An Integrated Environment for Testing Mobile Ad-Hoc Networks *

Yongguang Zhang†
HRL Laboratories, LLC
Malibu, California
ygz@hrl.com

Wei Li
Department of Computer Sciences
The University of Texas at Austin
liwei@cs.utexas.edu

ABSTRACT

Mobile Ad-Hoc Network (MANET) has become an increasingly active research area with a plethora of work in ad-hoc routing, media access, and protocols, etc. However, much of the effort so far has been in simulation with only a few systems that have ever been implemented and none that we know have been tried in a scale beyond a dozen nodes. One reason is the high complexity involved in implementing and testing actual ad-hoc networks, and the lack of software tools for doing so. We have thus built an inexpensive and flexible environment to support such tasks and to facilitate network research. The core component is a mobility emulator to test an ad-hoc network of virtually any scale and with any mobility scenario without actually moving the nodes physically.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*wireless communication*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*routing protocols*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Design, Experimentation

Keywords

Mobile ad hoc networks, MANET, multi-hop routing, emulation, testbed, packet filter

*Updates and other information about this software are available from `http://www.wins.hrl.com/projects/adhoc`

†Dr. Zhang is also an Adjunct Assistant Professor in the Department of Computer Sciences, the University of Texas at Austin

MOBIHOC'02, June 9-11, 2002, EPFL Lausanne, Switzerland.

1. INTRODUCTION

A wireless mobile ad-hoc network (MANET) consists of a collection of "peer" mobile nodes that are capable of communicating with each other without help from a fixed infrastructure. The interconnections between nodes are capable of changing on a continual and arbitrary basis. Nodes within each other's radio range communicate directly via wireless links, while those that are far apart use other nodes as relays. Nodes usually share the same physical media; they transmit and acquire signals at the same frequency band, and follow the same hopping sequence or spreading code. The data-link-layer function manages the wireless link resources and coordinates medium access among neighboring nodes. The network-layer function maintains the multi-hop communication paths across the network; all nodes must function as routers that discover and maintain routes to other nodes in the network. Mobility and volatility are hidden from the applications so that any node can communicate with any other node as if everyone were in a fixed wired network. Applications of ad-hoc networks range from military tactical operations to civil rapid deployment such as emergency search-and-rescue missions, data collection/sensor networks, and instantaneous classroom/meeting room applications. All these suggest that MANET is a very complex system that involves significant interactions across layers and with the environment.

Testing and evaluating MANET algorithms in real systems are necessary for their success in real world use. However, running MANET systems in a non-trivial size is costly due to high complexity, required hardware resources, and inability to test them under a wide range of mobility scenarios. While there has been a plethora of work on MANET such as routing protocols DSR [13], AODV [15], and DSDV [16] to name a few, much of this effort so far has been in simulation only. Only a few systems have been implemented so far, and none we know has been tried in a scale beyond a dozen nodes. One reason for this lack-of-practice is the high complexity involved in implementing and testing any actual ad-hoc network. Not only that the implementation will involve sophisticated system-level programming, but also that a thorough test requires deployment of large-scale test-bed in various mobility patterns. Without comprehensive tools and test bed support, implementing a MANET can be a very daunting task [11].

In this research, we have developed a flexible environment for developing and testing mobile ad-hoc networks. It includes a mobility emulator called *MobiEmu* to test an ad-hoc network of virtually any scale and with any mobility sce-

nario without actually moving the ad-hoc nodes physically. It emulates the physical node movement and the network topology changes, with real implementation on everything above the data-link layer.

2. OVERALL ARCHITECTURE

MobiEmu is a software platform for testing and analyzing "live" ad-hoc network protocols and applications (*test subject*). It uses a fixed network of n linux machines to emulate a mobile ad-hoc network of n nodes. In a real ad-hoc network, the connectivity topology among nodes is dynamic since nodes frequently move in and out of the communication ranges of one another. MobiEmu mimics this real-world situation on the testbed network by dynamically installing or removing packet filters. The goal is to create the same network dynamics for the test subject, so that testing and analyzing ad-hoc networks can be easily done in a laboratory setting.

It is not a new idea to emulate complex network realistically in a laboratory setting. Several such systems have been developed and used for fixed networks, for example, NIST Net[1], the Ohio Network Emulator (ONE)[2], and dummynet[3]. They are mainly designed to emulate wide area networks with different bandwidths and delays, but not for mobile ad-hoc networks. Concept-wise, the packet filtering technique used in MobiEmu is the same as CMU's macfilter tool [11]. However, MobiEmu goes beyond macfilter to be an integrated environment. Section 7 has more detailed comparisons.

MobiEmu control software runs on every testbed machine. The emulation is *scenario*-driven, where the input is a history of locations and movements of every node. MobiEmu also has a user interface component, where the user can preview, control, and visualize the ad-hoc network in action.

To test an ad-hoc network implementation with MobiEmu, the user can run "live" ad-hoc network software (i.e., the test subject) on the testbed machines and treat them as if they were in a real ad-hoc network. Alternatively, the user can implement the ad-hoc network (the test subject) in a set of mobile computers. He/she can then load the MobiEmu control software on each computer to set up a convenient testing environment. Either way, the MobiEmu software ensures that the test subject would not sense the difference and operate the way it would on a real ad-hoc network.

Fig. 1 illustrates the architecture for the MobiEmu ad-hoc network testbed. Each *testbed host* is a computer emulating a mobile node of the ad-hoc network. It connects to all other nodes with a dedicated network (the *testbed network*). The testbed network can be any type of local networks, such as fast Ethernet or 802.11 wireless LAN. Although physically the testbed network is well connected, the MobiEmu system will enforce a partially connected topology at the data-link layer.

The MobiEmu system operates in a master/slave architecture. The *master controller* runs on a dedicated host outside the testbed network; a *slave controller* runs at each testbed host. The master controller controls all slaves and synchronizes their actions: the master dictates when the connectivity topology should change and the slaves enforce such

[1] `http://www.antd.nist.gov/itg/nistnet/`
[2] `http://irg.cs.ohiou.edu/one/`
[3] `http://info.iet.unipi.it/~luigi/ip_dummynet/`

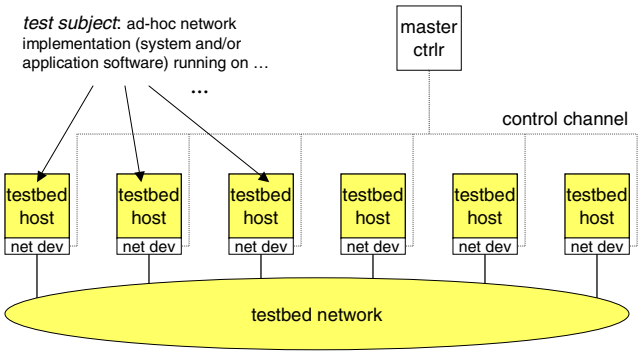

Figure 1: Architecture of the MobiEmu testbed

changes. The master/slave communication is on a *control channel*. The control channel should be separated from the testbed network (e.g., a second Ethernet network) to avoid interfering the ad-hoc network operations. Fig. 2 illustrates the internal structure for MobiEmu system. The details are explained in the following sections.

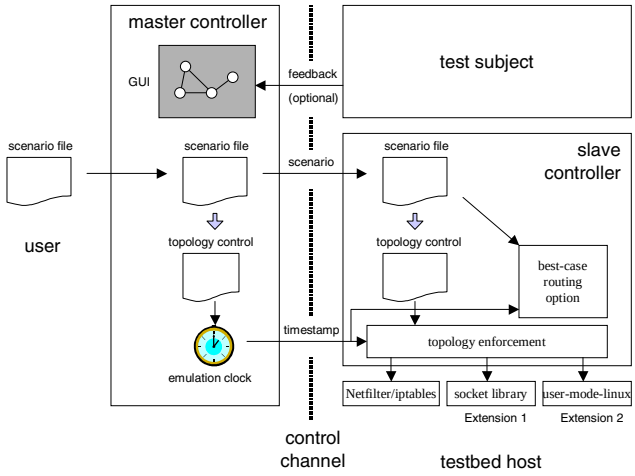

Figure 2: MobiEmu system structure

3. MASTER CONTROLLER AND GUI

The master controller is the brain of the MobiEmu testbed. It contains a graphical user interface (GUI) for controlling the testbed (see Fig. 3).

3.1 Scenario Definition

Before starting the emulation, the user need to load a scenario file through the GUI. The scenario file is a list of timestamped location and movement definitions for all nodes. Currently, MobiEmu accepts two types of format: the native ns2 format, and a simplified format. The first format is the same mobility scenario file format as used in the ns2 network simulator [1] with CMU wireless extension [2]. That is, any files generated by CMU's `setdest` tool (and used in simulations) can be used as is to drive the emulation. A simplified format is also provided to do without the ns2-specific syntax like `$ns_ at ... "$node_(..) setdest ..."`. For example, a line in a simplified scenario file

```
M 59.27 7 130.21 237.29 148.08 207.65 1.89
```

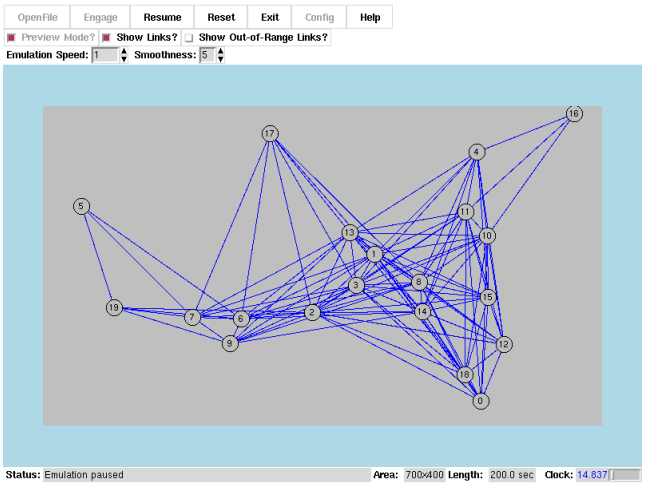

Figure 3: A screen shot of the master controller GUI

means that at 59.27 seconds node 7 is at location (130.21, 237.29) and is moving toward (148.08, 207.65) at the speed of 1.89 meter/sec.

The topology control rules generated from the scenario file are timestamped connectivity rules like:

```
59.27 7 2 1
59.27 7 3 0
```

which means that at 59.27 seconds node 7 can communicate with node 2 but not with node 3.

As the first step of the emulation, the master will broadcast the scenario to all the slaves. Then both the master and the slaves will generate the topology control rules independently. Note that the master will maintain all the rules, but each slave can maintain only the subset of rules that apply to that node.

3.2 Emulation Clock and Synchronization

Once the user starts the emulation, the master controller will start a master clock, called the emulation clock, from time zero (0). Whenever the emulation clock reaches a timestamp associated with a topology control rule, the master will instruct the slaves to execute this rule. This is done by the master broadcasting the timestamp and the slaves enforcing all the rules matching this timestamp. In other words, the slave does not maintain a clock by itself – it depends on the timestamp messages to trigger the rules.

The emulation clock progresses at real time. However, the GUI allows the user to slow down, speed up, pause, or resume the emulation. Nonetheless, this can only change the emulation clock, which determines when the next topology control rule will be applied. It has no effect on the pace of the test subject. For example, pausing the emulation will freeze the motion of all nodes and fix the connectivity topology, but whatever is on the testbed machines will keep on running. This feature can be useful for debugging ad-hoc routing software.

An alternative approach is to use a globally synchronized clock at each slave. Instead of the master broadcasting the timestamp, it can simply broadcast the the clock control commands (speed factor, pause/resume). The advantage is to save the control channel bandwidth used by the timestamp messages. However, as we will show you later (Section 6.1), the bandwidth used by timestamp messages is rather moderate so such saving may not be necessary. Therefore, the main reason why we choose the timestamp approach over the globally synchronized clock approach is for simplicity and the ease of programming (Section 4.2).

3.3 Ad-Hoc Network Visualization

Once the scenario file is loaded, the GUI will visualize the ad-hoc network on a canvas board (see Fig. 3). Each node is represented by a circle with node number at their current location. If two nodes are within the communication range (as defined by the scenario), a solid blue line will be shown between them. Otherwise, they will be linked by a grey line. During the emulation, all nodes and links will move according to the scenario. Each link can change color between blue and grey when the two nodes move in and out of range. The user can also choose not to show any type of links in order to save graph refresh time.

Another distinguished feature of MobiEmu is the capability of *user-defined visualization* for the ad-hoc network. This provides a very useful tool for the user to view the current state of the test subject (ad-hoc network software). For example, some ad-hoc routing algorithms assign different roles to different nodes, such as selecting nodes to be "cluster heads" (e.g., ZRP [5]), or marking nodes "green" or "black" color depending on its internal state (e.g., VDBP [9]). If we are to test these algorithms, we can instrument the routing code so that it sends a command to the master controller whenever the node changes color, and the corresponding node on the canvas board will change color accordingly.

MobiEmu supports the following user-defined visualization:

- *Node color.* Each testbed machine can send commands to change the colors of the corresponding node on the graph. Three types of colors can be changed independently on a node: the circle outline, the circle fill, and the node ID text.
- *Node label.* Each testbed machine can send a text string which will be displayed on the canvas board as a label attached to the corresponding node. This is useful for feedback information about a node, such as ad-hoc routing statistics.
- *Link color.* Each testbed machine can send commands to change the color of a link between this node and any other node. For example, some routing algorithms construct "backbones" (a set of special links) in an ad-hoc network. A test subject for this algorithm can show distinct color for the backbone links in the graph. Another use of link-color visualization is to debug ad-hoc routing protocols. For example, the route daemon can set the color of a link to a node base on that node's route metrics in the route table (e.g., number of hops). This way, we can visualize the route table changes and contrast them with the topology changes.
- *Link label.* This is similar to node label.

Finally, the GUI supports a preview mode in which the user can play the scenario in any speed, without actually engaging the slaves. This is useful to study how nodes move in the given scenario.

3.4 GUI Implementation Choices

The MobiEmu master controller is written entirely in a scripting language (Tcl/Tk). The advantages are that it is portable (currently works in Linux and Windows) and can be easily customized. For example, MobiEmu has been used in a DARPA NGI project at HRL to test multi-tier heterogeneous ad-hoc networks that involve satellites, vehicles, and foot soldiers. First, CMU's `setdest` tool was modified to generate heterogeneous nodes and heterogeneous mobility patterns (the original `setdest` can have only one type of nodes and same communication range). Then, the MobiEmu GUI was easily modified to display different icons for different type of nodes. A canvas dump is given in Fig. 4 to demonstrate the tool's visualization power.

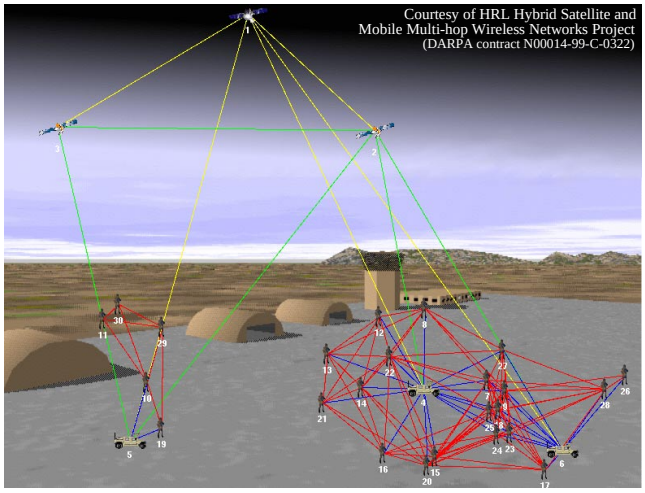

Figure 4: Another screen shot for the ad-hoc network visualization (background is Kandahar airport)

The disadvantage of using Tcl/Tk is the speed. The graph update can sometimes lag behind emulation time if large number of nodes and links are displayed. This can be compensated by using a faster computer or by reducing the "smoothness" value (see Fig. 3) that specifies how frequently the graph should be refreshed. In addition, there is an option in MobiEmu to turn off visualization entirely so that the master controller is dedicated to emulation.

4. SLAVE CONTROLLER AND PACKET FILTERING

In the MobiEmu system, the slave controllers are responsible for enforcing the topology. If according to the current topology node A is out-of-range from node B, the slave controller running at node B must silently block all packets coming from A. This should be done below the network layer so that they are transparent to the test subject (ad-hoc network protocols and applications).

4.1 Packet Filtering

The basic technique used for selectively dropping network packets is the same as many Unix packet-filtering firewalls. Since the MobiEmu slave controller is implemented in Linux, we use the Netfilter/iptables facility [18]. It sits in-between the kernel IP stack and network device drivers and manipulates every packet in or out of this host according to pre-defined rules. Rules can be set or changed at any time through a command interface. The MobiEmu slave uses this interface to set filter rules for the out-of-range nodes.

For example, if node A is out-of-range from node B and A's MAC address is `01:23:45:67:89:0a`, MobiEmu will set the following rule at B:

```
iptables -t mangle -A PREROUTING -m mac \
        --mac-source 01:23:45:67:89:0a -j DROP
```

This rule drops all packets from A's MAC address, without regarding to their actual sources or destinations (whether B is the destination or a hop in the path). Later, once A moves within range, B can simply remove the above rule to re-enable communications from A to B. Multicast and broadcast packets are treated the same way as unicast packets because only the source address is used in filtering.

Usually, the same type of rules will be set at node A to block traffic from the other direction, but it is possible to block one direction and not the other to emulate the effect of a unidirectional link. The node number to MAC address mapping is established at the beginning of the emulation (see the section below).

Using MAC address filtering does have one requirement that the testbed network consist of a single LAN segment (one subnet). Otherwise, when a packet crosses a subnet boundary, the gateway will change the source MAC address. An alternative that works across LANs is to filter on IP address instead of MAC address.

Since iptables or similar packet-filtering systems are standard in Linux kernel, most Linux machines can support MobiEmu without a specially compiled kernel.

4.2 Master/Slave Communications

The master/slave interactions take place over the control channel, which is a separate network so as not to interfere with the testbed network. It is often acceptable, however, to overlay the control channel on the testbed network because the control traffic is rather light and predictable after the emulation starts.

We implement the control channel as a multicast UDP address. To send to the control channel, the master and slaves simply send a well-formatted UDP message to the multicast address. To receive from the control channel, they simply listen to that address. The control channel implements the following interactions:

- *Scenario.* Before each emulation, the master controller informs all slaves the scenario selected for this run.
- *Ping.* The master controller queries all slaves on their status (whether they are alive and ready).
- *Ping Reply.* A slave responds to ping with a reply message. If the slave has finished processing the scenario and is ready for emulation, it includes its IP and MAC addresses in the reply message. This is how each slave will learn every other slave's node number and corresponding network addresses.
- *Timestamp.* The master controller multicasts the emulation clock value to trigger slaves into action (section 3.2).

- *Feedback.* This message is for a slave to feedback information to the master, such as for user-defined visualization purpose (section 3.3).

The first three types of messages are handshakes between master and slave to prepare the emulation. An emulation run is ready only if the master has received positive replies from all slaves. If there is a message loss, the handshakes will be repeated until all slaves are ready. Once the user starts the emulation, the control channel carries only timestamp and feedback messages. Timestamp messages are sent only when the topology changes. To ensure reliability, a timestamp message will be repeated three times in a 10msec interval. Duplicated timestamps will be ignored by the slaves; a lost timestamp message may set back the topology change in that host by 10msec. This lagging is temporary because it will catch up when the next timestamp arrives on time.

As we explained earlier, alternatively we could use globally synchronized clock at each slave and send clock control commands (speed factor, suspend/resume) instead of timestamp messages. Doing so would require each slave to maintain an accurate clock and to set up a chain of timer wake-up events. However, due to the nature of an interpreted language, clocks in Tcl/Tk could easily drift if not programmed carefully, so maintaining a synchronized clock might not be easy. Further, mixing timer events with message-driven logic could make the programming task complicated and error-prone. In contrast, the slave controller is currently implemented with a simple message-driven loop.

4.3 Best-case Ad-hoc Routing

One optional feature of MobiEmu is its built-in routing-layer support for testing higher layer network protocols or applications. Normally, to test any application one must also choose an ad-hoc routing system to run. Obviously, without a functioning ad-hoc routing layer there will be no functioning ad-hoc network. However, some application developers may be interested only in the ad-hoc network environment but not the ad-hoc routing itself. Since a robust and proven ad-hoc routing implementation is not yet widely available, these users should be allowed to test their applications without worrying the routing layer. Examples of such studies may include studying TCP over an ad-hoc network or studying multi-party games in a MANET environment. The "best-case" ad-hoc routing option in MobiEmu supports these applications with a functional ad-hoc network environment without running a routing protocol.

The "best-case" routing works as follows. Whenever a MobiEmu slave controller receives a timestamp messages (i.e., when there is topology change), it computes the shortest path (Dijkstra's algorithm) to all other nodes under the current topology. For each node that is not a direct neighbor, it adds a host-specific route into the system route table (using system command `route add`). Before, packets destined to an out-of-range node would be sent directly and be dropped upon arrival by the packet filtering rules. Now with this route table entry, these packets will be forwarded to the proper next-hop neighbor instead. Later, this entry will be deleted if the corresponding node moves within range. Or, if the route has changed to use a different next-hop neighbor, the route entry will be modified.

Since MobiEmu computes the route from the entire connectivity topology, this is really the best case scenario and no deterministic protocol can produce better route. Even though this is unimplementable in real life, doing so in an emulation environment can provide a base line study. For example, if we have measured the performance of an application using best-case ad-hoc routing, we can expect that to be the upper bound.

For obvious reason, if the test subject is an ad-hoc routing protocol implementation, this routing support option should not be activated.

5. EMULATION WITHOUT A DEDICATED TESTBED

In general, the emulation of an ad-hoc network should take place in a test network. However, in certain circumstance, such a dedicated testbed may not be conveniently available. It is therefore desirable to have MobiEmu emulation environment in general-purpose computer networks, such as the educational computer networks in many schools. However, MobiEmu requires root privilege to set and delete packet filter rules, which is not always allowable in many departmental computers. To overcome this obstacle, we have developed two MobiEmu extensions for unprivileged users to study certain ad-hoc network applications.

5.1 MobiEmu Socket Library

The first extension is to link a test subject ad-hoc network program with a library that does the packet filtering function at user-space. The emulation setup remains the same: we still use a network of computers as the testbed and run slave controller on every computer (as an unprivileged user). Instead of setting rules with iptables, the slave controller uses a temporary file to control whether packets from other nodes should be accepted or not. If a node is out-of-range, the slave controller will create an empty file under the directory `/tmp/MobiEmu-$USER` with the node's IP address as filename. If the node later moves within range, the slave controller will delete this file.

The MobiEmu socket library place a guarding layer above the Unix socket system calls to check `/tmp/MobiEmu-$USER` before or after making the system call. If a file corresponding to the communicating peer exists, it may not execute the system call or return with a different value. For UDP sockets, this is relatively easy to achieve by guarding `recvfrom()` and `sendto()`. For example, the following pseudo code illustrate a guarding function for `recvfrom()`:

```
int recvfrom_MobiEmu(s, ., ., ., ., .)
{
  recvfrom(s, ., ., ., &from, .);
  ip = inet_ntoa(from.sin_addr);
  if exist /tmp/MobiEmu-$USER/$ip {
    // drop this packet
    if s is nonblocking,
      return -1 and set errno to EAGAIN
    otherwise restart this function
  }
}
```

TCP sockets, on the other hand, is much harder to deal with. When two nodes move out of range, all packet flows between them should stop, including acks and retransmissions. Since these are not possible to control from user-space, we can only approximate that with an error return in the guarding functions for `send()` and `recv()`.

There are many tricks one can play in Unix to replace the system calls with their guarding functions. For exam-

ple, one can link a program to a different library at runtime by changing the dynamic linking targets. Or, one can use `ptrace()` facility to modify system calls on the fly (particularly useful for statically linked programs). Finally, if the source code is available, we can insert macro definitions during compilation (for example, using C compiler option `-Drecvmsg=recvmsg_MobiEmu`) and link with the precompiled guarding functions.

This extension is very useful in situations like teaching ad-hoc routing. In a class taught by the first author, a student was asked to implement DSDV [16] as a user-space daemon. The program used UDP multicast to propagate routing information. The student used MobiEmu's socket library extension for intensive testing and debugging. Later, the code was recompiled and linked with standard socket library and it worked on a real ad-hoc network perfectly. The MobiEmu system had greatly facilitated the project.

5.2 Virtual Ad-hoc Network with User-Mode-Linux

The second MobiEmu extension enables a virtual ad-hoc network with User-Mode-Linux. User-Mode-Linux (UML) [3] is a version of Linux operating system that runs in the user-space of another Linux operating system (the host OS). It can be considered as a virtual machine running a full-fledged Linux OS. It also does "virtual" networking among multiple UML instances: each virtual machine support any number of Ethernet devices and Ethernet packets are sent by the host OS to a multicast address, which effectively emulates a shared Ethernet segment. UML supports virtually all Linux system programs and applications. Since it provides full OS services, most systems will not be aware of the difference. UML is therefore a promising platform for testing system services including ad-hoc network implementation. [4]

This extension adds UML to MobiEmu so that UML and its virtual network can be used to test ad-hoc network systems and applications. Fig. 5 illustrates the architecture for this extension. We emulate a virtual ad-hoc network of n mobile nodes with n instance of UML running on n hosts. The test subject runs inside an UML, and each host also runs a MobiEmu slave controller. We modify the UML code to allow the slave controller to set packet filtering rules so that inter-UML packets can be selectively dropped. As usual, the slave controller receive topology control rules from the master and set the filter rules accordingly. This way, the UML network environment will behave like the ad-hoc network we are emulating, and the test subject can be tested under this ad-hoc network dynamics.

Since UML runs entirely in the user space, any unprivileged user can run MobiEmu and create an virtual ad-hoc network in many Linux computer networks with reasonable CPU speed and memory.

6. ISSUES AND DISCUSSION

6.1 Evaluation

We are interested in measuring the performance and overhead of MobiEmu and evaluate its scalability. The first question we want to answer is whether the control channel overhead is significant. As we have mentioned earlier, if the timestamp-message approach generates too much overhead, we will have to switch to a globally synchronized clock and rewrite MobiEmu slaves in different language.

[4]VMware is another virtual PC system for testing purposes. However, it requires root privilege to install and run, making it unsuitable for our "non-dedicated testbed" purpose. Further, it is a commercial software and it will be difficult for us to integrate with MobiEmu like we did to UML.

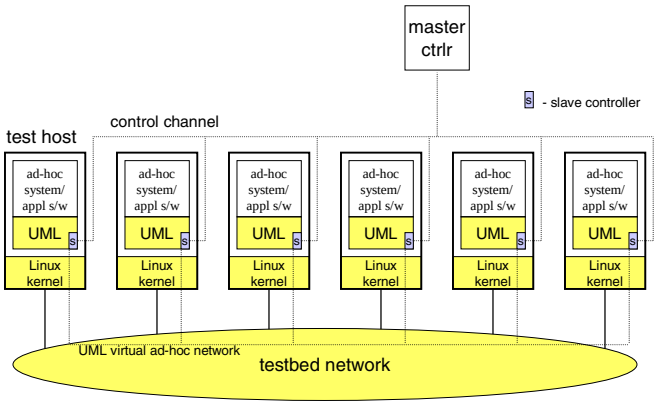

Figure 5: Architecture of MobiEmu's UML Extension

We have conducted a small experiment to measure the control channel overhead. We have installed MobiEmu in a Linux Beowulf cluster of 51 Pentium-III PCs. The testbed network is a Gbps Ethernet switch. Saving one host for the master controller, we are able to set up and emulate an ad-hoc network of 50 nodes. We have measured the control channel traffic volume under various 50-node mobility scenarios (randomly generated by ns2 `setdest`). We start the emulation with any test subject and simply let each testbed machine changes its connectivity according to the topology control rules. We count the number of control channel messages after the emulation starts. Since there is no feedback messages, all the traffic are timestamp multicast messages.

Table 1: Control channel traffic volume for 50-node scenarios

maxspeed	area	volume (# msg/sec)
1	1000x1000	3.327
2	1000x1000	5.885
4	1000x1000	11.452
8	1000x1000	16.123
16	1000x1000	28.955
1	500x500	5.715
2	500x500	11.439
4	500x500	21.444
8	500x500	30.503
16	500x500	52.309

Table 1 illustrates the results. Here the maxspeed (m/s) and area size (m x m) are parameters to the `setdest` scenario generator. From the table we can see that the control traffic volume increases as the maxspeed increases or as the area size decreases. This is reasonable because the number of timestamp messages are proportional to the topology change frequency. But even with the highest volume under our scenarios, it is only 52 message/second, or 20kb/s (each timestamp message is 48-byte including IP/UDP header), which can be easily handled by most networks.

The bottleneck for scalability is on the testbed network. It must have the same bandwidth as the total capacity of

the ad-hoc network. We are currently conducting study to evaluate the scalability of this approach. That is, we want to answer questions like, to emulate an ad-hoc network of certain size and certain density, what are the bandwidth requirements for the testbed network.

6.2 System Support for Implementing Ad-Hoc Routing at User-Space

In addition to the lack of testing environment, another source of frustration for MANET researchers is the sophisticated system-level programming so often required to implement an ad-hoc routing protocol. In a related project at HRL, we have been developing mechanisms to lift such burden from MANET researchers. Our goal is to enhance the underlying system services and to develop MANET-specific APIs, so that most ad-hoc routing protocols can be implementable entirely in user-space.

The first mechanism we worked on is the system support for on-demand routing [7]. Normally, actual packet routing is done in operating system kernel in a table-driven fashion. The role of a routing protocol is to update the route table periodically. Whenever the system needs to send out or forward a packet, it consults this table for a matching entry. If this entry does not exist, the kernel will drop the packet immediately. However, this is not a desirable behavior for many ad-hoc routing protocols, especially those that operate in an "on-demand fashion". In on-demand ad-hoc routing, not all routes would exist *a priori*; some must be "discovered" when needed [13]. In this case, the correct behavior should be: 1) withholding the packet, 2) notifying the ad-hoc routing daemon of a route request, and 3) waiting for route discovery to finish and update the route table. Unfortunately, most common operating systems (including Linux) have yet to support this routing mode directly.

In [7], the authors have discovered a simple way to achieve this in Linux without modifying the kernel. The basic idea is to use a local tunnel device called Universal TUN/TAP (`tun`) as the route target for nodes without yet a valid route. Packets to these nodes are thus routed to this device, and through this device to a user-space daemon for buffering. The daemon will then interact with the ad-hoc routing process, such as invoking route requests, and re-inject the packets back to the kernel (through a raw socket) once the route is ready. We are currently developing the API to allow easy implementation of any on-demand ad-hoc routing protocol in this framework, and we are adding this support to our ad-hoc network developing and testing environment.

7. RELATED WORK

The most notable ad-hoc network experimentation is conducted by researchers at CMU [11, 12, 13]. They have managed to construct a testbed consisted of 5 mobile nodes implemented as cars driving at about 25 to 40 km/h over a course with two stationary nodes separated by a distance of about 700m (2 - 3 radio hops). Later, they have constructed a bigger testbed consisted of 8 nodes driving around a 700m by 300m site. Researchers at BBN have also implemented a 10-node ad-hoc network in a real experiment [17]. One obvious disadvantage of such real testbed strategy is that they are not scalable and not reproducible.

Emulation is the next strategy toward repeatable and scalable experimentation. Researchers in CMU has used the ns2 emulation mode [4] in studying wireless network applications [8]. In such an emulation experiment, each packet from a real machine is sent to a centralized machine on which an ns2 simulation of the desired network is running, and the packet is delayed or dropped according to the behavior determined wholly inside the simulation; if the packet is not dropped within the simulation, it is then resent on the real network at the appropriate time to its real destination. Compared with our MobiEmu system, the ns2 emulation is good for testing end-host applications in an ad-hoc environment, but not for testing ad-hoc network or routing protocols because everything within the boundary of the ad-hoc network is simulated. In contrast, our mobility emulation approach is good for testing core ad-hoc network mechanisms. On the other hand, ns2 emulation is more accurate for end-host applications because it simulates all layers (physical layer and up).

CMU's macfilter [11] is a trace-driven emulation tool that is very similar in concept to our emulation. In fact, the development of our MobiEmu was in part inspired by macfilter. A similar idea is also used in another ad-hoc network testing tool called APE (Ad-hoc Protocol Evaluation) [10]. While macfilter and APE are merely packet-killers driven by trace, our tool allows more user controls, such as coordinating all test hosts and manipulating the emulation clock, and is applicable to more situations, such as emulation without a dedicated testbed. In addition, our tool provides a visualization of the mobility scenario as the emulation goes, as well as a user-defined visualization of the ad-hoc network operations. In contrast, CMU's scenario visualization tool (`ad-hockey` [11]) merely creates and plays wireless scenario without direct interaction with simulation or emulation. To summarize, MobiEmu goes far beyond macfilter and `ad-hockey` to be a complete integrated environment for developing and testing ad-hoc networks.

In addition to the packet-killer function, APE also provides data collection and analysis supports for testing and evaluating ad-hoc routing protocols. These functions are lacking in our MobiEmu. Other trace-driven emulation tools such as CMU and Berkeley's trace-based emulation [14] do not support multi-hop ad-hoc networks. Nor do the previous topology emulation tools that we mentioned earlier.

Described in a recent MobiHoc paper [6], "testbed on a desktop" is a strategy for providing an emulation of real-world wireless environments in a conveniently small "wireless testbed." Each wireless node has an external antenna, and the communication patterns among different nodes are restrained by shielding the antennas, attenuating the gain, and connecting the different wireless nodes through attenuators, splitter/combiners, cables, connectors and terminators. "Testbed on a desktop" is independent of the operating system of the implementation platforms and works with most modern wireless networking interfaces. Compared with our mobility emulation approach, the "testbed on a desktop" strategy provides more faithful emulation of the wireless communication environment. However, the complexity involved in configuring the physical components has really limited its configurations and its scale. In contrast, our tool can provide more flexible emulation and is scalable to a large number of nodes, but with only approximations on the physical and MAC layer.

8. CONCLUSION AND FUTURE WORK

We have developed an integrated environment for develop-

ing and conducting "live" tests of wireless ad-hoc networks in the laboratory setting. The MobiEmu tool allows the test of an ad-hoc network (both systems and applications) in large scale and with any mobility scenarios. It provides flexible control and visualization capabilities for better testing and understanding of the software dynamics. We have used this tool in various research projects with great success. It has reduced the time we spend in testing and debugging ad-hoc network protocol implementations. We have also used it in educational projects where new applications have been tried out in emulated ad-hoc networks.

We fully understand the limitation of our mobility emulation approach. Since it does not emulate the physical and MAC layer, it should only be used in testing but not in performance evaluation. On the other hand, detailed emulation of the physical environment and the MAC protocols would require significant computation and hard real-time processing. To do this, it requires a new architecture and a significantly more powerful testbed.

Researchers at HRL are currently working on the next generation ad-hoc network emulator called WiNE (Wireless Network Emulator). It will simulate both the physical environment (including propagation model, fading and path-loss, jamming and interference, antenna gain and receiver sensitivity, etc.) and the data-link layer (such as 802.11 MAC, among others). The new architecture consists of two main hardware components: a back-end cluster of high-speed processors for computing the physical world, and a front-end cluster of PCs that are similar to the MobiEmu testbed nodes. The detailed real-time simulation of the underlying physical environment is done at the back-end cluster with active-networking and parallel simulation technologies. Coupling the two clusters is a fast interconnect which handles the real-time control (Myrinet). The data traffic is handled by a separate data network, similar to the MobiEmu testbed network.

9. ACKNOWLEDGMENTS

We'd like to thank Gavin Holland of HRL for his encouragement and suggestions in writing this paper. He also provided us the "Kandahar" demo shown in Section 3.4 and a description of the WiNE project that he is working on now. We'd also like to acknowledge Vikas Kawadia of UIUC (who interned with HRL in Summer 2001) for his contributions in the on-demand routing support and in particular the Linux solution (Section 6.2). Thanks also go to Sharad Agarwal of UC Berkeley, Son Dao and Bo Ryu of HRL, and the anonymous reviewers for their useful feedback.

10. REFERENCES

[1] L. Breslau, et.al. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[2] CMU Monarch Group. CMU monarch extensions to the NS-2 simulator. URL: http://monarch.cs.cmu.edu/cmu-ns.html.

[3] J. Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*, Atlanta, GA, Oct. 2000. URL: http://user-mode-linux.sourceforge.net.

[4] K. Fall. Network emulation in the VINT/ns simulator. In *Proceedings of the 4th IEEE Symposium on Computers and Communications (ISCC'99)*, July 1999.

[5] Z. J. Haas. The routing algorithm for the reconfigurable wireless networks. In *Proceedings of IEEE International Conference on Universal Personal Communications (ICUPC'97)*, San Diego, California, Oct. 1997.

[6] J. Kaba and D. Raichle. Testbed on a desktop: Strategies and techniques to support multi-hop manet routing protocol development. In *Proceedings of the 2001 ACM International Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc'01)*, Long Beach, California, Oct. 2001.

[7] V. Kawadia and Y. Zhang. Implementing on-demand routing in linux. HRL Technical Report, 2002.

[8] Q. Ke, D. A. Maltz, and D. B. Johnson. Emulation of multi-hop wireless ad hoc networks. In *Proceedings of 7th International Workshop on Mobile Multimedia Communications (MoMuC'00)*, Oct. 2000.

[9] U. C. Kozat, G. Kondylis, B. Ryu, and M. K. Marina. Virtual dynamic backbone for mobile ad hoc networks. In *Proceedings of IEEE ICC'01*, 2001.

[10] H. Lundgren, D. Lundberg, E. Nordstrm, C. Tschudin, and J. Nielsen. A large-scale testbed for reproducible ad hoc protocol evaluations. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2002)*, Orlando, Florida, Mar. 2002.

[11] D. Maltz, J. Broch, and D. Johnson. Experiences designing and building a multi-hop wireless ad hoc network testbed. Technical Report CMU-CS-99-116, School of Computer Science, Carnegie Mellon Univ. (http://www.monarch.cs.cmu.edu/papers.html), Mar. 1999.

[12] D. Maltz, J. Broch, and D. Johnson. Lessons from a full-scale multihop wireless ad hoc network testbed. *IEEE Personal Communications Magazine*, 8(1):8–15, Feb. 2001.

[13] D. A. Maltz. *On-Demand Routing in Multi-hop Wireless Mobile Ad Hoc Networks*. PhD thesis, Carnegie Mellon University, 2001. URL: http://www.monarch.cs.cmu.edu/monarch-papers/maltz-thesis.ps.gz.

[14] B. Noble, M. Satyanarayanan, G. Ngyuen, and R. Katz. Trace-based mobile network emulation. In *Proceedings of SIGCOMM'97*, Cannes, France, Sept. 1997.

[15] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, Feb. 1999.

[16] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of ACM SIGCOMM'94*, London, U.K., Sept. 1994.

[17] R. Ramanathan and R. Hain. An ad hoc wireless testbed for scalable, adaptive qos support. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2000)*, volume 3, pages 998–1002, Chicago, Illinois, Sept. 2000.

[18] R. Russell. Linux 2.4 packet filtering HOWTO. URL: http://netfilter.samba.org/documentation/.