

TinkerNet 2

Danny Turner
Mike Buchanan

Morgan Conbere
Chris Roberts

Frank Salim
Mike Roberts

Jay Markello
Mike Erlinger

July 18, 2006

Abstract

TinkerNet¹ was developed as a low-cost platform for teaching bottom-up, hands-on networking at the undergraduate level. In the original TinkerNet “throw away” PCs, cheap components, and free software were used to enable students to build their own networking stack from Ethernet up to TCP or UDP, and to have their packets actually transmitted on the wire. Since nothing was emulated, standard networking tools such as packet sniffers could be used to test student generated traffic from a host located on the TinkerNet network. Over the past summer TinkerNet has been moved to a software only system based on User Mode Linux (UML). This paper discusses TinkerNet history and the design, development, and availability of the newest version.

1 Introduction

As computing grows and matures, we are continually adding layers of abstraction and encapsulation to make our day-to-day usage and programming tasks easier. Most of the time this is useful and highly desirable: it is a safe bet that

the Internet would not have taken off in the 90s if every programmer needed to implement their own TCP/IP stack to interact with the network. Abstracting away the growing complexity of a modern computer is a necessary part of computing today.

However, it is occasionally both useful and important to be able to pull back those interfaces and see the actual workings of the systems we build on. Just as it is dangerous for users of the Standard Template Library [17] to not understand the workings of the data structures implemented there, it is important for students to understand the workings of some of our more complicated systems. There is a history of labs and programming environments that allow just that [3, 2, 1, 11, 5]. These systems remove any unnecessary complexity and leave exposed the features most important for students to gain the all-important hands-on understanding that is otherwise lacking. Our system, TinkerNet (software or hardware version), provides that experience for understanding low-level networking. TinkerNet provides direct, real-world access to Ethernet packets, and gives students the features necessary to implement an OSI network stack from the data link layer all the way up through IP to UDP, simplified TCP, and even simple application protocols. We feel that by giving students

¹This work was supported in part by the National Science Foundation under grant NSF-DUE-0443012 to Harvey Mudd College

a hands-on understanding of how the protocols that have been hidden away by the now-universal Berkeley Socket API [6] behave, students will not only have a better grasp of the theoretical workings of an internetwork, but perhaps even have a better understanding of proper usage of sockets.

The 2002 SIGCOMM Workshop on Educational Challenges for Computer Networking [14] exposed many issues related to teaching computer networking: top-down versus bottom-up approach; one course versus many courses; required course versus elective course; and undergraduate versus graduate emphasis. Throughout the workshop discussions one recurring theme emerged: the need for a laboratory to augment lecture. While the principles of networking can be presented in lectures, the group recognized that real understanding occurs when students actively develop and evaluate systems based on those principles – there is no good substitute for hands-on experience with real networks [12] [13]. All of the discussed laboratories shared a few common issues: initial cost of the laboratory and cost of continued maintenance. TinkerNet uses well-known open-source software and inexpensive “obsolete” hardware which we believe mitigates these issues. TinkerNet represents what we believe is a novel and powerful environment for teaching undergraduates about the details of networking and network protocols.

We introduced TinkerNet at ACE in 2004 [7]. Since then we have made improvements in usability, installation and deployment, functionality, and sample laboratory exercises. Most importantly we have packaged it for widespread deployment. The intent of this paper is to reintroduce TinkerNet as a software only system based on UML and to present a more complete set of laboratory exercises (and their grading),

The rest of this paper is organized as follows.

Section 2 gives a brief description of the design, goals, and architecture of the original TinkerNet, followed by a description of our current set of laboratory exercises. We then discuss our work this summer on moving TinkerNet to UML and creating a grading environment for the laboratory exercises. Finally, we conclude with our future plans in Section 8.

2 System Overview

At its core, TinkerNet (Figure 1)

is a system for easily letting students insert code for processing, generating, and responding to network packets into an OS kernel and booting it on a real PC. The system is designed to work with very limited hardware resources, and can likely be assembled with parts that can be found unused in a decent sized institution.

When using TinkerNet, students are provided with a skeleton source tree containing the function prototypes they must implement, as well as a GNU Makefile pre-configured (to build the student’s source, to link the student object code to the existing object code for handling the *admin* network, and to prepare the image to be sent to a *node*). Using tools on the *server*, students can have their kernel remotely booted on one of the *nodes* and view output from that kernel. At no time does the student have to be aware of the existence of the *admin* network or the infrastructure in place to support it. Finally, when the student is done testing a particular build of their kernel, they can simply push a button on the *server* interface (*tinkerboot*) and have their *node* reboot and rejoin the ready pool.

The *nodes* have an extremely limited processing requirement: at most they need to keep up with incoming network traffic. The size of the

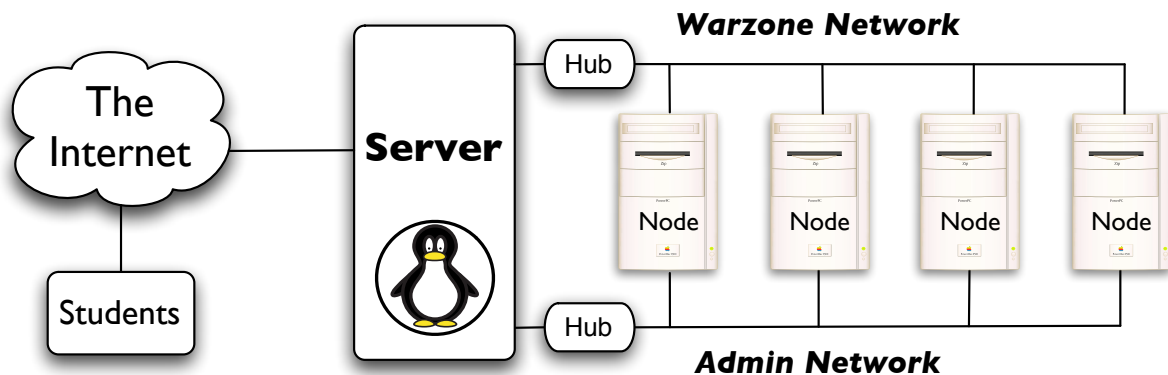


Figure 1: TinkerNet Architecture

kernel that they execute is on the order of three to four megabytes. The *nodes* need no hard drive: most in our installations have been given a network enabled bootloader and boot instructions via a floppy disk. Thus the total requirements for a *node* in a TinkerNet cluster are: two network cables, two network cards, power, and effectively any PC that will still boot from a floppy.

In addition to *nodes* for student kernels to be booted on, there are a few other hardware requirements. The *server* connects to both the *warzone* and *admin* networks, just like a *node*, but it also connects to the institution's production network on a third interface. The *server* provides a home for the students and for all the software to make TinkerNet operational. Also, two hubs or switches are required to create the *admin* and *warzone* networks. Ideally at least the device for the *warzone* network would be a hub, since that makes all the *warzone* traffic available to all the students, giving them a more robust network experience.

2.1 Student Kernels

The kernel running on each *node* is a modified version of OSKit [8]. OSKit was developed and distributed by the University of Utah's Flux Group, but is no longer maintained. OSKit includes file system support, POSIX threading, executable loading, video drivers, and more. The needs of TinkerNet, however, are minimal, and so many of the provided modules are not included in the build process. Modules we do take advantage of include the OSKit Standard C Library implementation, network drivers, and the memory manager.

Students are given three files, two of which are makefile related. The other file is a C file which is a template with all prototypes that students need to complete. From this C template students can access any of the Standard C Library functions, as well as the *netprintf* debugging function.

2.2 Downloading and Managing Student Kernels

To boot the student OSKit kernels we employ a modified version of the GRUB (v0.97) boot loader on each *node*. The modified boot loader is written to a floppy disk (or optionally the *node*'s hard drive), which is then booted each time the *node* is powered up. Upon booting, GRUB sends a packet to the *server* informing the *node* control daemon (*tinkercontroller*) that it is now operational and waiting to boot a kernel. When a student decides to boot a kernel, the kernel is sent through the student interface (*tinkerboot*) to (*tinkercontroller*), which saves the kernel to a special directory. *Tinkercontroller* then sends a special signal to whichever *node* is waiting (a *node* is chosen more or less at random). The selected *node*'s GRUB then processes the signal, which contains the name of the kernel and some parameters to pass to the kernel when booting it. GRUB then uses TFTP to retrieve the kernel over the administrative network.

Tinkercontroller (the *node* control daemon) is the heart of the software side of TinkerNet. This program, written in Python, acts as a mediator between students, administrators, and the TinkerNet *nodes*. It is *Tinkercontroller*'s responsibility to keep track of which *nodes* are free, waiting, and missing, to transfer kernels, to log debug data, and to relay both student and admin commands to the *nodes*. These are tasks that in general could be handled by separate programs, but since they would be accessing a common database of information, it is easier to create threads that handle specific actions. For example any time a student boots a kernel a new thread is spun off to handle the debugging information for that student and node. The thread is then killed off when the student releases the

kernel, and the *node* is rebooted.

2.3 Student Interface

The student interface, (Figure 2) *tinkerboot* has been completely rewritten to use wxPython, the Python version of the popular wxWidgets library, instead of Tk for the interface. New features for the interface include an integrated debug log (the old version opened a separate xterm window), and the ability to send custom packets (UDP packets containing student specified data).

2.4 Administrative Interface

The administrative interface, (Figure 3) *tinkeradmin* allows an administrative user (multiple copies of *tinkeradmin* can be run without interfering) to reboot each *node*, and more importantly to see the status of each *node*: the current user, percentage of packet loss, boot status, MAC addresses, and *node* IP address. The administrative user also has access to the debug log of each *node*, making it easier for a lab assistant to help students debug their networking code.

3 Laboratory Experiments

We have created a semester-long set of laboratory experiments focused on student development of a fully functional network protocol stack. In this set of experiments each new experiment builds on previous experiments. We begin with an experiment to review some issues around programming in C², and then work our way up from

²In the standard undergraduate curriculum at both UCR and HMC, the majority of the programming is in C++ and many students will be unfamiliar with the differences.

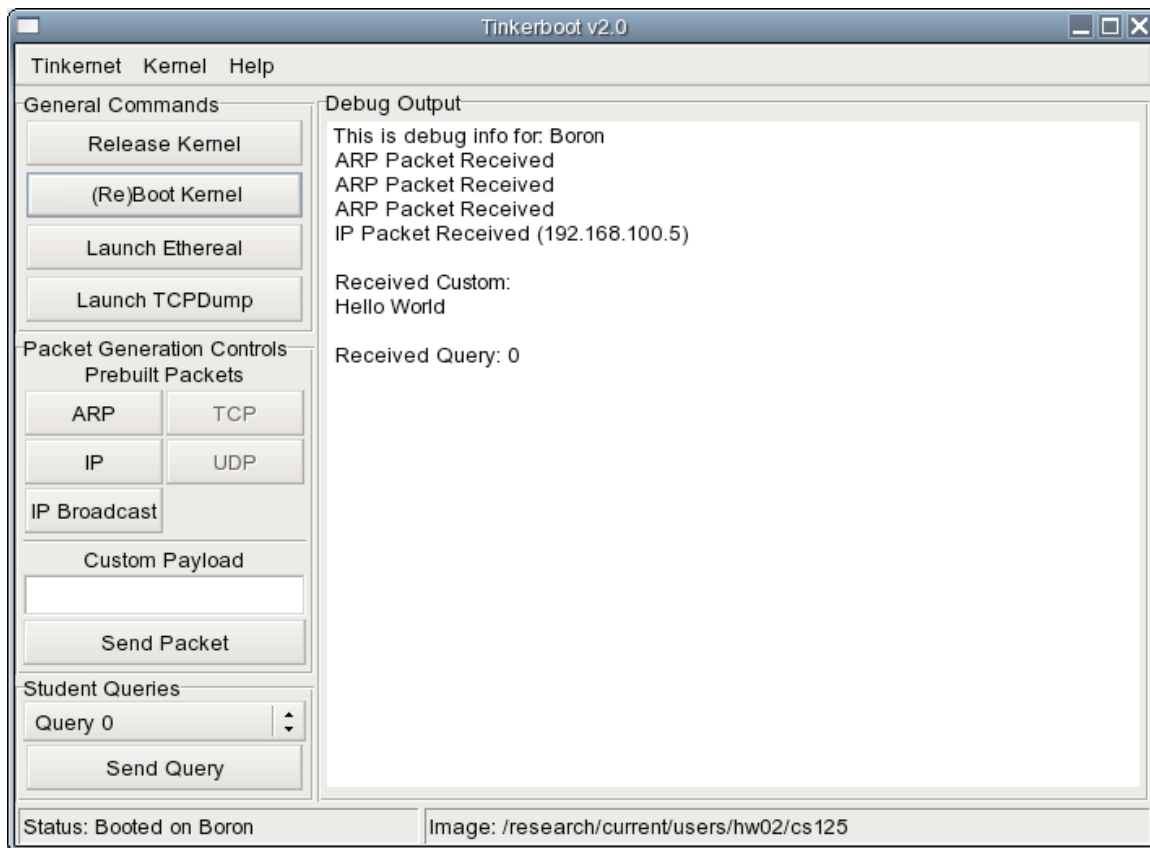


Figure 2: Tinkerboot Interface

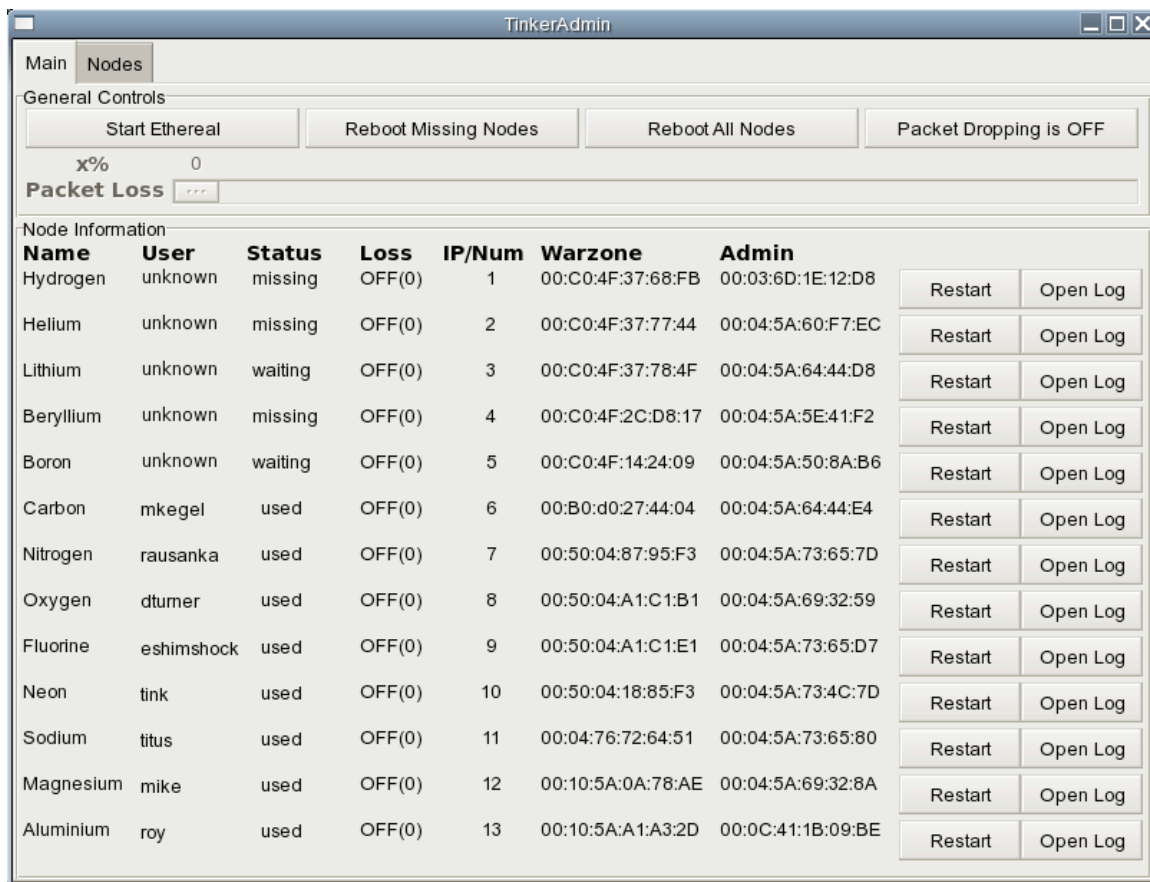


Figure 3: Tinkeradmin Interface

raw Ethernet packets to fully functional IP and then UDP. The final two experiments have students create their protocol and implement Blast [19], a microprotocol which fragments and reassembles large messages. We believe that there are many other experiments which could be created, but that a full implementation of TCP would require much more time than is available in a semester. We envision TinkerNet being used in advanced courses to implement application protocols and/or network devices, such as a router.

3.1 Current Set of Laboratory Experiments

- Lab 1: The goal of this assignment is to gain proficiency with C programming, and to (begin to) learn the differences between C and C++. There are also a few exercises that address networking in general, focusing on concepts like byte ordering and use of structs.
- Lab 2: The goals of this assignment are to gain familiarity with the lab environment, to successfully compile a TinkerNet kernel, and to implement functions that send and receive Ethernet packets.
- Lab 3: The goals of this assignment are to gain more familiarity with the lab environment, to successfully compile a TinkerNet kernel, and to implement functions that send and receive ARP packets.
- Lab 4: The goals of this assignment are to implement an end-host version of the Internet Protocol. The implementation must be able to recognize IP packets addressed to

your IP address and ignore those addressed to other IP addresses.

- Lab 5: The goals of this assignment are to implement the sending and receiving of UDP datagrams, as well as a simple service to test this functionality.
- Lab 6: The goals of this assignment are to design, to create, and to implement a peer-to-peer protocol that will be used to locate other hosts on the network running the same protocol. Using this protocol, two machines will simultaneously boot on TinkerNet, locate each other, and then transmit data between themselves.
- Lab 7: The goals of this assignment are: to implement the microprotocol Blast. Blast fragments and reassembles large messages and attempts to recover from dropped fragments by retransmitting them.

4 TinkerNet 1 Evaluation

In the summer of 2005 TinkerNet received a major overhaul. The goal was to make TinkerNet easily deployable. Code was refactored, commented, and made publicly available. Also a step-by-step guide for building a TinkerNet was put onto a wiki. In the fall of 2005 the refactored TinkerNet was used in the networking class at Harvey Mudd College. Having used the original TinkerNet for 3 years and the refactored TinkerNet for 1 year several observations had become clear.

4.1 Software

TinkerNet is entirely written in C and Python³. The administrative code is written in python, while the code that interfaces with student code is written in C. At the conclusion of the overhaul, TinkerNet's software was well commented and documented. However, there were still some less than ideal situations.

TinkerNet relies heavily on several open source projects. Two of the key components, OSKiT[9] and GRUB⁴ are no longer being maintained. Presently, both are publicly available, compile, and run properly. However, a change in any of the above would severely cripple our ability to make TinkerNet easily deployable.

Other problems exist in the administrative code. Many of these problems are minor. Instead of using the *arping* utility to send an ARP packet TinkerNet uses a much more circuitous process. The system removes the node from its host table, attempts to send a UDP packet via the socket interface, and then relies on the socket interface to send an ARP packet because it has no entry in the host table. Other, unfortunate design choices plague the core of the system. One example is with the *netprintf* function, a users only guaranteed output, involves UDP messages over the administrative network. But, for record keeping as well as reliability the initial message is transmitted over a randomly chosen UDP port and each subsequent message is transmitted to the next port. Further issues are documented on the TinkerNet Wiki.

On the other hand, the TinkerNet frame work lends itself to more than just a networking lab-

oratory. The TinkerNet frame work is sophisticated enough to handle distributed computing as well as an laboratory setup for an operating systems class.

4.2 Labs

A laboratory is no more useful than the labs that are completed in it. As such, several labs have been developed for the TinkerNet Environment. Each lab attempts to have students implement key sections of the networking stack. In the first lab students work at the ethernet level while the last lab has them implementing a subset of TCP.

Students have always been happy with the requirements and instruction provided by the labs. However, the labs are very difficult to grade. The grader had to run each individual's code and try out several different cases. Also no grading criteria had been developed. This lead to grading usually occurring in a pass or fail manner which seemed too inflexible for the time commitment of each lab.

4.3 Hardware

The heart of TinkerNet is the network of nodes that students use to run their code. Each node is a throw away computer, all of our nodes have been computers that were going to be scrapped for parts, with a floppy drive and two network cards. Our nodes run on 16 megabytes of ram, no hard drive, and an integrated video card. In effect the total cost of the network has been, the two hubs and an extra network card for each computer. The minimal cost of this system is one of its best features. Many virtual simulators are neither free nor easy to use. Also many systems using actual computers have heavy requirements for their nodes. **It would be nice to have**

³We leave it as an exercise for the reader to determine which language is truly better.

⁴<http://www.gnu.org/software/grub/grub-legacy.en.html>

some citations to back this up.

While TinkerNet has little monetary cost, we do pay a slight price in terms of time. When a node starts it reads code from its floppy disk that tells it to wait for a kernel to be sent. Once it receives a kernel it boots. To remove a kernel we must restart a computer. This takes approximately 30 seconds and can not be shortened due to the system bios. Under light usage this is not a problem since there are more nodes available than wanted by students. However, in the past this becomes a problem when students use more than their share of the nodes. In one example a student used 5 nodes because he re-launched the *TinkerBoot* utility instead of releasing and reloading kernels.

One final bug has persisted through the refactoring of TinkerNet. Nodes in the network go missing. Nodes go missing in two distinct ways. The primary way nodes are lost is during booting. We send a node a kernel and it boots, however, the server does not acknowledge the node. A poor but successful solution to this problem is to restart missing nodes after a set amount of time. The other way nodes get lost is when they are restarting. When a user is finished with a node they tell *TinkerBoot*. *TinkerBoot* then tells the server to restart the node. However, the node either refuses to restart or when it does restart it never tells the server. To date the only solution we have found for this problem is power cycling the nodes.

5 TinkerNet 2

Tinkernet 2 represents a radical departure from the construction of the original Tinkernet system. In Tinkernet 2 we use User-Mode Linux (UML) to run the user's networking code. This

means that we no longer need to have a rack of cheap machines as was required before. Instead Tinkernet can be run on one nice desktop. This greatly reduced material overhead makes Tinkernet 2 much easier to deploy than Tinkernet 1 was.

5.1 Auto Grader

After discussions with several CS professors it was determined that TinkerNet had strong labs. However, the lack of easy and consistent grading methods was sufficient to prevent its widespread use. As a direct result an autograder was developed for TinkerNet.

The autograder was designed as a modular system. The autograder has three stages. First it takes a number of options from the user: lab to grade, where to write output, and other information needed to grade a lab. These options can be given interactively or via the command line. Next it runs the script to grade the given lab. Finally it updates or creates the grade log.

Each lab in the autograder system is its own class. Each class must conform to a simple interface. To facilitate easy grading of labs a series of helper functions have been created. While python is very useful for writing high level networking applications, it has little support for working at the ethernet level. Scapy⁵ **There probably is a paper we can site here.** is an open source python package for low level networking. Many of the helper functions take advantage of the Scapy interface. But, in maintaining the modularity of the autograder all Scapy functions are wrapped. This allows for the autograder to be minimally modified to work on platforms where there is no direct access to the

⁵<http://www.secdev.org/projects/scapy/>

Ethernet layer.

5.2 Labs

As a direct result of the autograder the Tinker-Net labs were slightly modified. Each lab now requires the ability to have a call and response with the autograder over the network. We estimate that this will add no more than ten minutes to each lab.

Below is an example of the tasks required in the first lab.

- TASK 1

Check incoming frames to see if they are addressed to your system, or to Ethernet *broadcast*. If the frame is not destined for either address, discard it. In the case of this project, discarding a packet involves returning from the *tinkernetrecv* function without processing the packet.

- In all your compares be sure to determine whether you are looking at *char*, *int*, *hex*, etc.
- *tinkernetrecv* is passed the incoming Ethernet frame
- *machineInfo.testMAC* above is the Ethernet address of your box
- create a **struct** that corresponds to the *interesting* header info of an Ethernet packet, i.e., destination, source, packet type fields, etc.
- create a string to represent the *broadcast* address
- use *memcmp* and your **structs** to look for packets for your machine (usual '==' will compare pointers, not structure content).

- TASK 2

Check the protocol type field of the Ethernet frame. If the protocol contained in this Ethernet frame is IP or ARP, print out an appropriate message, such as *ARP Message Seen*. If the encapsulated protocol is not ARP or IP, discard the frame. You can checkout RFCs or other online info for the appropriate information (but not source code) for the appropriate defines.

- TASK 3

Implement a function that takes a hardware address (array of 6 bytes), a pointer to some data, the length of that data, and a protocol number. The function should then compose the proper Ethernet frame and send it on the network using *tinkernetsend*. This function takes two arguments, a pointer to the data, and an integer which contains the length of the data in bytes. Use this function to send the following payload (hex) to tinkerbelle⁶:

```
45 00 00 2e 09 b1 00 00 30 11 36
2e c0 a8 64 c7 c0 a8 64 c8 30 39 27
0f 00 1a 00 00, followed by exactly 18
bytes, which should contain your username,
padded with nulls (0). Tinkerbelle will save
your packet and also return it to you, so
watch for it.
```

- TASK 4

Capture the exchange Using the *Debug Output* window indicate the major steps in your code, eg.,

This is debug info for: Alpha

⁶To find the hardware address, use */sbin/ifconfig* on tinkerbelle2

```

Ready
ARP Message Seen
ARP Message Seen
IP Message Seen
    for this host
IP Message Seen
    for this host
IP Message Seen
    ethernet broadcast
IP Message Seen
    for this host
Sending Task 3 data to tinkerbell12

```

- **TASK 5**

To facilitate the grading of labs we are asking⁷ you to combine the first three task to do the following. After checking the protocol field and destination of each ethernet frame do one of the following four things.

- If the destination is to your machine, and not broadcast, and the type is IP use the function from Task 3 to send the following payload to tinkerbell:
45 00 00 2e 09 b1 00 00 30 11
36 2e c0 a8 64 c7 c0 a8 64 c8
30 39 27 0f 00 1a 00 00 49 50
20 20, followed by exactly 14 bytes, which should contain your user name, padded with nulls (0).
- If the destination is to broadcast and the type is IP use the function from Task 3 to send the following payload to tinkerbell: 45 00 00 2e 09 b1 00 00 30 11 36 2e c0 a8 64 c7 c0 a8 64 c8 30 39 27 0f 00 1a 00 00 49 50 42 20, followed by exactly

14 bytes, which should contain your user name, padded with nulls (0).

- If the destination is to your machine or broadcast, and the type is ARP use the function from Task 3 to send the following payload to tinkerbell:
45 00 00 2e 09 b1 00 00 30 11
36 2e c0 a8 64 c7 c0 a8 64 c8
30 39 27 0f 00 1a 00 00 41 52
50 20, followed by exactly 14 bytes, which should contain your user name, padded with nulls (0).
- If the destination is not to your machine or broadcast do nothing.

5.3 Simulator

We run the networking code that the user wrote in User-Mode Linux, virtualization software that runs linux inside of linux. This gives us easy access to a simulated network. This system is controlled by a pair of python programs, tinkernet.py and tinkerbboot.py. Tinkerbboot.py is the user interface to the system, with which they load their code, send packets, and receive debugging information. Tinkernet.py controls the User-Mode Linux instances on which the user's code is run.

5.3.1 User-Mode Linux

When we first approached the idea of revising the old TinkerNet, one of the most appealing ideas was to use virtual operating systems instead of a cluster of old computer parts. We reviewed our options once we decided to follow this path. We found Xen, User-Mode Linux, and VMware.

Xen was too complicated, and was not small and simple enough for our needs. VMware was a commercial solution, even though the software

⁷Well, not so much asking as telling you, since you will get an F on any lab where you skip this step.

has no cost. User-Mode Linux is a virtualization scheme that has well documented networking and port communication abilities. It is incorporated into the Linux Kernel source code, and thus is stable and up to date with current kernel development.

User-Mode Linux is a patch for the linux Kernel written by Jeff Dike. It has now been incorporated into the Kernel tree and is used for development of new kernel versions. The patch works by changing the memory space that Linux expects to use such that it will fit in the normal process space inside of another linux kernel. Thus, User-Mode Linux appears as a process inside of linux.

We built a custom User-Mode Linux kernel in order to have the smallest memory footprint possible. This allows TinkerNet to handle more users with less delay. This also allows us to remove many extraneous features that could give student code more access to networking tools than we intended.

We also needed to build a root filesystem. The default rootfilesystems distributed by the creators of User-Mode Linux are large and do not include the libraries we need to allow the student code to execute. We attempted to build our own filesystem, but this quickly became unmaintainable. Instead we used a prebuilt filesystem based off of Debian 3.0 Woody. We then installed backports of libnet and libpcap, and had a working filesystem.

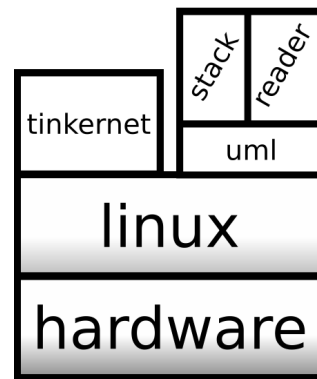
5.3.2 User-space Networking

Once User-Mode Linux was running, we needed a system of running a student written stack in the linux user-space. Thus, we need a framework that can extract packets from a network device and present them in raw form to the user as well

as send raw packets out over a network device. There is not a single library that does this, so we combine the abilities of libnet and libpcap.

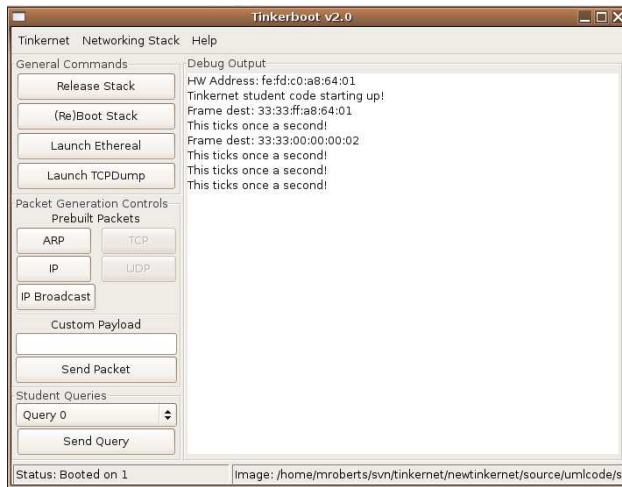
With this userspace code and User-Mode Linux, it is possible to create a virtual node on the network that can safely run student networking code.

5.3.3 tinkernet



This UML based network simulation is controlled by tinkernet.py, a piece of software written in Python. Tinkernet.py keeps track of the UML nodes that are being used on the system. It relays user commands to the UML nodes and sends debug data from the nodes to the users. For each node it has a devoted thread that is connected with the user's instance of tinkerb主oot. Tinkernet.py also has the ability to connect to administrative programs with which an administrator can keep track of what students are using the system. Tinkeradmin.py, which was used in TinkerNet 2, is an example of this type of program. It's goal is to remove any necessity of interacting directly with UML. Ideally, it would not even be necessary for users to know that the system is based on UML.

subsubsection tinkerb主oot



The student interface, `tinkerboot.py`, for virtual TinkerNet is essentially the same as that for TinkerNet with minor modifications. Essentially we had to update the existing codebase to be compatible with the API for `tinkernet.py` which replaced `TinkerController.py`. In doing this we changed the message formatting to use a standard readline function and put new lines at the end of commands written to a socket. We also changed debug to use a socket between the uml node and `tinkernet.py` and another between `tinkernet.py` and `tinkerboot.py` for debug information rather than a file interface. Other than this we did minor changes so that commands would reflect the new paradigm of uml nodes rather than the old paradigm of net-booting physical machines.

In `tinkerboot` you start a node by selecting select and boot in the Networking Stack menu. You then select your userspace stack and `tinkerboot` will send it to `tinkernet` so that it can be run on a uml node.

The General Commands section is mostly self explanatory. Release Stack will shut down the node you own. ReBoot will launch it again and

the Launch commands launch their respective programs.

The Packet Generation Controls section is also somewhat self explanatory although perhaps more complex. When you press ARP we unset the arp table entry and then attempt to send a UDP packet. When we press IP we set the arp table entry and send a UDP packet. IP Broadcast is simpler since we simply send a packet to the broadcast IP address

Custom Payload simply sends a UDP packet with the string you type in to your node.

Query sends the string “query” to the node at port `QUERY_PORT` plus query number.

5.3.4 Test Code

To test the TinkerNet code, a short test sequence was added to the system. This code prints “A packet for me!” any time a packet is received for the node.

5.4 Future Work

To make TinkerNet 3 fully functional we would have to implement a few features that we left out due to time constraints. The most important of these features that we were unable to realize is packet dropping. In TinkerNet 2 `tinkeradmin` (deprecated in TinkerNet 3) had an option which could force a certain percentage of packets in the warzone network to be dropped. This feature would be crucial for any TCP lab to be at all realistic. Besides this we hope to eventually utilize more of user-mode linux’s built in functions. Specifically we would like to make our uml nodes use the user-mode linux option which allows it to interface with `tty`’s and thus be able to remove the overhead of the admin network.

Once we have completed these tasks and assured ourselves that there are few if any bugs left in TinkerNet 3 we plan to deploy it for our networking class so as to get user input and experience running TinkerNet in a real environment. Once this is completed we hope to maintain and update our TinkerNet code base and documentation so that TinkerNet can be widely adopted. We view these efforts as a crucial part of the success and continuation of our project. It is our hope that TinkerNet will be appreciated and well received in the teaching community and that it will be utilized in many courses.

6 Availability

TinkerNet is now available to anyone interested in creating their own TinkerNet. All pertinent information can be found at: <http://www.cs.hmc.edu/tinkernet>.

7 Related Work

TinkerNet, is a low-cost, flexible, stand-alone laboratory for running networking experiments, which combines ideas from various papers [16] [15] [2] [20]; with open source software [8] [10] [18]. Comer's [5] networking laboratory description is similar to TinkerNet, but differs in significant ways. The most significant being Comer's need for special hardware (Console Multiplexor and Reset Controller) and his use of an operating system with limited features and accessibility, XINU [4]. TinkerNet is based on commodity hardware and the readily-available OSKit[8][10], Linux, and GNU software. Our software choices are more widespread within the computing community, and thus TinkerNet will both benefit from the use of these other projects and have

more acceptance because it involves well known and easily available technology. An additional advantage of the TinkerNet approach is that it is accessible even to those institutions (e.g., undergraduate institutions) that do not have on-going research in the area of computer networks.

8 Future Work

We recognize that maintenance of the TinkerNet code base and documentation will be a continuing activity. We view these efforts as a necessary part of sharing TinkerNet. We also plan on developing more laboratory experiments. It is our hope that over the next couple of years TinkerNet can become a part of many networking courses. Our documentation is currently published as a Wiki, in the hopes of fostering a user and development community around the project as it continues to mature.

We also are considering expanding TinkerNet in other directions. We believe that without too much effort TinkerNet can become a basis for an operating system laboratory. We are also considering using the TinkerNet approach to teach systems administration.

9 Acknowledgments

We would like to thank Chris Lundberg and Roy Shea for their initial work on the TinkerNet project, as well as Dr. Mart Molle for his interest and support.

References

- [1] Aburdene, M., et. al. An undergraduate networked system laboratory. In *Proceedings of the 2002 American Society for En-*

- gineering Education Annual Conference and Exposition, Session 2258*. ASEE, 2002.
- [2] Richard Chapman and W. Homer Carlisle. A linux-based lab for operating systems and network courses. In *Linux Journal*, September 1997.
 - [3] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The nachos instructional operating system. In *USENIX Winter*, pages 481–488, 1993.
 - [4] Douglas E. Comer. *Operating System Design, The XINU Approach*. Prentice Hall, 1984. ISBN 0-13-637539-1.
 - [5] Douglas E. Comer. *Hands on Networking with Internet Technologies*. Prentice Hall, 2002. ISBN 0-13-048003-7.
 - [6] Michael J. Donahoo and Kenneth L. Calvert. *TCP/IP Sockets in C*. Academic Press, 2001. ISBN 1-55860-826-5.
 - [7] Mike Erlinger, Mart Molle, Titus Winters, Roy Shea, and Chris Lundberg. Tinkernet: A low-cost networking laboratory. In *Computing Education 2004, Sixth Australasian Computing Education Conference*. ACM Press, January 2004.
 - [8] Flux Group. The oskit project, June 2002.
 - [9] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.
 - [10] Ford, B., et. al. The Flux OSKit: A Substrate for Kernel and Language Research, October 1997.
 - [11] Hill, J. M. D., et. al. Using an isolated network laboratory to teach advanced networks and security. In *SIGCSE Bulletin*. ACM Press, February 2001.
 - [12] Joint Curriculum Task Force. *Computing Curricula 1991*. ACM Press, 1991.
 - [13] Joint Task Force. *Computing Curricula 2001 Computer Science*. ACM Press, 2001.
 - [14] Kurose, J., et. al. Workshop on computer networking: Curriculum designs and educational challenges, August 20 2002.
 - [15] M. Levin. A prototype for a data communications laboratory or a data comm lab in a closet. In *ACM SICSE Bulletin*, volume 29, pages 179–183. ACM Press, 1997.
 - [16] J. Mayo and P. Kearns. A secure-networked laboratory for kernel programming. In *ACM SICSE Bulletin*, volume 30, pages 175–177. ACM Press, September 1998.
 - [17] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996. ISBN 0-201-63398-1.
 - [18] D. Nelson and Ng Y. M. Teaching computer networking using open source software. In *ACM SICSE Bulletin*, volume 32. ACM Press, July 2000.
 - [19] Larry L. Peterson and Bruce S. Davie. *Computer Networks, A Systems Approach*. Morgan Kaufmann, 2003. ISBN 1-55860-832-X.
 - [20] Rickman, J., et. al. Enhancing the computer networking curriculum. In *ACM SICSE Bulletin*, volume 33. ACM Press, June 2001.