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

Amit Kumar Saha Khoa To Santashil PalChaudhuri Shu Du David B. Johnson
Rice University, Houston, Texas

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 implementation of ad hoc network routing protocols that merges these two types of evaluation tools. 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 existing, unmodified *ns-2* simulation models. 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 over a multihop mobile ad hoc network; the demonstration features mobile robots being remotely operated based on the video stream transmitted over the network. We also present a detailed performance evaluation of PRAN to show the feasibility of our architecture.

1. Introduction

PRAN (Physical Realization of Ad Hoc Networks)[1] is a new system for 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 mobility. Although, simulation and real physical implementation are each valuable as techniques in evaluating ad hoc network 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, resulting extra effort in coding, debugging, validation, and maintenance.

PRAN allows *existing* simulation models of ad hoc network routing protocols to be used — *without modification* — to create a physical implementation of the same protocol. The 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 node in the ad hoc network. Our system, PRAN, is designed to take advantage of the large base of existing simulation code and 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 PRAN, the protocol code on each node in the network runs in a single user-level process and uses standard interfaces to transmit and receive packets from the kernel, simplifying protocol debugging and making the system highly portable between different host operating systems. Our implementation of PRAN 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 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) [7] and the Ad hoc On-Demand Vector Routing protocol (AODV) [15] on FreeBSD and Linux using the existing, unmodified *ns-2* simulation models. The user-level code is identical between our implementations on FreeBSD and Linux, and the small amount of new operating system kernel support code required by PRAN is identical for both protocols. We also illustrate the ability of the resulting protocol implementations to handle real, demanding applications by presenting a demonstration of our DSR implementation transmitting real-time video over a multihop mobile ad hoc network including mobile robots being remotely operated based on the transmitted video stream. All video and robot control messages were transmitted over the ad hoc network running our DSR implementation.

The rest of this paper is organized as follows. In Section 2, we describe the motivation behind PRAN, and in Section 3,

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

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 concludes the paper.

2. Motivation

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

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

With our PRAN 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 design and development of a new ad hoc network routing protocol progress, the same code can be used directly for simulation and physical implementation evaluation, allowing new features or designs to be tested in the convenience of simulation and in the complex reality of real mobile nodes and real radios.

3. Related Work

Simulation models of many ad hoc network routing protocols have been created in simulators such as *ns-2* [4], GloMoSim [23], OPNET [14], and QualNet [20], 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 merging simulation and physical implementation efforts is to base the new physical implementation of some protocol on the existing code of a simulation model 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 [18]. Like our work, their implementation of the routing protocol 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 a Linux-only kernel interface used by the user-level process for kernel routing table updates; their Linux kernel modifications also interact closely with the kernel routing table data structures, which are different in different operating systems.

Another approach merging simulation and physical implementation is the use of an existing physical implementation of some protocol as the basis for a new simulation model of the same protocol. For example, AODV-UU [12] is a physical implementation of AODV, which can also be executed within *ns-2* as a simulation model of AODV. However, AODV-UU supports only the AODV protocol and does not attempt to be portable to operating systems other than Linux for which it was designed.

More general projects in this area, supporting arbitrary ad hoc network routing protocols rather than a single specific protocol, include the Rooftop C++ Protocol Toolkit (CPT) [17], the *nsclick* simulation environment [13], 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 [10]; 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. creates a new C++ framework for ad hoc network routing protocol implementation and also provides a new, integrated simulator for simple testing protocols implemented in this environment.

TOSSIM [11] provides a high fidelity simulation for TinyOS and mote hardware, such that TinyOS applications can be run in this simulation framework. The basic difference with PRAN is that TOSSIM was designed from scratch with the objective of easy portability of TinyOS applications from the simulation environment to the actual mote hardware, whereas PRAN is more generic and provides for deploying any simulated ad hoc network routing protocol in *ns-2*.

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 non-handling of interrupts.

EmStar [5] is a software environment for developing and deploying wireless sensor network applications on Linux-based hardware platforms like iPAQs. It has a similar goal as PRAN towards providing a useful environment for simulation as well as deployment. However, as with TOSSIM, EmStar was designed from scratch to support easy migration from simulation to implementation, thus making it inapplicable to existing and commonly used simulators.

PRAN also shares some similarity to network emulation systems [3, 9], 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 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 4.2.1.

In contrast to each of these previous projects, PRAN allows the use of *existing, unmodified* protocol simulation models to create new physical protocol implementations. Specifically, we support protocol models from the widely used *ns-2* network simulator, rather than requiring use of new implementation environments, and thus retain all the benefits of *ns-2* simulation, such as rapid prototyping and a widespread user community. Existing protocol modules can easily be used to create new physical implementations, and new protocols or modifications to existing protocols can easily be coded and tested in both simulation and physical implementation. We demonstrate that PRAN can be ported to multiple routing protocols and operating systems. Additionally, the PRAN architecture can be applied to other protocol simulation systems.

4. PRAN System Architecture

The PRAN architecture consists of two parts: 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 process, we created 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 reception 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 packet format. For this module to interface with the real network, we introduce a small amount of kernel support to connect the simulation model 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 Figure 1, for several different packet scenarios. A packet to be forwarded by the node is received at the operating system kernel *device driver* of the network interface hardware, and is then passed to the user level *routing protocol simulation model* via the packet format *converter*; the routing protocol then passes the packet back to the operating system kernel via the converter, and the kernel finally passes the packet to the device driver for transmission. For reception of a packet destined to an application running on this node, the routing protocol simulation model, after processing the packet, passes the packet back to kernel, which transfers it to the user application through the standard IP input function.

4.1. Kernel-Level Support

In order for the network to interact with the user-level protocol module, we used a BSD raw IP socket [22], 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. Raw sockets are also standard BSD sockets implementations, available in many operating systems including UNIX, BSD, Linux, and Microsoft Windows. Thus, the use of raw sockets simplifies operating system portability. We also considered other techniques for passing data between kernel and user space, such as memory mapping and remote procedure calls. However, since data to be passed between kernel and user space resides in the IP stack, the use of a raw IP socket is 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 queueing. In UNIX-like systems, the use of *copyin* and *copyout* requires the size of data to be known, while IP packets vary widely in size. Another option for passing information between user and kernel space was Netgraph [8]. Netgraph provides an efficient method to interface with different network components in the kernel, including user-level process interfaces through socket nodes. However, Netgraph is not a standard feature in most operating systems and is generally only available on FreeBSD.

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 destined for other nodes (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, re-

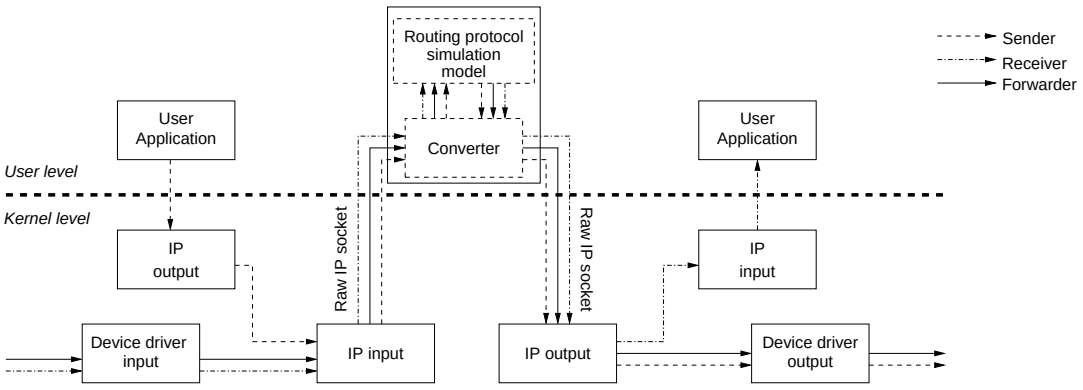


Figure 1: Flow of a Packet through a Node in PRAN in Different Scenarios

gardless 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 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 Figure 1.

In order to determine the address of the next-hop node towards 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, the packet format converter in the user-level routing agent may pass this information to the kernel by appending the IP address of the next-hop to the packet. If next-hop information is present, this value is used by IP output rather than using the existing kernel IP routing table mechanism to determine the next-hop address. The decision to append additional information to the packet instead of passing them separately is closely related to our decision to use raw sockets. When the packet is passed between the kernel and user level through raw socket, if information associated with that packet is separated from the packet, some synchronization mechanism would be required to associate the information to the packet. This would require substantial control overhead as data exchanged through raw sockets and those through memory mappings happen asynchronously.

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 source route header. Other protocols such as AODV could use the kernel's routing table mechanism, but this creates problems for a user-level routing protocol implementation, since the protocol cannot correctly manage the contents of the kernel routing table by keeping track of the last time that each table entry was used [18]. By instead utilizing this new mechanism to allow the user-level routing process to completely manage the routing decisions, these problems can be avoided.

Another area of changes we made was to allow the ad hoc network routing protocol to take advantage of received signal strength information, for example to determine when the currently used route is about to break. Based on this information, the protocol can initiate a search for a new route to the destination while the current route is still active. This optimization, known as preemptive Route Maintenance [6], reduces or even eliminates latency in searching for a new route when the current route breaks. To support this or other uses of receive signal strength for a received packet, we modified the wireless network interface driver to append the received signal strength value to each incoming packet. This information is passed up with the entire packet to the user-level routing process, where the protocol can extract the signal strength information and determine appropriate actions.

Finally, in order to a detect broken link to the next hop, many mobile ad hoc network routing protocols take advantage of link-layer acknowledgment that already exists at the MAC layer. To support this, the device driver is modified to pass the link-layer transmission status to the user-level process. We discuss in Section 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 in which unmodified *ns-2* protocol simulation models could be executed, interfaced to the kernel to process real packets being sent, received, or forwarded by the node on which this simulation model is executed. 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 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 should wait until *real time* (on the node itself) reaches the next scheduled event execution time, and then execute the event. 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, are not changed in any way; the simulation code still maintains its own queue of pending events to be executed, as if it were running in a standard simulation environment.

A similar type of *real-time* event scheduler is also used by network emulation systems [3, 9]. 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. However, in PRAN, each node executes independently; the event queue inside the PRAN user-level protocol process contains only events that should occur at this node itself.

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 and finally transmits the packet using the simulated physical layer. Similarly, when a packet is received at a mobile node from the simulated physical layer, the MAC layer schedules the received packet to the link layer, which then schedules that packet to the routing layer. 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*, we needed to handle the reception of packets from outside the simulation environment. Each node in the physical implementation runs its own copy of the simulation model of the ad hoc network protocol, and packets sent by one node to another are sent over the real network as real packets, rather than being handled internally as a normal *ns-2* event.

To integrate the reception of new packets from outside the simulation environment, we allow the receipt of such a new external packet to terminate the event scheduler's wait for the real time of the next scheduled event execution time. Specifically, the event scheduler loop blocks itself with the operating system until *either* the next scheduled event execution time arrives *or* an external packet arrives at this node that must be handled by the protocol. If a packet arrives before the next scheduled event time, we handle that packet. 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 fact, they are generated by 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). The following pseudocode summarizes the complete event processing loop in PRAN:

```
while 1 do
  nextEarliestEvent = getNextEvent();
  eventTime = nextEarliestEvent.time;
  timeout = eventTime - gettimeofday();
  Wait for (packet from rawSocket or timeout);
  if (timeout expired) then
    Remove nextEarliestEvent;
    Execute nextEarliestEvent;
  else
    Handle received packet;
  end if
end while
```

When the kernel receives a packet that must be handled by the simulated protocol, the kernel uses a raw IP socket to send the packet to the converter which then sends the packet to the user level protocol module. The following section describes the conversion of packets.

4.2.4. Conversion between Packet Formats

Most simulators, including *ns-2*, use an abstract, internal packet format that is different from native packet format, for ease accessing different packet headers and packet header fields in writing the simulation code for a protocol. For the simulator to work transparently in a physical implementation with real packets and with the existing, unmodified protocol simulation model code, an extra software layer must convert

between abstract and native packet formats. On receiving an external packet from the kernel, this converter changes the packet from native format into the simulation abstract packet format; on transmitting a packet outside the node's simulation environment, this converter changes the packet from the simulation abstract packet format into the native packet format.

4.2.5. Transmission of Packets

When the user level protocol module needs to transmit a packet, the packet is received by the converter which then converts the format of the packet from *ns-2* packet format to native format (dependent on the host byte order). The converted packet is then sent to the operating system kernel using a 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, 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 [7]. 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 simulated ad hoc network routing protocol, 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 simulated routing protocol.

In particular, in *ns-2*, a pointer to the internal *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 link-layer feedback), this pointer is still available for the routing layer to use to access the original packet. Replacing the lower layers as present in *ns-2* with the real operating system and hardware does not allow us to directly do this in the same way.

We solve this 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 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 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 simulated environment. 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 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 *ns-2* deletes the *ns-2* packet as normal; if, however, it is a NACK packet, then *ns-2* has a reference to the packet, and the packet can be processed exactly as a failed packet is processed in unmodified *ns-2*.

4.2.6. Application to Other Network Simulators

A number of different discrete event simulators exist and have been used for simulating and evaluating ad hoc network protocols. Among the more frequently used are *ns-2* [4], GloMoSim [23], OPNET [14], and QualNet [20]. 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 makes it possible to replace their abstracted link-layer, MAC, and physical layers with the real operating system and hardware, through our interface to the kernel. If abstract packet formats are used in the simulator, as in *ns-2*, the same type of packet format converter can be used.

5. PRAN Architecture Portability

To demonstrate the simplicity, portability, and effectiveness of PRAN, we present in this section the example implementation 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 *ns-2.26*. However, as mentioned in Section 4.2.6, the PRAN architecture is general purpose and should be able to be applied also 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 protocol that is being implemented. Similarly, the protocol implementation is independent of the underlying operating system and hence is unaware of the operating system

that the machine is running. For example, in our example protocol implementations described here, the user-level code 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

In this section, we demonstrate the simplicity of supporting multiple ad hoc network routing protocols in PRAN. First, we describe our efforts in supporting two popular ad hoc network routing protocols, DSR and AODV. Then we explain how the support for other protocols, simulated in *ns-2*, are similar.

5.1.1. Example DSR Implementation in PRAN

DSR is a source routing protocol. Each packet sent using DSR contains a source route. Here we briefly describe the protocol. The DSR protocol consists of two mechanisms: Route Discovery and Route Maintenance. To perform a Route Discovery for a destination node *D*, a source node *S* broadcasts a ROUTE REQUEST that gets flooded through the network in controlled manner. This request is answered by a ROUTE REPLY from either *D* or some other node that knows a route to *D*. To reduce frequency and propagation of ROUTE REQUESTs each node aggressively caches source routes that the node learns or overhears. Route Maintenance detects when some link over which a data packet is being transmitted breaks. When such a route breakage is detected, a ROUTE ERROR is sent 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 in *ns-2*. We describe below DSR-specific considerations that the DSR conversion module needs to support.

DSR is a source routing protocol with its own IP protocol number and header following the IP header. Thus, all packets transmitted over the DSR network have DSR as the IP protocol number identified in the IP header. To transmit a DATA packet in a DSR network, the converter needs to insert a DSR header between the IP header and transport protocol header. The transport protocol header and its data becomes the data part of the newly created DSR packet. The converter also needs to change the packet's IP protocol field from the original transport protocol to DSR (the original IP protocol value is stored in the DSR header as specified by the DSR specification [7]). Before the DATA packet leaves the DSR network or is delivered to the application at the final destination, it needs to be sent up to the conversion module. The conversion module removes the DSR header and reconstructs the original IP packet with the original transport protocol as its IP protocol. Similarly, DSR control packets also have their own DSR header on top of the IP header. However, since DSR control packets are generated and freed by the DSR routing module, there is no non-DSR packet to modify. The converter only has to convert packets from *ns-2* format to native packet format. After the DSR protocol module processes the packet and constructs the new source route header, next hop information is passed from the protocol module to the converter to be sent to the kernel.

5.1.2. Example AODV Implementation in PRAN

AODV is another widely studied on-demand ad hoc network routing protocol. Here we briefly describe the protocol. When a source *S* needs a route to a destination *D*, *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 route to *D*. In this process each forwarding node creates a *reverse route* back to *S*. Upon reaching a node with a route to *D* the node replies back to *S* with a ROUTE REPLY containing the number of hops that *D* is from itself and the most recent sequence number for *D* known to the replying node. When a node forwards this reply, it creates a *forward route* to *D* by remembering the next-hop node towards *D*.

As with DSR, support for AODV requires only addition of an AODV-specific packet format conversion module. Different from DSR, however, AODV does not require its own protocol header. All AODV control packets are UDP packets with a special UDP port. DATA packets transmitted over the AODV network are the same packets that were created by the application. Thus, the only requirement for the AODV conversion module is to directly convert between *ns-2* and native packet formats. Since DATA packets transmitted on the AODV network are regular packets, no special processing needs to be done before the packets leave the AODV network or is delivered to the application at the final destination. However, for protocol-independent support in the kernel, all packets arriving at a node are passed up to the conversion module before being delivered to the application or to an external network. This 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 protocol simulation code. After the AODV protocol module processes the packet and determines next hop information from its routing table, next hop information is passed from the protocol module to the converter to be sent to the kernel.

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 following interfaces that are a normal part of *ns-2*.

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.

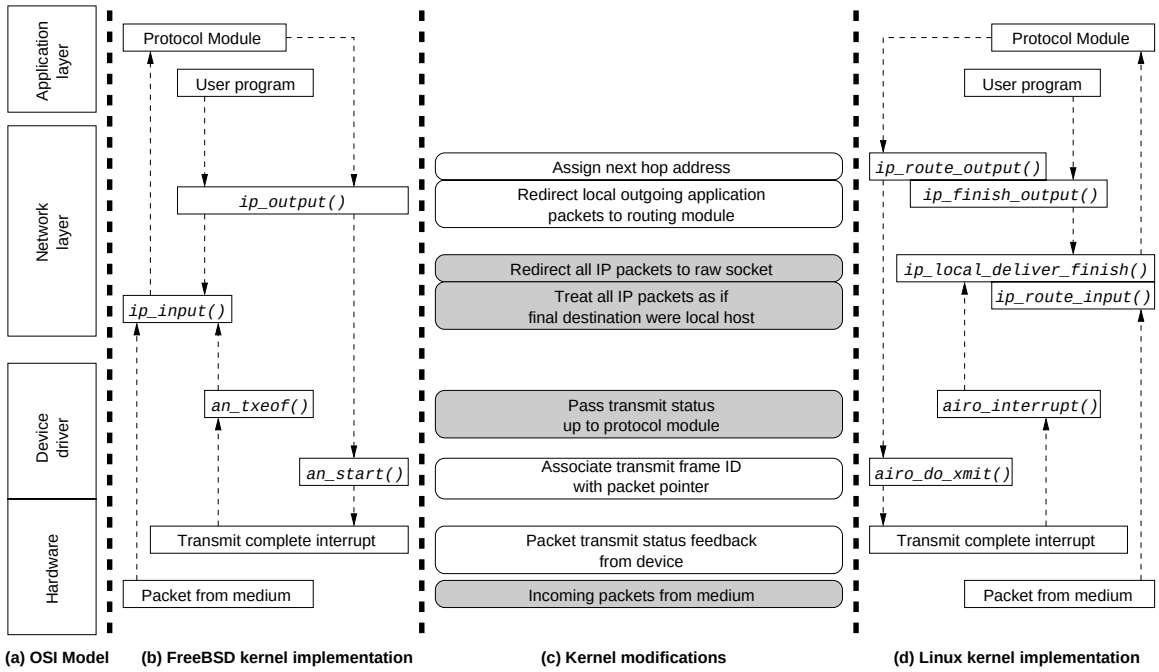


Figure 2: Kernel Modifications in FreeBSD and Linux to Support PRAN

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()*. This callback function is required only for those protocols that respond to link layer transmission failures. If such a callback is provided, then the converter invokes the callback 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 invokes 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. User-level protocol code interfaces with the kernel through 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 implementation between different operating systems requires only changes to this code.

General kernel modifications for the architecture were described in Section 4.1. In Figure 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. Column (a) in Figure 2 identifies the protocol layers present in common network protocol stack implementations, e.g., based on the OSI Reference Model. Columns (b) and (d) show the relevant operation in the FreeBSD and Linux kernels, respectively, for each protocol layer, for incoming and outgoing packets. Between these, column (c) 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 column (c) is aligned with the actual functions in Linux and FreeBSD in columns (b) and (d) where the modification is made.

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 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 are made in the outgoing IP function, *ip_output*, to fill next-hop information with the value 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

Table 1: Packet Processing Times in our PRAN Implementation

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

to be transmitted by *an_start*, we associate the Ethernet frame identifier (ID) with the packet pointer (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 take place 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) happens at 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 socket interface in function *ip_local_deliver_finish*, where the kernel determines which higher-layer protocol the packet should be delivered to. 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 happen 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 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 happen 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.

Figure 2 shows that kernel modifications, as described in Section 4.1, are located at similar interfaces in FreeBSD and in Linux, it also shows that 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 codes. As a matter of fact, the FreeBSD and Linux kernel networking codes evolved from entirely different code bases. FreeBSD networking code evolved from the original Berkeley Extensions. The Linux networking stack, on the other hand, was intentionally separated from BSD code due to copyright issues with the BSD stack at the time. The Linux networking stack was originally developed, lead by Ross Biro, in 1992 [21]. The Linux networking stack, however, does share similarities with FreeBSD in that both operating systems are POSIX compliant. However, the examples of FreeBSD and Linux implementations in this section show that modifications in the kernel are standard across all systems that have normal protocol layering and not just for POSIX-compliant systems.

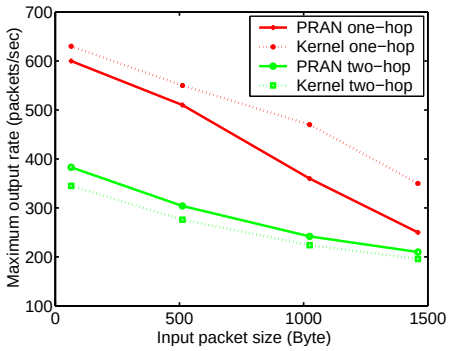
6. Performance Evaluation

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.

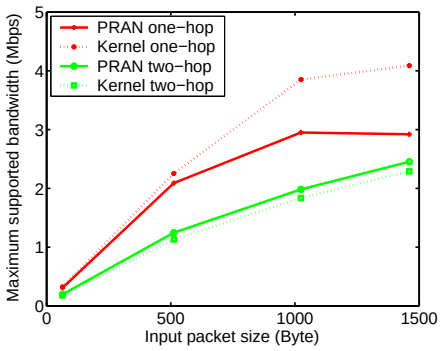
6.1. System Processing Overhead

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. Table 1 shows the processing time that is incurred in forwarding a single 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 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 level instruction.

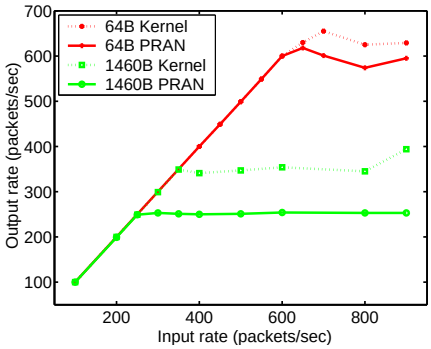
The kernel processing time for an incoming packet in Table 1 is the time between when the kernel is about to determine routing information for the packet and when the packet is received by the user level protocol module (Section 5.2). The user level processing time for a packet is the time between when 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 for Linux; from our measurements, these differences appear to be due



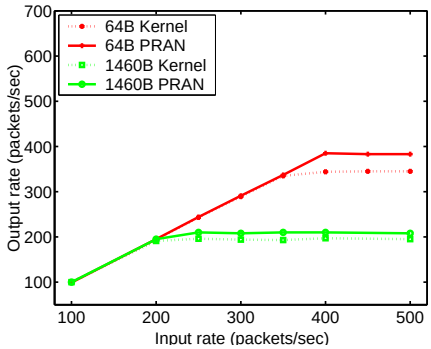
(a) Maximum supported packet rate



(b) Maximum supported bandwidth



(c) Performance scaling of PRAN for one-hop



(d) Performance scaling of PRAN for two-hop

Figure 3: Comparison of supported packet rate and bandwidth in PRAN DSR and optimal FreeBSD kernel DSR implementation

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 implementation 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). Our goal in these measurements was to determine the maximum packet rates (and thus maximum bandwidth) that each DSR 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 are shown in Figure 3. The output packet rates shown were measured at the receiving node, counting the packets actually received from the source node. These measurements were done using the same IBM ThinkPad X31 laptops, running the 5.1-RELEASE version of FreeBSD.

When forwarding packets over the 2-hop route, the maximum achievable output packet rate (and thus the maximum achievable bandwidth) is about half that of the 1-hop scenario (Figures 3(a) and 3(b)). This is expected because unlike in the 1-hop scenario, there is more contention for the same wireless channel in the 2-hop scenario. With increasing packet size, the supported packet rate goes down considerably, in both the optimal DSR kernel implementation kernel and in PRAN. 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.

As the rate of sending packets increases (Figures 3(c) and 3(d)), 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 vari-

ous packet headers during processing requires extracting the value from that field. For example, Figure 4 shows the assembly language code generated by the 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. 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) or may not convert some fields at all (if they are not used in processing a given packet).

6.2. PRAN Demonstration

In order to validate the usability of PRAN, and to demonstrate the resulting implementation of a protocol, we constructed a test network of mobile and stationary nodes in our department building. Our test network consisted of two mobile robots and four stationary ad hoc network nodes, with the robots remotely controlled based on real-time live video from each robot transmitted over the ad hoc network, using standard Microsoft Windows NetMeeting video [2]. All video and robot control messages were transmitted over the ad hoc network with our protocol implementation. We show here the operation of our implementation of DSR on FreeBSD and omit for brevity demonstrations for configurations using AODV or Linux. We chose the DSR implementation on FreeBSD to show the usability of PRAN using the configuration with the lowest supported data bandwidth (Table 1).

Figure 5 summarizes the configuration of this 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 with 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 Figure 5, and two were mobile, shown as **M1** and **M2**. **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) [16]. With this reduced transmit power level, our network created multihop routes of up to 5 hops in length. We validated during our demonstration that the traffic was using multiple hops for substantial parts of the demonstration period.

Each mobile node in our network was implemented as a robot, which we could control by software commands over the ad hoc network. We used the Koala robot [19], 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, and one running FreeBSD as the gateway to the ad hoc network. Figure 6 illustrates the configuration of each robot mobile node.

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

NetMeeting sends all video data packets using UDP. However, when a call is first placed, NetMeeting uses TCP

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)`

Figure 4: Comparison of assembly language code generated for incrementing a field in a sample packet format in PRAN and native packet formats (the `%eax` register points to the packet in memory)

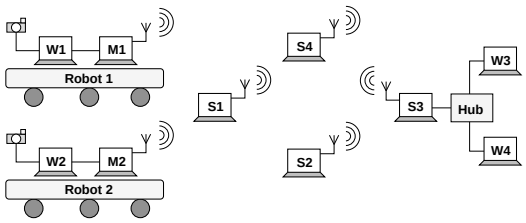


Figure 5: PRAN Demonstration Configuration

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.

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

Figure 7 shows a summary of the types of packets transmitted during the demonstration run and the number of bytes of network overhead caused by each. Network overhead includes all ROUTE REQUEST, ROUTE REPLY, and ROUTE ERROR packets, as well as the DSR source route header in each data packet. In this figure, each transmission of an overhead packet (whether from the originator of the overhead packet or from a forwarding intermediate node) is counted separately.

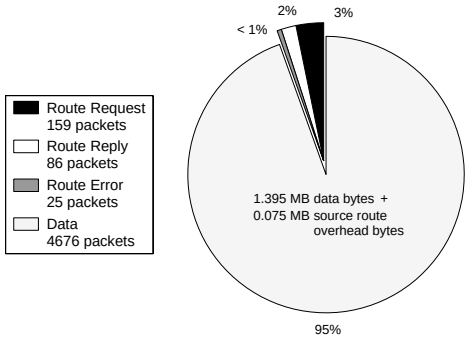


Figure 7: Packet Type and Overhead Distribution

Among ROUTE REQUEST, ROUTE REPLY, and ROUTE ERROR packets, the number of ROUTE REQUESTs is the greatest, since these packets are flooded through the network. The number of ROUTE REPLYs is greater than ROUTE ERRORs, since a Route Discovery is initiated from a single ROUTE ERROR, but this may result in the return of more than one ROUTE REPLY, if multiple paths to the target node exist or if multiple other nodes reply with a route to this target from their Route Cache.

Figure 8 shows the Packet Delivery Ratio (PDR) for the entire run of the demonstration. The PDR is defined as the total fraction of application-level data packets originated that are actually received at the intended destination node. The horizontal dashed line shows the overall PDR for the entire demonstration run, and the solid line shows the PDR separately for each 10-second interval. There is a sharp dip in the PDR at around time 300 seconds, about half way through the demonstration run. At this time, the mobile robot was the farthest from the rest of the network and thus was experiencing temporary wireless signal fading. This behavior occurs in real networks but is not modeled accurately in most

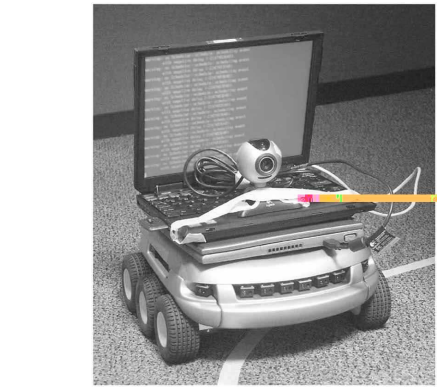


Figure 6: One of the mobile nodes in our PRAN demonstration. The Koala robot is carrying two laptops and a video camera. The top laptop is running FreeBSD and DSR, and the bottom laptop is running Windows XP and NetMeeting.

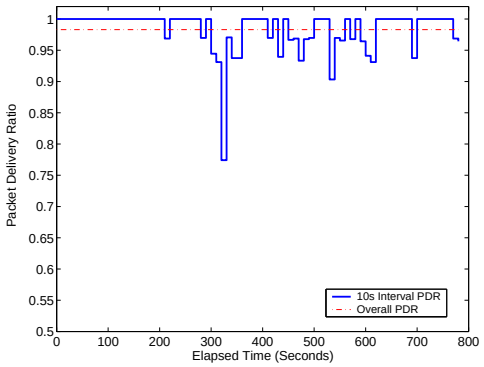


Figure 8: Packet Delivery Ratio (the y-axis ranges from 0.5 to 1)

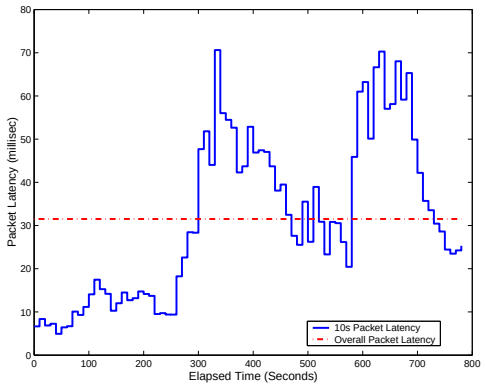


Figure 9: Packet Delivery Latency

purely simulation evaluations of ad hoc network protocols, since it depends on more realistic physical layer radio modeling than is usually done.

Finally, Figure 9 shows the Packet Delivery Latency during this demonstration run. Packet Delivery Latency is measured only for application packets, and is defined as the time between originating a packet at the source node and receiving it at the destination node. The horizontal dashed line shows the overall Packet Delivery latency for the entire demonstration run, and the solid line shows it separately for each 10-second interval. As with the PDR, the Packet Delivery Latency is worse (increases) at around time 300 seconds when the robot was farthest away. This created longer routes for packets to travel from the robot to the destination node. Additionally, the weak signal strength at this location can cause additional RTS/CTS retransmissions due to dropped packets at the IEEE 802.11 MAC level, adding to Packet Delivery Latency.

7. Discussion

7.1. An Alternative Approach

A simpler approach to PRAN would be to package the entire simulation packet (in its abstract format) as data inside an IP packet without being converted to native format. This approach would no longer require the conversion module (Section 4.2.4). Next-hop information would still be passed down to the kernel and processed as specified in Section 4.1. When this IP packet reaches the next-hop node and arrives at the user process, the simulation packet would be immediately available for the simulation protocol module to process. By removing the conversion module, new simulation protocol module could be quickly be implemented, 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 to the kernel.

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

7.2. Benefits of the PRAN Architecture

With PRAN, experimental changes to the protocol can be quickly implemented at the user level. Additionally, a single change can be made in the simulation module that can be tested in both the simulator and validated in the real physical environment.

By sharing code with simulation modules, the architecture retains many useful debugging and logging features that are common in simulation code.

7.3. Applicability of the PRAN Architecture

PRAN provides an effective way to validate experimental protocols on real physical networks. This also means that protocol modules using this architecture cannot assume any global knowledge (e.g. best routes, locations of all mobile nodes, etc.) that are often available in simulation environments. While a number of simulation protocols do require some global knowledge from the simulator, and thus cannot be used in PRAN without additional modifications, any deployment of such protocols to a distributed network environment would require modifications to detach from such knowledge, or to obtain them out-of-band. These deployable versions can then be used unmodified with PRAN.

Since physical behaviors of wireless signals such as signal fading, multipaths, and delays are very difficult to model in simulations, PRAN is most beneficial for testing wireless network protocols. By employing real wireless transmissions and mobility, PRAN can uncover issues related to the dynamic wireless medium that cannot be accurately modelled in simulations. For example, when the wireless signal is weak, some routing packets 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 this specific 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.

8. Conclusion

The common method of evaluation for ad hoc network protocol is network simulation, allowing repeatable behavior and stressing the protocol. On the other hand, physical implementation allows the real protocol to be tested. Typically these two methods are orthogonal to each other, requiring completely separate implementations. Our PRAN architecture allows the protocol code to be written just once, and used in the simulation environment as well as in the physical implementation. Furthermore, existing, *unmodified* simulation models of ad hoc network protocols can 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. New protocol features and options can also be tested and evaluated first in simulation, and then moved without modification into the physical environment.

In this paper, we have described the PRAN architecture and created example implementations of DSR and AODV on FreeBSD and Linux from their existing *ns-2* models. The user-level code is *identical* between our implementations on FreeBSD and Linux, and the small amount of new operating system kernel support code required by PRAN is *identical* for both protocols. We have also shown 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; the demonstration featured mobile robots being remotely operated based on the transmitted video stream. All video and robot control messages were transmitted over the ad hoc network running our DSR implementation. We have also presented a detailed performance evaluation of PRAN to show the feasibility of our architecture.

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.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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