Design and Performance of PRAN: A System for Physical Implementation of Ad Hoc Network Routing Protocols

Amit Kumar Saha, Khoa Anh To, Santashil PalChaudhuri, Shu Du, and David B. Johnson, *Member*, *IEEE*

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 performance of PRAN, a new system for the physical implementation of ad hoc network routing protocols that unifies these two types of evaluation methodologies. PRAN (Physical Realization of Ad hoc Networks) 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 PRAN of the DSR and AODV routing protocols in FreeBSD and Linux using the standard existing, unmodified *ns-2* simulation model of each. We illustrate the ability of the resulting protocol implementations to handle real, demanding applications by describing a demonstration with this DSR implementation transmitting real-time video streams over a multihop mobile ad hoc network; the demonstration features mobile robots being remotely operated based on the real-time video stream transmitted from the robot over the network. We also present a detailed performance evaluation of PRAN to show the feasibility of our architecture.

Index Terms—Computer system implementation, mobile communication systems, tools.

1 Introduction

PRAN (Physical Realization of Ad hoc Networks)[1] is a new system for the easy implementation of ad hoc network routing protocols. PRAN is motivated by the need in the community for a system by which ad hoc network routing protocols can be easily tested in a real-life system. The behavior of a real ad hoc network can be quite dynamic, as the wireless nodes in the network cooperate to forward packets for each other to allow nodes not within direct wireless transmission range of each other to communicate. Factors such as node movement and variations in radio propagation conditions can create frequent, rapid changes in network topology, presenting a challenging environment for operation of the ad hoc network routing protocol.

Ad hoc networking is currently a very active area of research, yet evaluating the many proposed routing protocols for ad hoc networks remains difficult. Currently, most ad hoc network routing protocol designers simulate new protocol designs using one of the commonly available network simulators such as *ns-2* [4]. Only a handful of those designs are actually implemented and tested in a real system using real radios and actual mobility. Although simulation and physical implementation are each valuable as techniques in evaluating ad hoc network routing protocols, a full evaluation of some protocol normally requires two separate implementations of the protocol, one for the simulation model and one for the real physical implementation. Such an approach requires substantial extra effort in coding, debugging, validation, and maintenance.

In contrast, PRAN allows *existing* simulation models of ad hoc network routing protocols to be used—*without modification*—to create a real, physical implementation of the same protocol. The ad hoc network routing protocol functionality at each node is entirely defined by the existing simulation code, but when run under PRAN, the protocol executes in reality with the node sending and receiving real packets as a real node in the physical ad hoc network. PRAN is designed to take advantage of the large base of existing simulation code, as well as the ease of implementation of new simulation code, but to move beyond simulation-only evaluation of the many new and proposed ad hoc network routing protocols. In addition, by sharing code with simulation modules, PRAN retains many useful debugging and logging features that commonly exist in simulation code.

In our implementation of PRAN, the routing protocol code on each node in the network runs in a single user-level process on that node and uses standard interfaces to transmit and receive packets from the kernel, simplifying protocol debugging and easing portability between different host operating systems. To support PRAN on a new operating system, a small amount of new operating system kernel support code is required to redirect packets between the kernel network stack and the user-level protocol

1. In Sanskrit, "pran" means "life" or "coming to life."

- *A.K. Saha, S. PalChaudhuri, S. Du, and D.B. Johnson are with the Department of Computer Science, Rice University, 6100 Main St. MS-132, Houston, TX 77005-1892. E-mail: {amsaha, dushu, dbj}@cs.rice.edu, santapc@gmail.com.*
- *K.A. To is with the Department of Electrical and Computer Engineering, Rice University, 6100 Main St. MS-132, Houston, TX 77005-1892. E-mail: takhoa@alumni.rice.edu.*

Manuscript received 25 July 2005; revised 9 June 2006; accepted 23 Aug. 2006; published online 15 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0216-0705.

process. In creating a physical implementation of some new ad hoc network routing protocol from its simulation model, only a straightforward packet format converter must be written to convert between the native packet formats used over the real network and the abstract packet formats that are typically used inside existing simulation environments; the actual *behavior* of the routing protocol has already been defined by the existing simulation model.

Our current PRAN implementation is based on the *ns-2* network protocol simulator, but the techniques used in PRAN should also be readily portable to other network simulation environments. Additionally, while the implementation of PRAN is designed for wireless ad hoc network routing protocols, the basic PRAN architecture can be extended to other types of protocols, such as transport protocols or wired network routing protocols.

We have evaluated the simplicity and portability of PRAN across multiple routing protocols and multiple operating systems through example implementations in PRAN of the Dynamic Source Routing protocol (DSR) [8] and the Ad hoc On-Demand Vector Routing protocol (AODV) [26] on FreeBSD and Linux using the standard existing, unmodified *ns-2* simulation model of each. The user-level code in these implementations is identical between FreeBSD and Linux, and the small amount of new operating system kernel support code required by PRAN is identical regardless of the routing protocol. We also illustrate in this paper 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 real-time 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 elaborate on the motivation behind PRAN, and in Section 3, we compare PRAN to previous efforts in this field. In Section 4, we describe the PRAN system architecture, and in Section 5, we discuss the protocol and operating system portability of this architecture. We evaluate PRAN's performance and describe a demonstration of its operation in Section 6. Section 7 discusses several issues with our architecture, and Section 8 presents conclusions.

2 Motivation

There are many advantages inherent to evaluating network protocols using simulation. 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 of the protocol in a testbed, since simulation avoids the need for actually 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 in simulation the complexities of real radio propagation, realistic node mobility, and application data traffic workload. Kotz et al. [13] demonstrated that experimental results can differ substantially from simulation results, since simulation environments often make significant, unrealistic assumptions about the environment in which wireless transmissions are carried out.

Physical protocol implementation, on the other hand, allows the real system itself to be measured and can help to validate simulations, but protocol evaluation using physical implementation is generally much more difficult than simulation-based evaluation. For example, physical implementation must deal with real packet formats and application programming interfaces (APIs), whereas such factors can be simplified and abstracted in simulation. In addition, evaluation using real physical implementation is generally much more time and equipment-intensive than is simulation 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.

With our PRAN (Physical Realization of Ad Hoc Networks) architecture, we have designed a system to allow protocol evaluations to utilize *both* simulation and physical implementation with little extra effort. By writing the protocol behavior *once* in simulation code, the same code can be used *without modification* to also allow testing and experimentation of the protocol as part of a real ad hoc network. For example, as the design and development of a new ad hoc network routing protocol progresses, the same code can be used directly for simulation-based and physical implementation-based evaluation, allowing new features or designs to be tested in the convenience of simulation and then easily moved into the complex reality of real mobile nodes and real radios. By using the *same* code between simulation and physical implementation, our approach also simplifies the validation of simulation results against reality.

3 Related Work

Simulation models of many ad hoc network routing protocols have been created in simulators such as *ns-2* [4], GloMoSim [34], OPNET [23], and QualNet [31], and physical implementations of several of these protocols have also been created. We concentrate here on the different approaches in merging simulation and physical implementation.

One approach to merging simulation and physical implementation efforts is to start with the existing code of a simulation model of some protocol and to modify it to create a new physical implementation of that 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 [29]. Like our work, their implementation of the routing protocol itself runs in a single user-level process with interfaces to the kernel. They report that many modifications were required to the AODV protocol design and to the Linux kernel in creating their implementation, some due to simplifications that had been made in the existing AODV simulation model. Their implementation also is not directly portable between different operating systems and supports only the AODV protocol; although they state plans to create FreeBSD Unix and Windows implementations based on their work, significant modifications will be necessary. For example,

they suggest a new FreeBSD virtual device driver to replace the 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 to merging simulation and physical implementation is to start with an existing physical implementation of some protocol and to modify it to create a new simulation model of the same protocol. For example, AODV-UU [17] is a physical implementation of AODV that 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, supporting arbitrary ad hoc network routing protocols rather than only a single specific protocol, include the Rooftop C++ Protocol Toolkit (CPT) [28], the *nsclick* simulation environment [22], 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 separate Click Modular Router framework [12]; simulation can then be done on *ns-2*, but unlike PRAN, *nsclick* replaces most of the standard operation of *ns-2* and is not compatible with the full *ns-2* environment. Allard et al. created a new C++ framework for the physical implementation of ad hoc network routing protocols and also provided a new, integrated simulator for the simple simulation-based testing of protocols implemented in this framework.

TOSSIM [15] provides a high fidelity simulation for TinyOS and sensor mote hardware, such that TinyOS applications can be run in this simulation framework. The basic difference with PRAN is that TOSSIM was designed from the beginning to support simulation of real TinyOS applications that are written to operate on real hardware, whereas PRAN is instead designed to support protocol implementation based on simulation models that are easier to implement than real application code; PRAN is designed to leverage the large existing base of *ns-2* simulation protocol code and to allow easy transition from simulation to real physical protocol implementations. Moreover, unlike PRAN, 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 nonhandling of interrupts.

EmStar [5] is a software environment for developing and deploying wireless sensor network applications on Linux-based hardware platforms like iPAQs. EmStar supports the execution of the resulting applications in real sensor networks as well as in simulation. However, as with TOSSIM, EmStar was designed from the beginning to support the implementation of applications that execute on real hardware, whereas PRAN is designed to work with existing simulation models and to allow easy transition of them from simulation to real physical protocol implementations.

Liu et al. [16] present a testbed that generates traces from real experiments and then feeds these traces to the simulator. In this manner, the simulation is subject to identical network conditions as was present when the traces were collected in the real experiment. This is a notable contribution toward introducing more realism into a simulation environment. However, the simulator, called SWAN, developed by the authors, has been designed from scratch, keeping in mind the importance of integrated simulation and implementation. Hence, this approach cannot be applied to the large class of existing simulation environments. In contrast, PRAN provides a generic architecture in which any event-driven simulator can be augmented to an integrated simulation and implementation platform. Moreover, SWAN, unlike PRAN, uses IP tunneling to forward packets between devices, thus prohibiting compatibility with any other device running a different native implementation which uses packet formats as defined in the standard for the protocol under consideration.

PRAN also shares some similarity to network emulation systems [3], [11], [18], but unlike network emulation, the resulting protocol implementation under PRAN executes entirely as a *real* system. Although the protocol behavior at each node in PRAN is defined by the existing unmodified simulation code, each node executes independently, entirely on the real mobile hardware with real radios and real packets, in the same way as it would using any other technique for physical implementation of a real ad hoc network. Additional details on network emulation are discussed in Section 4.3.

In contrast to each of these previous projects, PRAN allows the creation of new physical protocol implementations from *existing*, *unmodified* protocol simulation models, rather than requiring the use of new implementation environments. Specifically, in our current implementation, we support protocol models from the widely used *ns-2* network simulator 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. We demonstrate that PRAN can be ported to multiple routing protocols and operating systems. Additionally, we believe the basic PRAN architecture could be applied easily to other protocol simulation systems such as GloMoSim [34], OPNET [23], or QualNet [31].

4 PRAN System Architecture

The PRAN architecture consists of two parts on each ad hoc network node: a single user-level process and a small amount of operating system kernel support. The user-level process executes the protocol implementation on the node, using the existing code for the simulation model of the protocol. In this user-level process, PRAN provides an environment in which this protocol code can run unmodified, acting for that node as it would inside the original simulator, but operating on real packets and using real radios. This environment is composed of an event scheduler, support for handling asynchronous receipt of packets, transmission of outgoing packets, and a packet format converter that serves to translate between the protocol simulation packet format and the native network

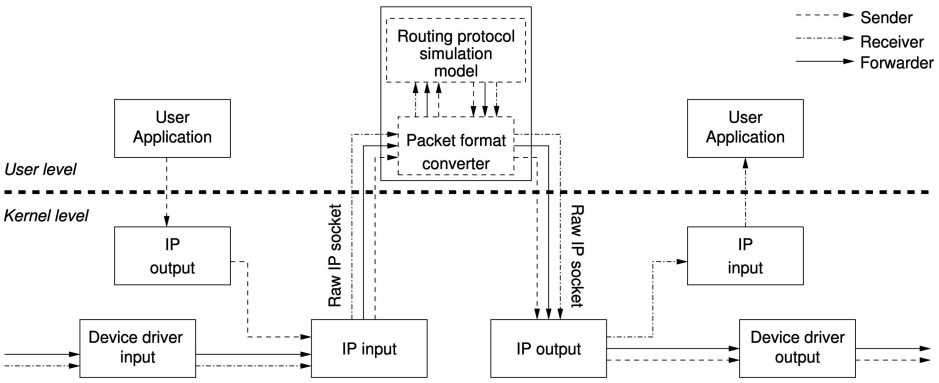


Fig. 1. Flow of a packet through a node in PRAN in different scenarios.

packet format. For this module to interface with the real network, we introduce a small amount of kernel support to connect the protocol code in the user process to the physical network. The user process uses only standard network socket API (Application Programming Interface) calls to interface with this kernel support.

The packet flow through a node implementing PRAN is illustrated in Fig. 1 for several different packet scenarios. A packet to be forwarded by the node is received at the operating system kernel *device driver* for 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.

4.1 Kernel-Level Support

In order for the network to interact with the user-level protocol module, we used a standard raw IP socket [33], which allowed us to pass whole IP packets between the physical network and user-level protocol engine. We used a raw IP socket for this for several reasons. Raw sockets provide a standard interface for passing the payload and full headers of IP packets in and out of the kernel. This allows the routing protocol to manipulate the IP header and routing protocol-specific extension headers (not understood by the kernel) and to send and receive separate routing packets (such as a ROUTE REQUESTs and ROUTE REPLYs). Additionally, raw sockets provide socket buffering when sending packets between the kernel and user level. The raw socket interface is also widely available as a standard part of the BSD Sockets API, available in many operating systems including UNIX, BSD, Linux, and Microsoft Windows. Thus, the use of raw sockets simplifies operating system portability. We chose not to use a memory-mapped interface for passing packets between the kernel and user-level protocol process, since the packets already reside in the IP stack and the use of a raw IP socket is thus simpler. Although memory mapping would allow additional information to be easily passed along with each packet between kernel and user level, this technique would not provide packet queuing. Furthermore, in UNIX-like systems, the use of *copyin* and *copyout* requires the size of the data to be known, but IP packets vary widely in size. We also chose not to use any of the existing specialized kernel interfaces for user-level processes interacting with packets in the kernel networking stack, such as Netgraph [9] or TUN/TAP [14], since these interfaces are not as widely supported in different operating systems as are standard raw IP sockets. Netgraph is generally only available on FreeBSD, and implementations of TUN/TAP are currently available only for Linux, Solaris, and FreeBSD; raw IP sockets, on the other hand, are available on essentially all systems supporting the BSD Sockets interface, including these systems and others such as Microsoft Windows.

Changes in the kernel to support PRAN are small and exist mainly in IP input and output processing routines in order to support the user-level routing protocol implementation.

Since most routing protocols potentially need to process packets received by a node for which the IP destination address indicates a different final destination (such as in the case of ROUTE REPLY, ROUTE ERROR, and route forwarding for DSR and AODV), it is necessary for the IP input routine to pass all incoming packets to the user-level routing agent, regardless of the IP destination.

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 path is illustrated in Fig. 1.

In order to determine the address of the next-hop node toward the destination 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, using its own algorithm or data structures, the packet format converter in the user-level routing process may pass this information back to the kernel along with the packet to be forwarded by appending the determined next-hop IP address to the packet when writing the packet into the raw IP socket to the

kernel. If next-hop information is present, this value is used by the IP output routine rather than using the existing kernel IP routing table mechanism to determine the next-hop address. The decision to append this additional information to the packet instead of passing it separately is closely related to our decision discussed above to use a raw IP socket. When the packet is passed between the kernel and user level through the raw IP socket, if information associated with that packet is passed separately, some coordination mechanism would be required to associate the information with the packet within the kernel, increasing the complexity of the PRAN kernel support code.

Appended next-hop information 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 DSR source route header. Other protocols such as AODV could use the kernel's routing table mechanism, but this may create problems for a user-level routing protocol implementation; for example, such an implementation of AODV could not correctly manage the contents of the kernel routing table by keeping track of the last time that each table entry was used [29]. By instead utilizing this new mechanism to allow the user-level routing process to completely manage the routing decisions, we avoid such problems.

Another area of change we made was to allow the ad hoc network routing protocol to take advantage of received signal strength information for each receive packet, 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 [6], [7], can reduce or even eliminate 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 through the raw IP socket, 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, we modified the wireless device driver to pass the link-layer transmission status to the user-level process. We discuss in Section 4.2.5 how the user-level process utilizes this information.

4.2 User-Level Support

In this section, we describe the environment we created running on each node in the ad hoc network, in which unmodified *ns-2* protocol simulation models could be executed, interfacing to the kernel to process real packets being sent, received, or forwarded by this node. The result is that the *unmodified* protocol simulation code acts as the physical implementation code as well.

4.2.1 Event Scheduler

In a discrete event simulator such as *ns-2* [4], 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.

In PRAN, we maintain that basic behavior, but a separate copy of this procedure executes at each node with the event queue containing only events to occur at that node. In addition, we change the event scheduler to instead operate in *real time*. That is, rather than immediately advancing the global virtual time to the next scheduled event time, the scheduler waits until *real time* (on the node itself) reaches the next scheduled event execution time; that event is then removed from the event queue and executed normally. The global virtual time is thus equal to the actual current real time at the beginning of each event. The rest of the event scheduler, including the event queue data structure and the interface to it by the protocol code itself, are not changed in any way; the simulation code still maintains this queue of pending events to be executed as if it were running in a standard simulation environment.

4.2.2 Interaction with the MAC Layer

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, which 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. Other wireless simulators provide similar detailed lower layers in order to accurately model the complex behaviors of these layers in real systems.

However, in PRAN, 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. So that the routing protocol layer can still interact with these lower layers without knowing that it is running in our PRAN environment rather than inside the actual simulator, we support the programming interfaces that the simulator expects for these functions. These programming interfaces are exported by the packet format conversion module described in Section 4.2.4.

4.2.3 Reception of Packets

In addition to the basic event processing loop described above in Section 4.2.1, adapted from the existing event scheduler behavior of *ns-2*, PRAN must handle the receipt of packets from outside the simulation environment. Each node in the physical implementation under PRAN runs its own copy of the simulation model of the ad hoc network routing protocol, and packets sent by one node to another

are sent over the real network as real packets, rather than being handled internally as normal *ns-2* events.

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 to reach the scheduled execution time of the next event in the event queue. 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. If a packet arrives before the next scheduled event time, we handle that packet then. This handling of the packet can potentially generate other events that are inserted into the scheduler's event queue in the same way as other simulated events in a normal simulation can (in fact, they are generated by the simulation code operating in the same way as if it were running inside the normal simulator). If, however, real time reaches the next scheduled event time first, then this existing event in the simulator event queue is removed from the queue and executed (in the same way as if it were running inside the normal simulator).

When the kernel receives an external packet that must be handled by the simulated protocol, the kernel uses the raw IP socket to send the packet to the user-level protocol process.

4.2.4 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 in accessing different packet headers and packet header fields in writing the simulation code for a protocol. For the simulator to work transparently under PRAN 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 its native format into the simulator's abstract packet format; on transmitting a packet outside the node's simulation environment, this converter changes the packet from the simulator's abstract packet format into the native packet format.

4.2.5 Transmission of Packets

When the user level protocol module needs to transmit a packet, the packet is received by the packet format converter, which then converts the format of the packet from the *ns-2* packet format into the native format (dependent on the host byte order). The converted packet is then sent to the operating system kernel using the raw IP socket (the processing of this packet by the kernel is described in Section 4.1). Thus, the routing layer 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, as noted in Section 4.1, 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 [8]. In the real hardware and operating system device driver, this feedback is signaled by an interrupt that occurs asynchronously after the packet has been transmitted. In order to support this link-layer feedback feature of the ad hoc network routing protocol in the unmodified simulation code, this asynchronous *packet transmission complete* interrupt, signaling the success or failure of the transmission attempt, must be passed to the simulation environment in a way that is compatible with the handling of this feedback by the routing protocol simulation code.

In particular, in a simulation under *ns-2*, a pointer to the *ns-2* packet data structure is passed down to the simulated MAC layer. If the packet cannot be successfully delivered to the next-hop node (as indicated by the simulated link-layer feedback), this pointer is still available for the simulated routing layer to use to access the original packet. Replacing the lower layers as present in simulation under *ns-2* with the real operating system and hardware does not allow us to directly do this in the same way.

We solve this problem by appending the *ns-2* packet pointer to the end of the native packet whenever an *ns-2* packet is converted to native packet format. When the packet is passed to the kernel through the raw IP socket, this *ns-2* packet pointer is saved inside the kernel as an opaque value (the kernel does not use the pointer *as* a pointer). The attached *ns-2* packet pointer is not transmitted with the packet when sending the packet over the wireless network interface. Instead, it is simply saved by the kernel until the packet transmission complete interrupt is received by the network device driver.

When this 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 simulation code in the user-level process. The kernel looks up the saved (opaque) *ns-2* packet pointer that corresponds to the delivered (or undelivered) native packet, appends that packet pointer value to the ACK or NACK packet, and sends the ACK or NACK packet 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 packet format 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). If it is an ACK packet, then the *ns-2* code deletes the *ns-2* packet as normal; if, however, it is a NACK packet, then the *ns-2* code has a reference to the packet and the packet can be processed exactly as a failed packet is processed in a standard simulation under unmodified *ns-2*.

4.2.6 Application to Other Network Simulators

As mentioned previously, 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 simulators are *ns-2* [4], GloMoSim [34], OPNET [23], and QualNet [31]. In Section 4.2, we described the implementation of PRAN for the *ns-2* simulator. We believe that PRAN can be applied to other discrete event simulators as well.

In particular, any discrete event simulator has an event scheduler loop similar to that discussed in Section 4.2.1 and the mechanism described in Section 4.2.3 can be used to modify that loop in the same way as we have done for *ns-2*. 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 PRAN 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 structure makes it possible to replace their abstracted link, MAC, and physical layers with the real operating system and real 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.

4.3 Comparison of PRAN Physical Implementation to Simulation and Emulation

Although a protocol physical implementation in PRAN is created from an *unmodified* simulation model of that protocol, the resulting physical implementation is entirely real, not simulated or emulated. The protocol implementation operates on real packets, using the real network, executing on real mobile nodes. *Only* the routing functions of the network layer protocol are implemented within PRAN. The rest of the network layer (e.g., processing of the IP packet header) executes normally in the original host operating system's protocol stack implementation. Everything below the network layer and everything above the network layer likewise executes normally in the real network interface hardware, in the original host operating system's protocol stack implementation, and in real application programs running on each node in the ad hoc network.

A similar type of *real-time* event scheduler is also used by network emulation systems [3], [11], [18]. In network emulation, a single simulation on a centralized machine executes the behavior of all nodes in a simulated ad hoc network and simulates events for all nodes; some of those nodes also represent real machines. When a real machine sends a packet, it is intercepted by the centralized simulation machine and injected into that simulation as a new event. The simulation then controls the behavior of that packet. When the packet reaches another simulated node that represents a real machine, the packet is transmitted again onto the physical network to be received by the destination machine. In contrast, with PRAN, each node executes independently; the event queue inside the PRAN user-level protocol process at any node contains only events that should occur at that node itself; there is no centralized simulation machine.

5 PRAN Architecture Portability

To demonstrate the simplicity, portability, and effectiveness of PRAN, we describe in this section the example implementation under PRAN of two ad hoc network routing protocols, DSR and AODV, on two different operating systems, FreeBSD and Linux. In this evaluation, we use the DSR and AODV models from version 2.26 of *ns-2*. However, as mentioned in Section 4.2.6, the PRAN architecture is general purpose and should be able to also be applied to other network protocol simulators.

The small amount of new kernel support required by PRAN is protocol-independent and, hence, is unaware of the actual routing protocol that is being implemented. Similarly, the user-level 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 protocol process code is *identical* between our implementations on FreeBSD and Linux, and the new kernel support code is *identical* for both DSR and AODV.

5.1 Portability across Multiple Protocols

We next describe our example implementations under PRAN of the DSR and AODV ad hoc network routing protocols, and we discuss how the support for other protocols, simulated in *ns-2*, is similar.

5.1.1 Example DSR Implementation in PRAN

DSR is an on-demand source routing protocol in which each packet sent contains a source route. The DSR protocol consists of two mechanisms, Route Discovery and Route Maintenance, which both operate entirely on demand. To perform a Route Discovery for a destination node **D**, a source node **S** broadcasts a ROUTE REQUEST that is flooded throughout the network in a 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 the frequency and propagation of ROUTE REQUESTs, each node aggressively caches source routes that it learns or overhears. Route Maintenance detects when some link over which a packet is being transmitted breaks. When a node forwarding a packet detects that the next hop link along the route is broken, the node sends 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 PRAN requires only the addition of a new protocol-specific packet format conversion. Kernel support is protocol and simulator-independent, and the protocol module itself already exists as a part of the *ns-2* source. We describe below DSR-specific considerations that the DSR packet format conversion module must support.

The DSR protocol uses a small extension header after the IP header in the packet (and before the transport layer header such as TCP); the presence of this DSR header is indicated by the protocol number field in the packet's IP header. To transmit a new data packet, the DSR code builds this DSR header but does so in *ns-2*'s abstract packet format. The packet format converter then converts this to native format before the packet is actually transmitted; the physical DSR header is inserted into the packet following the IP header, the current value of the IP protocol number field is copied into the DSR header, and the IP header's

protocol number field is modified to indicate that the DSR header follows the IP header. When receiving a data packet, the DSR header is converted back into the *ns-2* abstract format. Similarly, DSR control packets also use a DSR-specific header following the IP header. However, since DSR control packets are generated and freed only within the DSR routing module, there is no existing non-DSR packet to modify. The converter only has to convert packets from *ns-2* format to native packet format. For packets being sent following a DSR source route, the DSR protocol code determines the next hop to which to forward the packet and passes this information to the converter to be sent to the kernel with the packet.

5.1.2 Example AODV Implementation in PRAN

AODV is similar to DSR, but each node along a route in AODV determines the next hop based on its own routing table state, whereas, in DSR, the sender determines the entire route from only its route cache. When a source node **S** in AODV needs a route to a destination node **D**, node **S** broadcasts a ROUTE REQUEST to its neighbors. This request contains the last known sequence number for **D**. The request is flooded throughout the network until it reaches a node that has a fresh route to **D**. In this process, each forwarding node also remembers the previous hop to create 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 toward **D**.

As with DSR, support for AODV requires only the addition of a packet format conversion module. Unlike DSR, however, data packets in AODV are sent as normal IP packets with no AODV-specific header, whereas AODV control packets are sent as UDP packets to a reserved UDP port. Thus, the AODV conversion module can convert directly between *ns-2* and native packet formats, and it need not insert or remove any extra header in data packets. In order to maintain protocol-independence in the support code in the kernel, all packets arriving at a node, including data packets, are passed up to the conversion module and the user-level routing protocol code before being forwarded or delivered to the application or to an external network. Although an AODV data packet *could* be routed using the standard routing table in the kernel (if it was configured with the proper routing table entries), passing all packets instead to the user-level routing protocol code allows the *ns-2* AODV code to correctly maintain extra information in the routing table entries, such as the last time each entry was used, so that the expiration of unused entries can be properly handled by AODV. This approach also allows the routing protocol to extract useful information from the packet about the network, either for protocol operation or for logging or statistics within the routing protocol simulation code. After the AODV protocol module processes the packet and determines the next hop from its routing table, the next-hop information is passed from the routing protocol module to the converter to be sent to the kernel.

5.1.3 Support for Other ns-2 Routing Protocols

The unmodified *ns-2* simulation code for any ad hoc network routing protocol can be used directly in PRAN as long as the simulation code implements the programming interfaces described below that are a normal part of *ns-2*. For example, in addition to the *ns-2* source code for DSR and AODV, we have examined the source code to the *ns-2* simulation models for DSDV [25], TORA [24], and TBRPF [2] and found that they also implement these interfaces.

For reception of unicast or broadcast packets, *ns-2* requires that the routing protocol module implements the *recv()* function, which is 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 simulation code for the routing protocol 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()*. Such a callback function is required only for those protocols that respond to link layer transmission failures. If such a callback is provided, then the converter can invoke the callback to notify the protocol module of transmission failures.

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 simulation code for the routing protocol to implement the *tap()* function. If this function is provided, the converter can invoke the function to send promiscuously received packets to the protocol module.

5.2 Portability across Multiple Operating Systems

Support for different ad hoc network routing protocols such as DSR and AODV in PRAN is OS-independent. The user-level PRAN protocol code interfaces with the kernel through a standard BSD socket programming interface. The socket interface is common to many operating systems, including most UNIX-based systems (e.g., FreeBSD and Linux) and Microsoft Windows (with Winsock). All OS-dependent features reside in a small amount of kernel modifications, and porting the PRAN implementation between different operating systems requires only changes to this code.

General kernel modifications for the PRAN architecture were described in Section 4.1. In Fig. 2, we show examples of where the modifications are located within the FreeBSD 5.1 and Linux 2.4.20 (RedHat 9.0) kernel code. Fig. 2a identifies the protocol layers present in common network protocol stack implementations, e.g., based on the OSI Reference Model. Figs. 2b and 2d show the relevant operation in the FreeBSD and Linux kernels, respectively, for each protocol layer, for incoming and outgoing packets. Between these, Fig. 2c shows the kernel modifications made for PRAN; shaded boxes here represent modifications for incoming packets and white boxes represent modifications for outgoing packets. Each box in Fig. 2c is aligned with the actual functions in FreeBSD and Linux in Figs. 2b and 2d, where the modification is made.

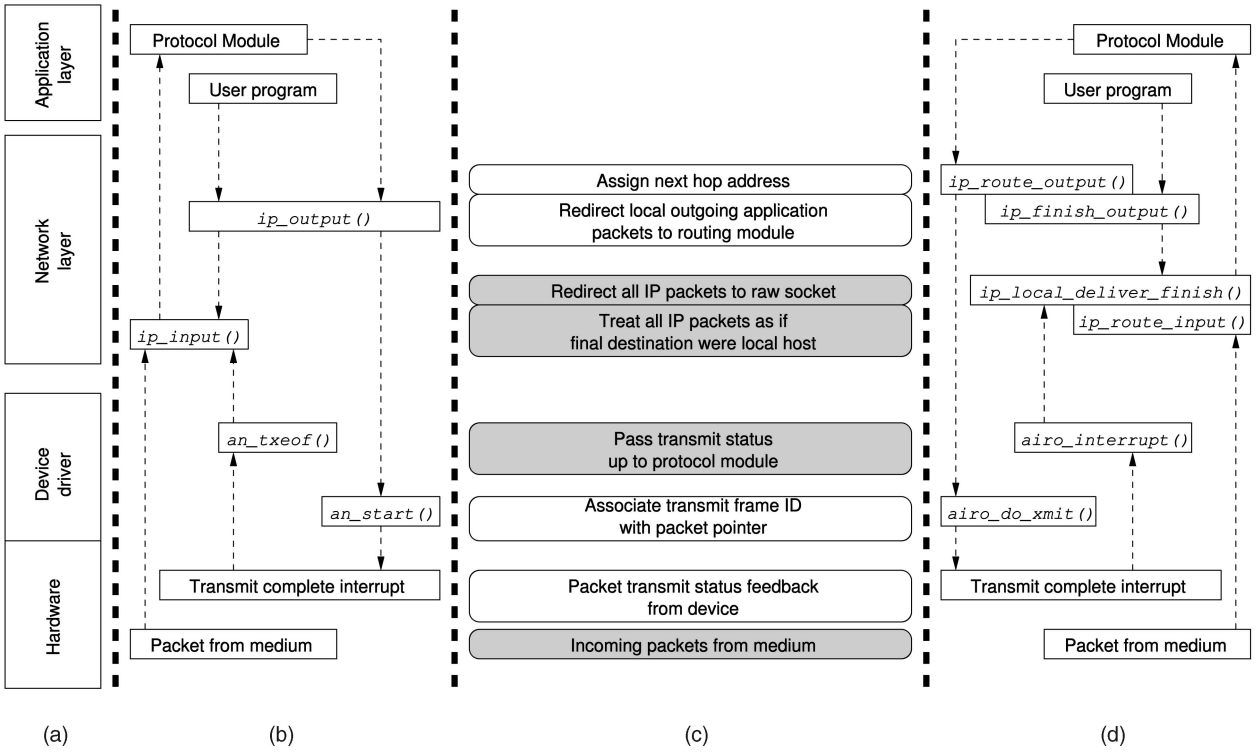


Fig. 2. Kernel modifications in FreeBSD and Linux to support PRAN. (a) OSI model. (b) FreeBSD kernel implementation. (c) Kernel modifications. (d) Linux kernel implementation.

In the FreeBSD kernel network protocol stack implementation, incoming IP packets from the wireless medium are processed in the *ip_input* function. Modifications were done in this function to pass all IP packets, regardless of their destinations, through a raw IP socket to the user-level protocol module. For outgoing traffic, after the protocol module sends each processed packet to the kernel with next-hop information, modifications were made in the outgoing IP function, *ip_output*, to utilize the next-hop information passed from the user-level routing protocol. For outgoing packets that originate from a local application, the packets are intercepted at *ip_output* and redirected through *ip_input* to the raw socket, where they are passed up to PRAN's 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 associated the Ethernet frame identifier (ID) with the (opaque) packet pointer from *ns-2* (Section 4.2.5). 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 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, modifications were made at similar interfaces with slightly different function calls. Incoming IP packet processing logic in Linux is divided into multiple functions. Specifically, modifications to treat 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) were made in the *ip_route_input* function, where the kernel decides whether to pass an incoming packet to a higher-layer protocol. The packet is then redirected to the raw IP socket interface in function *ip_local_deliver_finish*, where the kernel determines to which higher-layer protocol the packet should be delivered. Similarly, outgoing IP processing logic in Linux is also divided into multiple functions. Kernel modifications to assign a next-hop address for an outgoing IP packet were made in the *ip_route_output* function, where the routing entry is constructed. For an outgoing packet that originates from a local application, the packet is intercepted and passed up to the user-level protocol module for a routing decision at the end of the outgoing IP processing logic in the *ip_finish_output* function. Modifications in the Cisco 350 wireless LAN device driver in Linux were made in the same logical place as in FreeBSD. Here, the *airo_interrupt* and *airo_do_xmit* functions correspond to functions *an_txeof* and *an_start*, respectively, in FreeBSD.

As shown in Fig. 2, the kernel modifications to support PRAN, as described in Section 4.1, are located at similar interfaces in FreeBSD and in Linux; they are located at well-defined locations in the network protocol stack layering (i.e., IP input/output, Ethernet input/output, device driver input/output, and routing gateway assignment).

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 source code. In fact,

TABLE 1
Packet Processing Times in Our PRAN Implementation (1,400 Bytes/Packet)

Configuration	Kernel Processing Time for Incoming Packets (μs)	User Level Processing Time (μs)	Kernel Processing Time for Outgoing Packets (μs)	Total Processing Time (μs)
AODV on Linux	16.67	18.41	6.11	41.19
DSR on Linux	17.66	27.68	7.76	53.10
AODV on FreeBSD	248.10	77.91	13.32	339.33
DSR on FreeBSD	229.64	118.72	17.28	365.64

the FreeBSD and Linux kernel networking codes evolved from entirely different code bases. The FreeBSD networking code evolved from the original TCP/IP implementation created for BSD Unix in the early 1980s at the University of California at Berkeley; FreeBSD started with the 4.3BSD-Lite ("Net/2") distribution from Berkeley and later converted to the 4.4BSD-Lite code base. The Linux networking stack, on the other hand, was intentionally separated from BSD code due to legal issues with the BSD code at the time. The Linux networking stack was originally developed, led by Ross Biro, in 1992 [32]. The Linux networking stack, however, does share similarities with FreeBSD in that both operating systems are POSIX compliant. In general, the examples of FreeBSD and Linux implementations described above indicate that the kernel modifications to support PRAN should be standard across all systems that have normal protocol layering, not just for POSIX-compliant systems.

6 Performance Evaluation

In this section, we present quantitative measurements to show that the PRAN architecture, with its user-level protocol implementation, does not present a network bottleneck. We also describe a PRAN demonstration network to show that the architecture implementation can support demanding applications with realistic traffic loads.

6.1 System Processing Overhead

In order to measure the overhead incurred in the PRAN implementation, we set up a static network with IBM ThinkPad X31 laptops, each with a 1.4 GHz Pentium M processor and 256 MB of RAM, using Cisco Aironet 350 IEEE 802.11 wireless LAN cards. Table 1 shows the processing time that is incurred in forwarding a single 1,400-byte data packet at an intermediate node; we show the times separately for our AODV and DSR implementations under PRAN on Linux and on FreeBSD. The presented results are an average of four runs and the variation among runs was insignificant. The version of Linux used was Red Hat 9 with kernel version 2.4.20, and the version of FreeBSD used was 5.1-RELEASE. We show only results for data forwarding since most transmissions are for data forwarding. These times were measured in the kernel in terms of CPU counters using the Intel benchmarking instruction *rdtsc()*. At the user level, the same machine instruction is called using an assembly language instruction.

The kernel processing time for an incoming packet in Table 1 is the time between when the kernel is about to determine routing information for the packet and when the packet is received by the user level protocol module (Section 5.2). The user-level processing time for a packet is the time between when the packet is received at the user level and when the packet is sent back to the kernel. The kernel processing time for an outgoing packet is the time between when the user-level protocol module passes the packet to the kernel and when the kernel's IP function for processing outgoing packets receives this packet. Table 1 shows a large difference between the processing times in FreeBSD and in Linux; from our measurements, these differences appear to be due mainly to the differences in the time required by each operating system for getting in and out of the kernel, differences in the existing network stack implementations in each operating system, and differences in the implementation of common library functions such as *malloc()*. However, in all cases, the packet processing times are relatively small.

To further quantify the overhead of routing protocol implementations under PRAN, we compared the performance of our DSR implementation on PRAN to a theoretically optimal native DSR implementation done entirely inside the kernel. For the theoretically optimal DSR implementation, we made the packet sizes the same as they would be in DSR (including the DSR source routing header), but we used the native kernel IP packet forwarding to process each packet (thus adding zero CPU processing time due to DSR for each packet). These measurements were done using the same IBM ThinkPad X31 laptops and Cisco Aironet 350 IEEE 802.11 wireless LAN cards, running the 5.1-RELEASE version of FreeBSD. We chose the DSR implementation on FreeBSD, since that combination provides the lowest supported data bandwidth, as shown in Table 1. We were unable to directly compare the performance of our PRAN DSR implementation against a native in-kernel implementation of DSR since we did not have access to one that utilized the same hardware and operating system configuration (our earlier in-kernel implementation of DSR was done on the much older 3.3-RELEASE version of FreeBSD and used Lucent WaveLAN-II wireless LAN cards [19], [7]).

Our goal in these measurements was to determine the maximum packet rates (and, thus, maximum bandwidth) that each implementation could support with one source node originating UDP packets at a constant rate. We carried out this experiment for two scenarios: with two nodes using a 1-hop path and with three nodes using a 2-hop path. The results, averaged over four runs, are shown in Fig. 3 (the variation between individual runs was insignificant). The

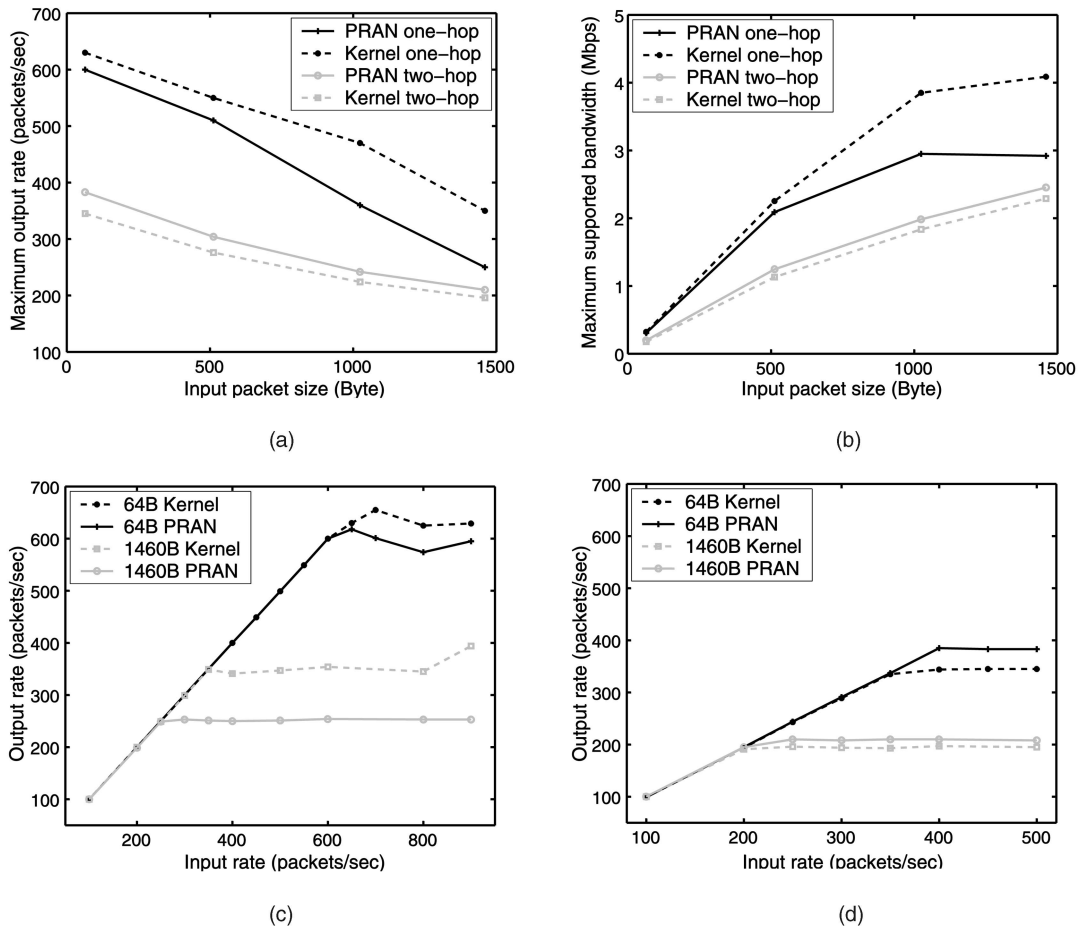


Fig. 3. Comparison of supported packet rate and bandwidth in PRAN DSR and optimal FreeBSD kernel DSR implementation. (a) Maximum supported packet rate. (b) Maximum supported bandwidth. (c) Performance scaling of PRAN for one-hop. (d) Performance scaling of PRAN for two-hop.

output packet rates shown were measured at the receiving node, counting the packets actually received from the source node.

When forwarding packets over the 2-hop route, the maximum achievable output packet rate (and, thus, the maximum achievable bandwidth) is about half that over the 1-hop route (Figs. 3a and 3b). This is expected, since with the 2-hop route, the intermediate (middle) node must receive and transmit each packet over the same wireless medium, consuming twice the wireless channel capacity as with the 1-hop route in which there is no intermediate node. With increasing packet size, the supported packet rate goes down considerably, in both the theoretically optimal DSR kernel implementation and in the PRAN DSR implementation. The difference is due to the fact that PRAN copies the entire packet (including the data payload) into user space, which requires allocating and deallocating memory as well as copying the data. Interestingly, with the 2-hop route, the PRAN implementation achieves a slightly higher maximum output packet rate (and correspondingly higher maximum supported bandwidth) than does our theoretically optimal DSR kernel implementation. We believe this anomaly is due to the extra levels of queuing present in the PRAN implementation, allowing more packets to be queued before the queues overflow and packets must be dropped; this extra queuing capacity helps to avoid dropping packets in cases in which the transmission of packets is temporarily delayed, such as due to wireless contention or collision and link-layer retranmission.

As the rate of sending packets increases (Figs. 3c and 3d), the PRAN implementation and the kernel implementation are able to achieve the same output packet rates until reaching a point at which the wireless channel and network hardware saturate; beyond this rate, the output packet rate remains roughly constant.

One source of overhead present due to the PRAN architecture is the need to convert packets between the native (IP and DSR) format and the abstract formats used within the *ns-2* code. However, much of the same conversion is effectively spread throughout the code for any protocol implementation using native packet formats, since accessing fields in various packet headers during processing requires extracting the value from that field. For example, Fig. 4 shows the assembly language code generated by the standard C-language compiler for incrementing an embedded 3-bit integer field within a sample packet header; since PRAN uses abstract packet formats within the protocol processing, incrementing this value is simple, as each field has already been extracted by the packet format converter. In contrast, for incrementing the

PRAN:	Native:

`incl 8(%eax)`	`movl %eax, %ecx`
	`movb 8(%eax), %al`
	`sall $5, %eax`
	`sarb $5, %al`
	`movsbl %al, %eax`
	`leal 1(%eax), %edx`
	`movb 8(%ecx), %al`
	`andl $7, %edx`
	`andl $-8, %eax`
	`orl %edx, %eax`
	`movb %al, 8(%ecx)`

Fig. 4. Comparison of assembly language code generated by the standard C-language compiler for incrementing a 3-bit field in an example packet format in PRAN (*ns-2*) and native packet formats (the `%eax` register points to the packet in memory).

same field using native packet formats, the generated code must extract the field, increment it, and reinsert it into the correct word of the header. Whereas PRAN converts each field once when converting to abstract formats and when converting to native formats, a protocol implementation using the native packet formats directly may effectively convert some fields multiple times if they are accessed multiple times in processing the packet (resulting in higher overhead for the native implementation) or may not convert some fields at all if they are not used in processing a given packet (resulting in higher overhead for the PRAN implementation).

6.2 PRAN Demonstration

In order to further validate the usability of PRAN and to demonstrate the resulting implementation of an ad hoc network routing protocol, we constructed a test network of mobile and stationary nodes in our department building at Rice University. 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 [20]. 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. As with the results shown earlier in Fig. 3, we chose the DSR implementation on FreeBSD to show the usability of PRAN since that combination provides the lowest supported data bandwidth (Table 1).

Fig. 5a summarizes the configuration of this test ad hoc network. In this section, we describe the design and operation of the different components of this network and we present the details of the demonstration.

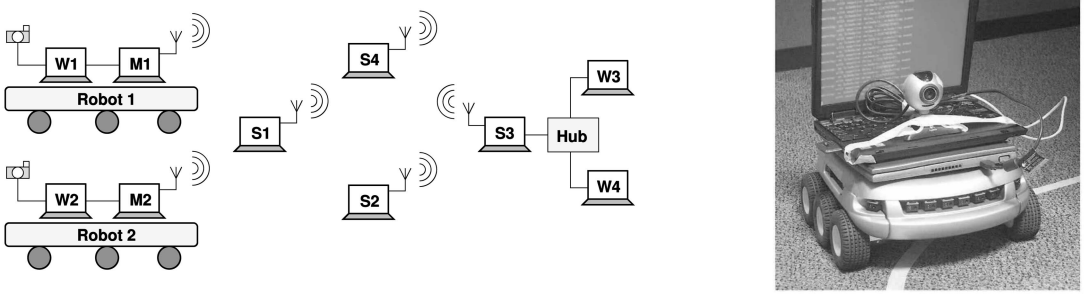
6.2.1 Wireless Nodes

Our test network included six wireless nodes implemented as laptops running FreeBSD 5.1-RELEASE, modified as described in Section 4.1. Each wireless node was an IBM ThinkPad model X31 laptop identical to those described in Section 6.1.

Of the six laptops, four were stationary, shown as **S1** through **S4** in Fig. 5a, and two were mobile, shown as **M1** and **M2**. Nodes **W1** to **W4** are Windows machines running Microsoft Windows NetMeeting. By moving the mobile nodes **M1** and **M2**, changing multihop routes were created through a varying sequence of the stationary wireless nodes and through the other mobile node. Each of these laptops used a Cisco Aironet 350 IEEE 802.11 wireless LAN card as the wireless interface, operating at 11 Mbps; 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. The stationary wireless nodes as well as the mobile ones used the same wireless configuration.

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, substantially reducing each node's maximum transmission distance (reducing the transmission power level by a factor of 2 generally reduces the maximum transmission distance by at least a factor of 4) [27]. With this reduced transmit power level, our network created multihop routes of up to five hops in length. We validated during our demonstration that the network traffic was using multiple hops for substantial parts of the demonstration period.

Each mobile node (laptop) in our network was physically carried on a mobile robot, which we could control by software commands over the ad hoc network. We used the



(a) (b)

Fig. 5. PRAN demonstration configuration. (a) Network configuration. (b) Mobile node configuration.

Koala robot [10], manufactured by K-Team S.A. of Switzerland. The robot is approximately 30 cm (12 inches) square and 20 cm (8 inches) in height. Each robot carried two laptops, one running Windows NetMeeting on Microsoft Windows XP Professional for traffic generation (nodes **W1** and **W2**) and one running FreeBSD as the gateway to the ad hoc network (nodes **M1** and **M2**). Fig. 5b depicts the configuration of each robot mobile node; the top laptop is running FreeBSD and DSR and the bottom laptop is running Microsoft Windows XP and NetMeeting.

6.2.2 Data Traffic Generation

In order to generate some sample network traffic for evaluating our implementation, and also to help with controlling the motion of the robots as mobile nodes in our network, we decided to send live real-time video from each robot over the ad hoc network to a centralized control location. By watching the received video from a robot at that control location, it would be possible to remotely "drive" the robot by sending real-time 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. Since we had experience with using Microsoft NetMeeting video in earlier experiments with DSR [7], we chose NetMeeting as the video application. By using Windows NetMeeting for the video, we also were able to demonstrate the compatibility of our implementation with standard, unmodified IP-based applications, as we do not have the source code for either Windows or NetMeeting.

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

6.3 PRAN Demonstration Evaluation

The use of video and remote control of the robots created an engaging demonstration of PRAN's capabilities. In particular, in driving a robot, the user watches the video display closely to avoid driving the robot into a wall; this is particularly true in turning a corner with a robot. 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 PRAN—worked very well.

The video and the robot response to remote commands at most times worked smoothly, although they were occasionally interrupted briefly, such as when a packet was dropped due to a change in the network route used or due to wireless interference from other communication. We had no control over the video encoding or transport protocol used by Microsoft NetMeeting (video over wireless was not a part of our research). Since NetMeeting transmits a full video frame only every 15 seconds, with deltas transmitted between these full frames [21], a single dropped packet can have a significant effect on video quality. However, since the Packet Delivery Ratio (discussed below) remained very high in our experiments, the video quality remained sufficient for navigating and operating the mobile robots in real-time. Overall, the video quality was similar to the quality achieved in our earlier experiments using Microsoft NetMeeting video with our native implementation of DSR that was done entirely inside the FreeBSD 3.3 kernel [7].

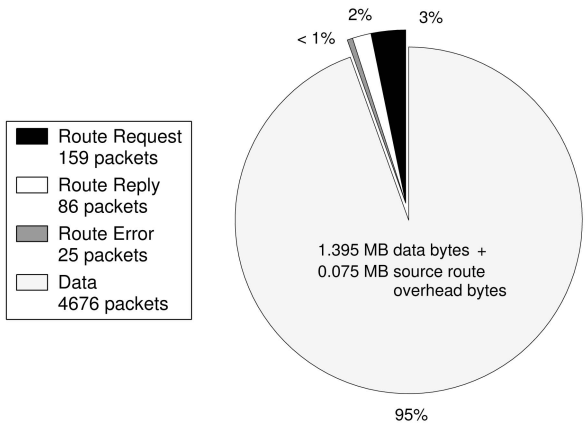


Fig. 6. Packet type and overhead distribution in PRAN DSR demonstration.

To further understand the performance of the PRAN system, we collected measurements during one run of our demonstration network. 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).

Fig. 6 summarizes 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.

Most of the overhead packets, among ROUTE REQUEST, ROUTE REPLY, and ROUTE ERROR packets, were ROUTE REQUESTs, 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, if multiple paths to the target node exist or if multiple other nodes reply with a route to this target from their DSR Route Caches.

Fig. 7a shows the Packet Delivery Ratio (PDR) for this 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 averaged over 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

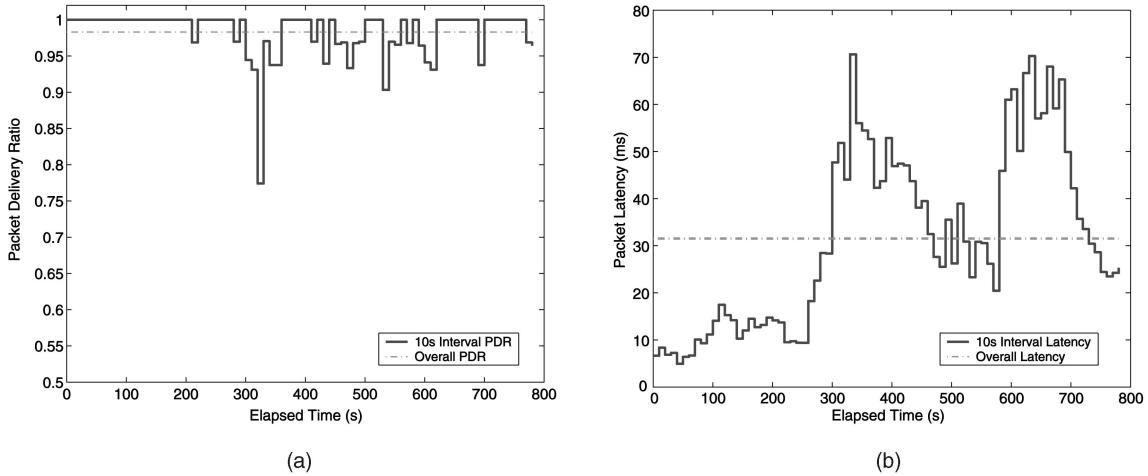


Fig. 7. Performance of DSR in the PRAN demonstration. (a) Packet Delivery Ratio (the y-axis ranges from 0.5 to 1). (b) Packet Delivery Latency.

mobile robot was the farthest from the rest of the network and thus was experiencing temporary wireless signal fading. This behavior occurs in real networks but is not modeled accurately in most purely simulation-based evaluations of ad hoc network protocols, since it depends on more realistic physical layer radio modeling than is usually done.

Finally, Fig. 7b shows the Packet Delivery Latency for packets during this demonstration run. The 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 averaged over 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 (more hops) for packets to travel from the robot to the destination node. Additionally, the weak signal strength at this location can cause additional link-layer RTS/CTS or data packet retransmissions due to dropped packets at the IEEE 802.11 MAC level, adding to Packet Delivery Latency.

7 Discussion

7.1 An Alternative Approach

A simpler approach to PRAN would be to package each simulation packet (in its abstract format) as data inside an IP packet without being converted to native format. The resulting IP packet could then be sent over the real network to the next-hop node and the simulation packet then injected into the simulation environment running there; the simulation packet would be immediately usable without format conversion. Although this approach would no longer require the conversion module and associated overhead (Section 4.2.4), next-hop information would still need to be passed down to the kernel and processed as specified in Section 4.1 so that the correct next hop would be used for forwarding the packet. By removing the conversion module, new protocol implementations could be quickly created, since there would be no protocol-specific implementation in either user or kernel space, aside from a small effort to retrieve next-hop information from the simulation module for use by the kernel.

There are, however, many problems with this approach, making it inappropriate for most implementation purposes. By not converting the simulation packets into native packet format, this approach prevents interoperability with other implementations of the same protocol. More importantly, this approach can significantly affect protocol behavior since the sizes of the resulting packets may be substantially different (and larger) than the corresponding native packets should be. 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 as compact as native packet formats. The effects of incorrect packet sizes on protocol behavior are especially important for wireless network protocols, where the capacity of the wireless medium is limited.

7.2 Applicability of the PRAN Architecture

Since the physical behavior of wireless signals such as signal fading, multipath, and collisions are very difficult to model accurately in simulation, PRAN is the most beneficial for testing wireless network protocols. By employing real wireless transmissions and mobility in a real network, PRAN can uncover issues related to the dynamic wireless medium that cannot be accurately modeled in simulations. For example, when the wireless signal is weak, some routing packets (e.g., sent as wireless broadcasts) may be successfully received, but most data packets will likely be dropped; use of such a route can substantially affect performance, but the subtlety of which packets are received and which are not is difficult to model accurately in simulation.

For this particular implementation of the PRAN architecture, we focused on supporting routing protocols and implemented kernel modifications for packet interceptions at the network layer (IP layer). Thus, protocols at other layers (e.g., transport protocols) cannot be supported with our current implementation of PRAN, although a similar approach could be used. Specifically, to support protocol modules at any layer, modifications parallel to that described in Section 4.1 can be made at the appropriate layer in the kernel networking stack. However, protocols that require tight constraints on real-time processing such as IEEE 802.11 (e.g., returning a CTS after receiving an RTS) may not be able to be directly supported due to the variable latency of entering the PRAN user-level process and returning to the kernel.

In addition, some protocol simulation models have made use of global knowledge available within the simulator to simplify the implementation of the simulation, and such simulation models cannot be used within PRAN without modifications to the simulation code. For example, some simulations of routing protocols for ad hoc networks have used global knowledge within the simulator such that nodes always immediately know their current set of neighbors without using any actual neighbor discovery mechanism; as another example, some simulations of geographical routing protocols have used global knowledge within the simulator such that the sender of a packet always knows the correct current geographical coordinates of the intended receiver node without using any actual location lookup or updating mechanism.

To be used with PRAN, such simulation models would require the addition of protocol mechanisms to replace this use of global knowledge, but likewise, any other technique for implementing a real physical implementation of such a protocol would require these additional protocol mechanisms. Furthermore, simulations that use such global knowledge may produce inaccurate results even in simulation, since they do not accurately model the overhead of such mechanisms nor the performance effects when such mechanisms may not be able to provide information that is as complete or up-to-date as is available through the global knowledge available within the simulator.

8 Conclusion

The common method of evaluation for ad hoc network protocols is currently network simulation. For example, simulation allows repeatable behavior in experiments, avoids the need to deploy and operate large numbers of physical mobile nodes, and allows experiments to be performed with more nodes than may be available within a limited hardware budget. However, it is difficult in simulation to accurately model the complex behaviors of real wireless communications, real node mobility, and real application workloads. Physical implementation of the protocol allows the real protocol to be tested in a real environment, avoiding these concerns, but due to implementation and experimental complexity, many protocol designs are evaluated only in simulation.

Furthermore, when both simulation and physical implementation are used, they have typically been done in a way that is orthogonal to each other, requiring completely separate implementations of the protocol, one for the simulation model and one for the physical protocol implementation. Our PRAN (Physical Realization of Ad hoc Networks) architecture allows the protocol code to be written just once and used in the simulation environment as well as in the physical implementation. In particular, PRAN allows existing, *unmodified* simulation models of ad hoc network routing protocols to be used to create such physical implementations. In addition to saving implementation effort for the physical implementation, reusing the existing simulation code avoids introducing new bugs in the implementation and eases later maintenance of the code. For example, new protocol features and options can be tested and evaluated first in simulation, and then moved without modification into the physical environment.

In this paper, we have described the PRAN architecture and the use of PRAN in creating example implementations of the DSR and AODV ad hoc network routing protocols on FreeBSD and Linux from the existing *ns-2* models of these two protocols. 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 PRAN is *identical* for both protocols. On Linux, the total packet processing times for a 1,400-byte data packet were 41 μs for the AODV and 53 μs for the DSR implementation under PRAN; on FreeBSD, the processing times were 339 μs for AODV and 365 μs for DSR. In comparing our implemenation of DSR under PRAN to a theoretically optimal native implemenation of DSR done entirely inside the kernel, we found the PRAN-based DSR implementation to add little extra overhead, supporting almost the same maximum packet rates and bandwidth as the theoretically optimal native kernel-based DSR implementation.

We have also shown in this paper the ability of the resulting protocol implementations to handle real, demanding applications by describing a demonstration of our PRAN DSR implementation transmitting real-time video over a multihop mobile ad hoc network; the demonstration featured mobile robots being remotely operated based on the transmitted live video stream. All video and robot control messages were transmitted over the ad hoc network running our PRAN DSR implementation. Averaged over the demonstration, the data packet delivery ratio was 98 percent and data packet delivery latency was 30 milliseconds.

We plan to publicly release the source code for our PRAN system to allow other ad hoc network researchers to easily experiment with a variety of protocols in real physical implementations.

Acknowledgments

An earlier version of this work was presented at the ACM SIGCOMM Asia Workshop 2005 [30]. This work was supported in part by NASA under grant NAG3-2534, by the US National Science Foundation (NSF) under grants CNS-0520280, CNS-0435425, CNS-0338856, CNS-0325971, and CNS-0209204, and by a gift from Schlumberger. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily

representing the official policies or endorsements, either express or implied, of NASA, NSF, Schlumberger, Rice University, or the US Government or any of its agencies.

REFERENCES

[1] J. Allard, P. Gonin, M. Singh, and G.G. Richard III, "A User Level Framework for Ad Hoc Routing," *Proc. 27th Ann. IEEE Conf. Local Computer Networks (LCN '02),* pp. 13-19, Nov. 2002.

[2] B. Bellur and R.G. Ogier, "A Reliable, Efficient Topology Broadcast Protocol for Dynamic Networks," *Proc. INFOCOM '99,* pp. 178-186, Mar. 1999.

[3] K. Fall, "Network Emulation in the Vint/NS Simulator," *Proc. Fourth IEEE Symp. Computers and Comm. (ISCC '99),* pp. 244-250, July 1999.

[4] "The *ns* Manual (Formerly *ns* Notes and Documentation)," K. Fall and K. Varadhan, eds., The VINT Project, Univ. of Califonia, Berkeley, LBL, Univ. of Southern California/ISI, and Xerox PARC, Nov. 2003, http://www.isi.edu/nsnam/ns/doc/.

[5] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks," *Proc. USENIX Technical Conf.,* pp. 283-296, June 2004.

[6] T. Goff, N.B. Abu-Ghazaleh, D.S. Phatak, and R. Kahvecioglu, "Preemptive Routing in Ad Hoc Networks," *Proc. MobiCom '01,* pp. 43-52, July 2001.

[7] Y.-C. Hu and D.B. Johnson, "Design and Demonstration of a Live Audio and Video over Multihop Wireless Ad Hoc Networks," *Proc. IEEE Military Comm. Conf. (MILCOM '02),* Oct. 2000.

[8] D.B. Johnson and D.A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," *Mobile Computing,* T. Imielinski and H. Korth, eds., chapter 5, pp. 153-181, Kluwer Academic, 1996.

[9] J. Elischer, "The Netgraph Networking System," http://www.gsp.com/cgi-bin/man.cgi?section=4&topic=netgraph, 2007.

[10] K-Team S.A., "Koala Robot," http://www.gsp.com/cgi-bin/man.cgi?section=4&topic=netgraph, 2007.

[11] Q. Ke, D.A. Maltz, and D.B. Johnson, "Emulation of Multi-Hop Wireless Ad Hoc Networks," *Proc. Seventh Int'l Workshop Mobile Multimedia Comm. (MOMUC '00),* Oct. 2000.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek, "The Click Modular Router," *ACM Trans. Computer Systems,* vol. 18, no. 30, pp. 263-297, Aug. 2000.

[13] D. Kotz, C. Newport, R.S. Gray, J. Liu, Y. Yuan, and C. Elliott, "Experimental Evaluation of Wireless Simulation Assumptions," *Proc. Seventh ACM Int'l Symp. Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '04),* pp. 78-82, Oct. 2004.

[14] M. Krasnyansky, "Universal TUN/TAP Driver: Virtual Point-to-Point (TUN) and Ethernet (TAP) Devices," http://vtun.sourceforge.net/tun/, 2007.

[15] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," *Proc. First Int'l Conf. Embedded Networked Sensor Systems (SenSys '03),* pp. 126-137, Nov. 2003.

[16] J. Liu, Y. Yuan, D.M. Nicol, R.S. Gray, C.C. Newport, D. Kotz, and L.F. Perrone, "Simulation Validation Using Direct Execution of Wireless Ad-Hoc Routing Protocols," *Proc. 18th Workshop Parallel and Distributed Simulation (PADS '04),* pp. 7-16, May 2004.

[17] H. Lundgren and E. Nordstrom, "AODV-UU," http://core.it.uu.se/core/index.php/AODV-UU, 2007.

[18] P. Mahadevan, A. Rodriguez, D. Becker, and A. Vahdat, "MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks," *Mobile Computing and Comm. Rev.,* vol. 10, no. 2, pp. 26-37, Apr. 2006.

[19] D.A. Maltz, J. Broch, and D.B. Johnson, "Quantitative Lessons from a Full-Scale Multi-Hop Wireless Ad Hoc Network Testbed," *Proc. IEEE Wireless Comm. and Networking Conf.,* pp. 992-997, Sept. 2000.

[20] Microsoft Corp., "Microsoft NetMeeting," http://www.microsoft.com/windows/netmeeting/, 2007.

[21] Microsoft Corp., "Microsoft Windows NetMeeting Resource Kit," http://www.microsoft.com/windows/NetMeeting/Corp/reskit/, 2007.

[22] M. Neufeld, A. Jain, and D. Grunwald, "Nsclick: Bridging Network Simulation and Deployment," *Proc. Fifth ACM Int'l Workshop Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '02),* pp. 74-81, Sept. 2002.

[23] OPNET Technologies, "OPNET Modeler," http://www.opnet.com/products/modeler/home.html, 2007.

[24] V.D. Park and M.S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," *Proc. INFOCOM '97,* pp. 1405-1413, Apr. 1997.

[25] C.E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," *Proc. SIGCOMM '94 Conf. Comm. Architectures, Protocols and Applications,* pp. 234-244, Aug. 1994.

[26] C.E. Perkins and E.M. Royer, "Ad-Hoc On-Demand Distance Vector Routing," *Proc. Second IEEE Workshop Mobile Computing Systems and Applications,* pp. 90-100, Feb. 1999.

[27] T.S. Rappaport, *Wireless Communications: Principles and Practice.* Prentice Hall, 1996.

[28] Rooftop Communications, "The Rooftop C++ Protocol Toolkit (CPT)," http://web.archive.org/web/19980614083648/www.rooftop.com/rnd.shtml, 2007.

[29] E.M. Royer and C.E. Perkins, "An Implementation Study of the AODV Routing Protocol," *Proc. Second IEEE Wireless Comm. and Networking Conf. (WCNC '00),* Sept. 2000.

[30] A.K. Saha, K.A. To, S. PalChaudhuri, S. Du, and D.B. Johnson, "Physical Implementation and Evaluation of Ad Hoc Network Routing Protocols Using Unmodified Simulation Models," *Proc. ACM SIGCOMM Asia Workshop,* Apr. 2005.

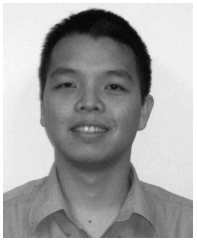
[31] Scalable Network Technologies, "QualNet Family of Products," http://www.scalable-networks.com/products/qualnet.php, 2007.

[32] T. Dawson and A. Rubini, "A Brief History of Linux Networking Kernel Development," http://tldp.org/HOWTO/NET3-4-HOWTO.html, 2007.

[33] G.R. Wright and W.R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation.* Addison-Wesley, 1995.

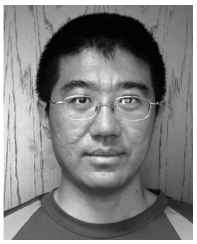
[34] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks," *Proc. 12th Workshop Parallel and Distributed Simulation (PADS '98),* pp. 154-161, May 1998.

Amit Kumar Saha received the BTech degree in computer science and engineering from IIT Kharagpur, India, in 1999 and the MS degree in computer science from Rice University, Houston, Texas, in 2003. He is currently a graduate student in the Department of Computer Science at Rice University and expects to receive the PhD degree in computer science in 2007. His current research is in mobile and wireless systems with a special emphasis on mesh networking.



Khoa Anh To received the BS and MS degrees from Rice University in 1999 and 2005, respectively. His graduate work focused on multihop wireless ad hoc networks. His interests include commercial deployments of wireless ad hoc networks and integrations of wireless network technologies. He is currently working for a network security company in Austin, Texas.

Santashil PalChaudhuri received the BTech degree from IIT Kharagpur in 2000 and the MS and PhD degrees from Rice University in 2002 and 2006, respectively, all in computer science. His PhD thesis focused on a cross-layered architecture for wireless sensor networks. Dr. PalChaudhuri's research interests include mesh, mobile, and wireless networking systems. For the past year, he has been associated with a wireless networking startup company.



Shu Du received the BTech and MTech degrees in computer science and engineering from Tsinghua University, Beijing, China, in 1999 and 2001, respectively. He received the MS degree in computer science from Rice University, Houston, Texas, in 2004. He is currently a graduate student in the Department of Computer Science at Rice University and is expecting to receive the PhD degree in computer science in 2007. His current research is in mobile and wireless systems, especially in routing and MAC protocols of multihop wireless systems.

David B. Johnson is an associate professor of computer science and electrical and computer engineering at Rice University. Prior to joining the faculty at Rice in 2000, he was an associate professor of computer science at Carnegie Mellon University, where he was a member of the faculty for eight years. Professor Johnson is leading the Monarch Project, developing adaptive networking protocols and architectures to allow truly seamless wireless and mobile networking. Related to this research, he has also been very active in the Internet Engineering Task Force (IETF), the principal protocol standards development body for the Internet. He was one of the main designers of the IETF Mobile IP protocol for IPv4 and is the primary designer of Mobile IP for IPv6. His group's Dynamic Source Routing protocol (DSR) for ad hoc networks has been approved by the IETF to be published as an experimental protocol for the Internet. Professor Johnson is currently the chair of SIGMOBILE, the Association for Computing Machinery's Special Interest Group on Mobility of Systems, Users, Data, and Computing. He was the general chair for MobiCom '03 and VANET '06, and was a technical program chair for VANET '05, MobiHoc '02, and MobiCom '97. He has been a member of the technical program committee for more than 30 international conferences and workshops, and has been an editor for several journals. He is a member of the ACM, the IEEE, the IEEE Computer Society, the IEEE Communications Society, USENIX, Sigma Xi, and AAAS.