# AC 2010-1884: A HANDS-ON APPROACH TO TEACHING WIRELESS AD HOC NETWORKS

**Sarvesh Kulkarni, Villanova University**

Sarvesh Kulkarni received a B.E. in Computer Engineering from the University of Bombay in 1994, the M.S. and the Ph.D. degrees in Computer Science from the University of Texas at Dallas in 1998 and 2002 respectively. Prior to 2002, he has worked in various industry positions in India and the US. He joined the ECE department at Villanova University in 2002, and is currently an Associate Professor of Computer Engineering. His research interests are: routing algorithms for wireless and wired networks, load-balanced adaptive routing techniques for wireless ad hoc networks, performance analysis and optimization of network parameters, rapid prototyping of autonomous robots, and network architectures for healthcare.

**Joseph Chop, Villanova University**

Joseph Chop received a B.S. in Computer Engineering in 2009, and is currently pursuing an M.S. in Computer Engineering, all from Villanova University. He is currently a research assistant, working in the field of multi-path routing algorithms for mobile ad hoc networks. He has previously worked for Lockheed Martin Information Systems & Global Services, King of Prussia, PA as a software and system engineer designing ship control systems. Prior to this, he has also worked for the Guam Power Authority, Harmon, GU performing network design and maintenance. His other interests include architectural and software design of embedded systems.



© American Society for Engineering Education, 2010

# A Hands-On Approach to Teaching Wireless Ad Hoc Networks

## Abstract

Advanced graduate level courses in mobile wireless networking teach the emerging field of multi-hop, wireless ad hoc networks from a purely research perspective. Simulation software is usually used to predict the performance of wireless ad hoc routing algorithms. However, due to the overly simplistic nature of the simulator's internal channel models and the level of abstraction involved, students do not fully comprehend the impact of design decisions on real world algorithm performance. In addition, the steep learning curve associated with simulators also presents an impediment to timely implementation of the algorithm in the simulator.

This paper proposes a hands-on approach to an upper level graduate course in wireless networking. Students are not only afforded a chance to design *their own* wireless ad hoc routing algorithm, but they also get to implement and test their work on an affordable mobile hardware platform. Portable mobile platforms have their own hardware limitations that are not easily captured by simulators running on highly capable host machines. This allows students the ability to identify the potential failure-points of a design based on unworkable assumptions in real world conditions, which would otherwise have not been possible.

We use the inexpensive Nokia N800 series of ultra-portable Internet Tablets running the Maemo-Linux operating system and the Scratchbox development toolkit. We also provide a simple framework to modify the stock Linux IP routing layer in its TCP/IP network protocol stack. This framework enables students to rapidly prototype their self-designed wireless ad hoc routing algorithms. The implementation phase can be structured to present different levels of challenge, depending on the students' familiarity with C programming under Linux and general technical proficiency. The final implementation provides a functional testbed for analysis and behavior of the routing algorithm's performance. Collaborative learning is encouraged with the instructor acting as facilitator and moderator.

## I. Introduction and Motivation

### *A. Ad Hoc Networks*

Research on wireless ad hoc networks commenced in the 1970s in the form of the Packet Radio Network (PRNET) project of the Defense Advanced Research Projects Agency (DARPA). The Survivable Adaptive Networks (SURAN) project[1] continued this line of work. The PRNET and SURAN projects fueled research into what are known today as ad hoc networks.

Ad hoc networks are wireless data networks formed by dynamic interconnections between autonomous mobile wireless devices (also called nodes). There are no base stations and the network topology is not preplanned; hence the term “ad hoc.” When two nodes in such a network are within transmission range of each other, they establish a communication link. Both data as well as control packets for routing, are transmitted over these bandwidth-constrained wireless links. As nodes move in and out of transmission range of one another, new links are established and some of the old ones may deteriorate and fail. Sometimes, when nodes move, physical

obstructions prevent them from communicating with one another. In addition, wireless links themselves, are prone to intermittent failure due to environmental conditions. At other times, battery powered nodes may fail, rendering them inoperative. These same nodes could become active later, once their batteries are recharged. Thus, the network topology changes frequently, making the task of routing packets difficult.

In addition, all nodes are not in direct communication range of one another. Therefore, data packets must be routed to destination nodes over multiple intermediate hops. In order to do so, each node acts as a router in addition to being a host. However, nodes do not usually have consistent global information about the network since there is no centralized administration to help them in making routing decisions. In the absence of such facilities, each node must operate independently to discover, maintain and use multi-hop routes to different destination nodes. Simply put, an ad hoc network is an unplanned, *multi-hop* network of autonomous nodes that operate in a co-operative fashion. The network is self-organizing, self-healing, and operates with very little or no external control. Such networks are useful for temporary deployment in areas that lack power and communication infrastructure, or where the existing infrastructure has been damaged or destroyed.

### *B. Current Approaches in Teaching Ad Hoc Networks*

For the reasons mentioned in section I.A, the technical challenges in mobile ad hoc routing are formidable. Accordingly, considerable research effort has been invested in designing such routing algorithms. A variety of solution approaches with their attendant strengths and trade-offs are reviewed broadly by Toh[2] and Royer et al.[3] In recent years, this body of research has gradually percolated into advanced graduate level courses in computer networks. However, since this is still very much a nascent field dominated by academic researchers, they (we) tend to disseminate this body of knowledge from a purely research perspective. Thus, the routing approaches are categorized based on event or timing triggers and route selection criteria[4]. Then, the solution algorithms are presented, usually at a very high theoretical level and with almost no implementation detail. Analytical proofs are outlined, and the simulation results from the papers are discussed. The course project sometimes requires students to design (at a high level) and critically analyze their own routing algorithm. If implementation is attempted at all, it is by means of a packaged simulator such as the open source *ns*-2[5] or the commercially available OPNET Modeler[6]. The course usually stops short of actually implementing the student-designed ad hoc routing algorithm on portable hardware. This results in three unfortunate consequences: the implementation is held hostage to the artificial limitations of the simulator, the actual limitations of the candidate hardware (for which the algorithm was designed) are never really tested and, the opportunity to excite the students' interest in the rapidly evolving field is missed.

The above three shortcomings in teaching this very cutting-edge topic provided us the motivation to design a new implementation-driven, hands-on approach to learning. In addition, the NRC reports on “How People Learn” and other publications[7] strongly support the idea of hands-on learning.

The rest of the paper is organized as follows. Section II discusses our mobile wireless networking course and its objectives and teaching challenges. Section III presents our choice of hardware and software platforms, and our proposed framework for implementing student-

designed ad hoc routing algorithms. Section IV makes some interesting observations on the constraints, dynamics and highlights of the learning process in this class based on our past experience. Section V concludes this paper.

## II. Course Objectives and Teaching Methodology

### *A. Course Objectives*

The Department of Electrical and Computer Engineering at Villanova University offers a late-evening course on wireless networking and mobile computing (numbered ECE 8408) in which, ad hoc networks are the main topic of study. The goals of this course are: i. provide students a broad understanding of concepts in wireless networking in general (IEEE 802.11 family of protocols), ii. examine the technical challenges in designing protocols for (harsh) wireless environments, iii. armed with an understanding of these issues, design an efficient (low-bandwidth, low-complexity) ad hoc routing algorithm on student-selected criteria that targets a real world application (e.g. disaster rescue operations), and iv. implement the designed algorithm in software, making design modifications if required, and test its performance (throughput, loss and delay) to verify that it meets its design goals. Needless to say, this is an ambitious schedule and the last goal (implementation and testing) is the hardest to fulfill.

### *B. Teaching Approach and Challenges*

Although this course requires the students to have taken at least one prior undergraduate course in computer networks, the students' preparation varies widely. A significant portion of the class consists of continuing or part-time students who may have taken their pre-requisite course a few years ago. Their majors range from Computer/Electrical Engineering to Systems Engineering. Even the more recent B.S. graduates do not have significant exposure to wireless networking as this topic is rarely taught in any detail in a core 'networks' course. Therefore, the course begins by revisiting the TCP/IP reference model for (wired) networking. As the class works its way through the layers in the reference model during the first two weeks, wireless networking concepts are taught by introducing wireless-specific protocols in place of their wired counterparts in the model. Emphasis is placed on the fact that a wireless protocol used as a drop-in replacement for a wired protocol results in severe performance degradation in the form of lowered throughput, higher packet latencies and higher packet drop rates. These effects are examined in greater detail by studying packet timeout mechanisms and the retransmission behavior at the link level as well as at the TCP layer.

Weeks 3 – 4 are spent reviewing some widely known wireless ad hoc routing protocols and their approaches. Concurrent with this schedule, students are divided into groups of two or three; each group selects a newer ad hoc routing protocol to present to the class. They are given 2 weeks to prepare for a 30 minute presentation scheduled for weeks 5 – 6. At the end of their presentation, they are also required to answer questions from the audience. The instructor's supervision and feedback is critical in the preparation of the class presentations. In the past, without adequate supervision and involvement, students' presentations have varied widely in quality and resulted in inefficient utilization of class time. Student presentations are graded by the instructor as well as the rest of the class on points such as clarity, depth, overall quality and satisfactory responses

to audience's questions. This exercise serves a valuable purpose: it prepares the students to be active learners and provides a solid knowledge foundation for designing their own ad hoc routing algorithm. Constructive, anonymous peer criticism makes them cognizant of their oversights and fosters collaborative learning. In fact, it has been our experience that the instructor's criticism is taken at face value, but peer criticism is usually parried and debated thereby enriching the learning process.

At this point, the semester is at the half-way mark with very little time left to design an original ad hoc routing algorithm and adequately implement and test it. This is primarily due to the steep curve in learning the workings of a simulator such as *ns*-2[5] or OPNET modeler[6]. While it is easy to simulate a prepackaged ad hoc routing algorithm in a simulator such as *ns*-2, it is extremely difficult to actually code a new algorithm within it. By the time students acquire a working understanding of the simulator, there is very little time left to complete a full implementation.

In addition, the more commonly used *ns*-2 simulator makes the unrealistic assumption that the ground topology is flat without any elevations or depressions. Possible obstructions in the signal path and the resulting signal attenuation are similarly ignored. The relatively recent 'shadowing model' for radio frequency (RF) propagation in *ns*-2 attempts to rectify this shortcoming. However, due to modeling inaccuracy, it leads to misleading results. Newport et al.[8] demonstrate a large number of similar environment and RF channel modeling inaccuracies in simulators. It must be noted that it is possible to use more realistic topology and RF models in simulator software. However, since the model must be applied to *each* packet transmission on every hop, the sheer computational complexity makes this exercise futile. On the other hand, no such models are necessary if we use a real world implementation on a physical platform.

Therefore, a conscious decision was made to explore the feasibility of replacing the simulation aspect of the course in favor of a real platform-based implementation (prototype).

## III. Implementation Framework

### *A. Hardware and Software Platform*

Several factors such as capability, flexibility, ease of learning and cost were considered in the selection of a suitable platform for prototyping. The Nokia N800 series of low-cost, pocket-size Internet Tablets running the Debian-derived Maemo flavor of the Linux operating system (kernel 2.6.x) met these requirements (see Figure 1). It has 128 MB RAM, 256 MB non-volatile flash memory for programs, and two secure digital (SD) card slots for data storage. The tablet has a Texas Instruments (TI) OMAP 2420 processor consisting of an ARM1136 core (at 400 MHz), a TI C55x DSP (at 220 MHz) and a PowerVR MBX graphics processing unit (GPU). Other features include a touch sensitive high resolution (800x480) display, built in camera, 2-channel 16-bit audio analog input/output, and an FM radio receiver. External communication is by means of an IEEE 802.11b/g wireless interface, a mini USB port and a Bluetooth interface. In short, the N800 is a full-fledged Linux based computer, with a wide range of capabilities that costs under $200.

*Figure 1: The Nokia N800 Internet Tablet*

In order to compile code for the above ARM-based N800 on an Intel/AMD 32/64 bit host platform, an X86-to-ARM cross-compiler is necessary. The open source Scratchbox toolkit and development environment is recommended for this purpose. The automated download scripts and complete installation instructions are available from the official Maemo website[9]. The Scratchbox environment also acts as a 'sandbox' that separates the cross-compilation environment from the host machine's environment. After compiling programs for the N800 in Scratchbox, the executable binaries and dependent libraries can be transferred to the N800 over a wireless connection using a secure shell (ssh) or physically by means of an SD card.

*B. The Linux Routing Mechanism*

Each Linux device (node) maintains in its kernel a routing table that provides routes to other nodes. Routes can be inserted into the kernel routing table in one of two ways: by manually typing a command on a command line interface, or through dynamic routing algorithms (protocols) operating at the application layer. Most modern applications will use the Netlink socket application programmer interface (API) to manipulate the kernel routing table. At a minimum, each destination node in the routing table is associated with an outgoing interface. When routes are inserted in the kernel routing table, additional information may be associated with a route including its cost metric, scope, and route type. If the destination is not a neighboring node, a next-hop node address is also associated with that destination. When a node wishes to send a packet to a destination, a routing table lookup occurs, and the data is sent to the appropriate next-hop node through the associated outgoing interface. To speed up the routing table lookup process, the Linux kernel also implements a routing cache which is enabled by default. Therefore subsequent lookups to the same destination will use the cached route without consulting the main routing table. Route caching can dramatically speed up the packet forwarding process if the routing table contains hundreds of routes. If there is a cache miss, normal lookup procedures on the main routing table are invoked.

*C. Design Steps for a Multi-hop Ad Hoc Wireless Routing Algorithm*

The first step in implementing any routing algorithm (protocol) is to have a well defined protocol structure and functionality. Therefore, it is essential for students to analyze the shortcomings of existing protocols such as: study optimizations to existing protocols; efficient mechanisms to share state information among nodes (e.g. by controlling packet 'flooding'); study the different criteria for route selection. Therefore, student presentations and critical feedback on those presentations (see section II.B) play an important role. Actions governing route discovery, route addition, route deletion and the corresponding state machine need to be understood in great detail. The route discovery mechanism dynamically discovers other nodes in the network. Existing protocols usually use a form of broadcast message to discover neighboring nodes. The route addition mechanism determines the best route according to a set of design criteria and inserts the route in the kernel routing table. The route deletion mechanism removes stale and

failed routes from the kernel routing table and propagates this information to other network nodes.

The second step builds on the first by incorporating the ideas gleaned from existing protocols to design a new protocol based on new route selection criteria jointly decided by the students and the instructor. A state machine is essential in completely describing the action of a node when events of interest occur. Reception of messages from other nodes such as 'hello', or those indicating route failures/updates are all events of interest. In addition, events triggered internally by the node due to missing or stale routes also constitute events of interests. When such events are triggered, related protocol actions must be defined in pseudo-code in a manner that is consistent with the state diagram. For instance, if a node receives a message about a new route to a previously unknown node, it acts on that information by inserting the new route in its kernel routing table and by informing its neighboring nodes about the update.

The last step involves defining message formats for communicating information between nodes regarding discovery, addition, and deletion of routes. Several message formats may be required such as: a broadcast 'hello' message to discover neighboring nodes; a message to broadcast link costs; a unicast or broadcast message to share route information with others. Of course, the number, length, format and type of messages required all depend on the designed protocol.

*D. Implementation Framework*

Once the protocol behavior has been described with a state machine and its accompanying pseudo-code, the transition to C language code is relatively straightforward. We build on the work by Tonnesen et al. that implements the Optimized Link State Routing Protocol (OLSR)[10]. OLSR is a 'wireless ad hoc' version of the popular Link State Routing (LSR) routing algorithm in wired networks. OLSR runs as a background process or daemon called *olsrd*.

Since *olsrd* has been extensively developed, tested, and is RFC compliant, it serves as a good starting point for studying any routing protocol implementation. In addition, since *olsrd* is released under the open source Berkeley Source Distribution (BSD) license, the source code is available to study, analyze, and extend. Furthermore, *olsrd* has been compiled and tested on a number of operating systems (UNIX, Linux, BSD, Windows), platforms (N800, iPhone, netbooks) and a community wireless mesh network with 2000 nodes.

<u>OLSR Modification: Implementing a Custom Ad Hoc Routing Algorithm</u>

The four main components of *olsrd* are the socket parser, the packet parser, the information repositories, and the scheduler (refer to Figure 2). The socket parser is responsible for constantly polling protocol data. Functions can be registered with the socket parser to appropriately handle incoming data such as passing packets to the packet parser. The packet parser then parses incoming packets and drops, forwards, or further processes the data therein. The information repositories contain tables needed for making routing decisions such as the neighbor table, and the topology tables. The scheduler is responsible for running several registered events at set intervals. Rather than implement an entire routing daemon from scratch, *olsrd* can be used as a base for rapid protocol deployment. To this end, *olsrd* provides a plug-in mechanism to extend or

modify OLSR's functionality in the form of a dynamic linked library. The plug-in interface has access to each main component of *olsrd*, and can call custom functions to modify the functionality. The plug-in approach also allows us to concentrate on developing an ad hoc wireless routing protocol code without getting mired in low-level service code details such as IP hashing, polling sockets, or scheduler optimization. Furthermore, *olsrd* has pre-defined functions for writing to the kernel routing tables, and for defining routing rules through the Netlink socket API. These functions can be called from within our own protocol-specific plug-in to manipulate the kernel's routing decisions.

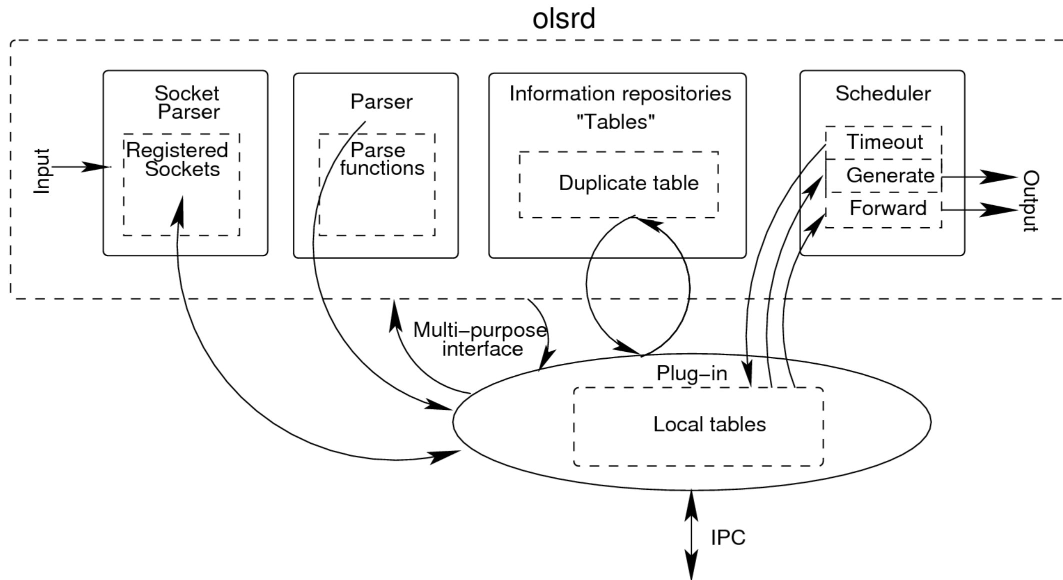


*Figure 2: The olsrd plug-in interface [10]*

We set up our framework by stripping out all OLSR protocol-related code, but retain the socket parser, the packet parser, and the scheduler. Following *olsrd*'s coding convention, the candidate protocol's message formats are defined as structures in C language. Algorithm-specific custom functions are written to transmit routing messages. If these messages need to be transmitted periodically, the corresponding custom function is registered with the scheduler to invoke it repeatedly after set intervals. All messages defined in the pseudo-code can be scheduled in a similar fashion.

The reception of a packet is an event trigger that must be handled by a function written for that purpose. Such functions may need to update its route cost, route timers, and/or the kernel routing table. These functions are registered with the packet parser. When a packet arrives, the packet parser calls the appropriate function based on the message type. State information can be maintained by storing node information in local hash tables based on IP addresses. The information stored includes the next-hop node address, the associated delay, and time-stamps of validity. We create a 'timeout function' which deletes a route from the kernel routing table and recalculates the affected routes. When a new route is added to the kernel routing table, we register this 'timeout function' with the scheduler to trigger its execution after a specified time interval. If updates to the route are received, the timeout event is re-registered with the scheduler

to execute at a later time. Therefore, (stale) routes that are not being updated will be removed dynamically when the scheduler executes the timeout function.

## Single Path Source-Destination Routing

Traditional wired and wireless routing algorithms search for a *single* best, (possibly) multi-hop route between source and destination using some predefined routing criteria and/or metrics. The framework and general guidelines for such an implementation have been presented above. It is easy to use and requires no specialized student knowledge beyond proficiency in C language and a rudimentary knowledge of Linux networking. A short tutorial with a few code examples illustrating the use of *olsrd* plug-in extensions is usually sufficient, and is in preparation. Since this approach does not require the modification and recompilation of the Linux kernel, it promises to dramatically reduce the time required to implement a student-designed ad hoc single-path routing protocol. Thus, routing protocol implementation can be designed as a four week course project assigned in the second half of the semester.

## Multi-Path Source-Destination Routing

If a node actively searches for, and concurrently utilizes *multiple* multi-hop paths to a destination, the routing is termed as multi-path routing. Although multi-path routing is rarely implemented, it is gaining in importance due to its load-balancing ability and fault-tolerance. Multi-path routing was supported in earlier Linux 2.6 kernel code, whereby two route entries to a destination could coexist in the routing cache. However, this feature was recently removed because of poor performance and lack of support from developers. Therefore, implementation of multi-path routing protocols in Linux currently presents a considerable challenge. However, the framework presented above can be easily extended to implement multi-path ad hoc routing protocols as follows.

Finely tuned multi-path routing protocols require a per-packet distribution to effectively balance the traffic load across multiple paths or routes. The Linux kernel does not natively support this scheme, but we do have a few options available to us. We could either disable the routing cache entirely or we could schedule an event to flush the routing cache periodically. Although both these solutions may work, they result in a severe performance hit due to the disabling of the natural caching mechanism. A viable alternative is to exploit the routing cache hash function to insert multiple paths to the same destination in the routing cache.

The routing cache hash function is based on four keys: source address, destination address, type-of-service (TOS) field, and *fwmark* field. Raghunathan and Kumar propose a modification of one of these fields to allow multiple routes to coexist in the routing cache[11]. They modify the packet's TOS field, then make a routing decision, and then restore the TOS field to its original value prior to transmitting the packet. Their scheme requires kernel modification as well as custom kernel modules. In order to avoid kernel modification and to keep the routing algorithm in userspace, we propose that the *fwmark* field be used instead. And, since the *fwmark* field is an internal system data field, and not in the packet header, packets at a node need not be modified, thus minimizing the packet processing latency.

We modify the *fwmark* field on every outgoing packet with *iptables,* the default Linux firewall. All packets are inspected by *iptables* and pass through a series of tables and chains during the kernel's routing process. We specify a rule to queue all outgoing or forwarded packets to userspace through the use of the QUEUE target. In our userspace code, we set the *fwmark* for every outgoing packet depending on the state of the network. We also create a series of user-defined routing tables that are each classified by a different *fwmark*. Routes to a common destination are stored in a separate routing table. Therefore if the routing protocol uses three routes simultaneously, there will be a total of three user-defined routing tables. In order to make a routing decision, a lookup is performed on the routing table that is associated with the packet's *fwmark*. Since the *fwmark* values vary for the same destination, the different routing table entries do not overwrite the existing cache entry to that common destination. This has the effect of achieving multi-path routing with fine-grained, per-packet distribution without the cache miss penalty.

In order to implement a multi-path ad hoc routing protocol, all the steps listed above for single-path routing protocols are required. Namely, the *olsrd* plug-in method as described in the previous section is used. In addition, the multi-path feature is implemented as follows. The stock N800 kernel does not enable policy routing, which is the ability to maintain multiple routing tables. We first enable this option in the kernel configuration, and recompile the kernel. Fortunately, this process is well documented and supported by the Maemo community. After the new kernel is 'flashed' to the N800, we can apply the necessary modifications to the plug-in. On start up, the routing plug-in makes the appropriate system calls to set up *iptables* to send all outgoing packets to the userspace queue. System calls are also issued to associate a *fwmark* value with each routing table that we intend to use.

Updates are now made to hash tables that store the node state information. For each neighboring node, information such as cost, delay, timeout, etc. is stored. In a single-path routing protocol, all routes are written to a common routing table. However, in this case, since routes are written to multiple routing tables, we need to also track the routing table where each route is stored. When a timeout occurs, this information is used to delete only the stale route from the appropriate table rather than deleting all routes to the destination. Of course, the timeout functions and route selection processing must be written to handle multiple routes as well.

Our multi-path routing framework requires the manipulation of the *fwmark* field for all outgoing packets. We utilize the LIBNETFILTER_QUEUE API to manipulate packets from the userspace queue. In a separate thread created from our plug-in, we continuously read packets from the userspace queue, set the *fwmark*, and then return the packet to the kernel for processing. The *fwmark* is set based on the current state of the network. The state is inferred from the number of routes and weights or costs stored in our local hash tables. For instance, when there is only a single next-hop node en-route to a destination (i.e. we have only one route), all outgoing packets to that destination are assigned the same *fwmark* value. Thus 100% of the packets are transmitted to that next-hop node. If two routes exist, in order to distribute 75% of the traffic over the first route, and the remaining traffic over the other, we set 75% of the packets with one *fwmark* value and the other 25% of packets with a different *fwmark*.

Because the above scheme requires the student to maintain a userspace packet queue, as well as multiple routing tables, implementing a multi-path routing protocol is somewhat harder than implementing a single-path routing protocol in the limited time available. Furthermore, since the kernel's stock IP layer is modified, the complete kernel has to be recompiled and flashed on to the N800. Therefore, only those highly motivated students that are well-versed in C programming, Linux internals and networking concepts are advised to take this path.

## IV. Some Observations

The Mobile Computing and Wireless Networking class is offered just once a week in the evenings so as to allow graduate students working full-time jobs to attend. The class size ranges from 10 - 20 students, with about half this number comprised of part-time students working full time technical jobs off-campus. The remaining students are full-time students with some pursuing a thesis option. Thus, there is a fair mix of students, some interested in the research aspects, and others interested in the implementation aspects of the course. Therefore, a hybrid teaching approach with a significant hands-on component is a logical choice.

At the midpoint in the semester, the students' presentations of ad hoc routing algorithms usually mention the results of simulation experiments. As novice learners, students do not critically analyze the assumptions, simplifications, or results presented in the research papers. Even practicing engineers in the class sometimes take the papers' claims at face value. At the conclusion of the students' simulation projects, the instructor revisits the students' original presentation and asks them to comment on what they have learned. At this point, most students who manage to complete the project are usually able to identify several shortcomings in the original paper that they had presented. It is our hope that by reducing the development time through real implementation, most students will be able to complete their implementation project, either individually, or in small groups. In addition it will be instructive to see how far simulation results deviate from real world results.

Since part-time students are usually on campus just one day a week, it is difficult to organize project teams and set mutually acceptable meeting times. A collaborative web based tool such as WebCT or Sakai helps immensely in hosting tutorials, posting ideas, providing peer/instructor feedback and assigning tasks.

## V. Conclusion

Even in a predominantly research dominated field such as wireless ad hoc networking, it is possible to couple research with design implementation. Our experience supports the findings in published studies that show that hands-on projects can considerably enhance the depth in student learning. Hands-on implementation of wireless ad hoc routing algorithms can take the form of either simulation or actual programming for a specific hardware platform. Usually simulation is the preferred method due to ubiquity of tools and cost. However, it has serious shortcomings in the form of a steep learning curve, flawed internal RF propagation models, and its sensitivity to chosen model parameter values. The low-cost, versatile and powerful ARM-based N800 tablet presents a viable alternative in the form of a hardware/software platform for real-world simulation. We present the framework for implementing a wide variety of wireless ad hoc

routing algorithms. It is possible for both categories of algorithms - single path routing as well as multi-path routing – to be implemented by students under this framework. Single path routing algorithms are appropriate for most students since their implementation does not require kernel modification or recompilation. Multi-path routing implementations, on the other hand, do require kernel modification as well as the added programming complexity of maintaining userspace queues and multiple routing tables, and should therefore be assigned only to advanced students for the added challenge.

Currently, work is underway to prepare detailed tutorials, to streamline the framework for implementation, and to design experiments to test student-designed routing algorithms for wireless ad hoc networks. At that time, a detailed course assessment will be conducted that incorporates students' opinions and experiences from surveys to gauge the usefulness as well as shortcomings of this approach, and to correct any deficiencies.

## Acknowledgments

Support from the following sponsors is gratefully acknowledged: The National Science Foundation (under NSF Grant DUE-0837637), The MathWorks Inc., Nokia Corporation, and the Center for Nonlinear Dynamics and Control (CENDAC) at Villanova University, PA.

### Bibliography

1. G. Lauer, “Packet-radio routing,” in M. Steenstrup, ed., *Routing in Communication Networks*, ch. 11, pp. 351-396, Prentice-Hall, NJ, 1995.
2. C. -K. Toh, *Ad Hoc Mobile Wireless Networks: Protocols and Systems*, Prentice Hall, NJ, 2001.
3. E. M. Royer and C. -K. Toh, “A review of current routing protocols for ad hoc mobile wireless networks," *IEEE Personal Communications Magazine*, pp. 46-55, Apr. 1999.
4. S. S. Kulkarni, “Adaptive Load-Balancing Over Multiple Routes in Mobile Ad Hoc Networks”, Ph.D. Dissertation, ch. 1, CS Dept., Univ. of Texas at Dallas, Oct. 2002.
5. The Network Simulator – ns-2, Information Sciences Institute, Univ. of Southern California, <http://www.isi.edu/nsnam/ns/> last accessed Mar. 2010.
6. Mobile ad hoc network models, OPNET Technologies Inc., <http://www.opnet.com/support/des_model_library/manet.html> last accessed Mar. 2010.
7. M. S. Donovan and J. D. Bransford, *eds., How students learn: Science in the classroom*, National Academies Press, 2005.
8. C. Newport, D. Kotz, Y. Yuan, R. S. Gray, J. Liu, and C. Elliot, “Experimental evaluation of wireless simulation assumptions”, *SIMULATION: Trans. of the Society for Modeling and Simulation Intl.*, 83(9), pp. 643—661, Sep. 2007.
9. Maemo.org – Development, <http://maemo.org/development/> last accessed Jan. 2009.
10. A. Tonnesen, O. Kure and A. Hafslund, “The UniK-OLSR plugin library,” *The OLSR Interop and Workshop*, San Diego, Aug. 2004.
11. Raghunathan, V. and Kumar, P. R., “ Wardrop routing in wireless networks,” *IEEE Transactions on Mobile Computing*, 8(5), pp. 636—652, 2009.