computers and Communications (ISCC'99)*, July 1999.

[7] Kevin Fall and Kannan Varadhan, editors. The *ns* Manual (formerly *ns* Notes and Documentation). The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC, November 2003. Available from `http://www.isi.edu/nsnam/ns/doc/`.

[8] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks. In *Proceedings of the 2004 USENIX Technical Conference*, 2004.

[9] Tom Goff, Nael B. Abu-Ghazaleh, Dhananjay S. Phatak, and Ridvan Kahvecioglu. Preemptive Routing in Ad Hoc Networks. In *Proceedings of the Seventh Annual Internation Conference on Mobile Computing and Networking (MobiCom 2001)*, pages 43–52, July 2001.

[10] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, edited by Tomasz Imielinski and Hank Korth, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[11] K-Team S.A. Koala Robot. `http://www.k-team.com/robots/koala/index.html`.

[12] Qifa Ke, David A. Maltz, and David B. Johnson. Emulation of Multi-Hop Wireless Ad Hoc Networks. In *Proceedings of the Seventh International Workshop on Mobile Multimedia Communications (MOMUC 2000)*, October 2000.

[13] E. Kohler, Robert Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click Modular Router. In *ACM Transactions on Computers Systems*, pages 18(30):263–297, August 2000.

[14] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 126–137. ACM Press, 2003.

[15] Henrik Lundgren and Erik Nordstr om. AODV-UU. `http://user.it.uu.se/~henrikl/aodv/`.

[16] Microsoft Corporation. Microsoft NetMeeting. NetMeeting home page: `http://www.microsoft.com/windows/netmeeting/`.

[17] Michael Neufeld, Ashish Jain, and Dirk Grunwald. Nsclick: Bridging Network Simulation and Deployment. In *Proceedings of the the Fifth ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2002)*, September 2002.

[18] OPNET Technologies. OPNET Modeler. `http://www.opnet.com/products/modeler/home.html`.

[19] Charles E. Perkins and Elizabeth M. Royer. Ad-Hoc On-Demand Distance Vector Routing. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[20] Bob Quinn and Dave Shute. *Windows Sockets Network Programming*. Addison Wesley, 1995.

[21] Theodore S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall, New Jersey, 1996.

[22] Rooftop Communications. The Rooftop C++ Protocol Toolkit (CPT). `http://web.archive.org/web/19980614083648/www.rooftop.com/rnd.shtml`.

[23] Elizabeth M. Royer and Charles E. Perkins. An Implementation Study of the AODV Routing Protocol. In *Proceedings of the Second IEEE Wireless Communications and Networking Conference (WCNC 2000)*, September 2000.

[24] Scalable Network Technologies. QualNet Family of Products. `http://www.scalable-networks.com/products/qualnet.php`.

[25] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[26] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 255–270, December 2002.

[27] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated Vol 2 The Implementation*. Addison Wesley, 1995.

[28] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.