TinkerNet: A Low-Cost and Ready-To-Deploy Networking Laboratory Platform

Titus Winters, titus@cs.ucr.edu
Computer Science and Engineering Department
University of California Riverside
Engineering Building Unit 2
Riverside, CA 92521

Ryan Ausanka-Crues, rausanka@cs.hmc.edu
Computer Science Department
Harvey Mudd College
301 Platt Blvd
Claremont, CA 91711

Mark Kegel, mkegel@cs.hmc.edu
Computer Science Department
Harvey Mudd College
301 Platt Blvd
Claremont, CA 91711

Erik Shimshock, eshimsho@cs.hmc.edu
Computer Science Department
Harvey Mudd College
301 Platt Blvd
Claremont, CA 91711

Daniel Turner, dturner@cs.hmc.edu
Computer Science Department
Harvey Mudd College
301 Platt Blvd
Claremont, CA 91711

Mike Erlinger, mike@cs.hmc.edu
Computer Science Department
Harvey Mudd College
301 Platt Blvd
Claremont, CA 91711

Abstract

TinkerNet [7] [1] was developed as a low-cost platform for teaching bottom-up, hands-on networking at the undergraduate level. Using "throw away" PCs, cheap components, and free software, TinkerNet enables 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 is emulated, standard networking tools such as packet sniffers may be used to test student generate traffic from a host located on the TinkerNet network. Over the past summer TinkerNet has matured and advanced thanks to an NSF deployment grant. This paper discusses TinkerNet design, development, and availability.

Keywords: Networks, Networking Labs, Education

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, 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 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.

We introduced TinkerNet at ACE in 2004 [7]. Since then we have made improvements in usability, installation and deployment, functionality, and sam-

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

ple laboratory exercises. Most importantly we have packaged it for widespread deployment. The intent of this paper is to re-introduce TinkerNet, discuss the new developments on the project, enumerate the requirements for institutions considering deploying the system.

The rest of this paper is organized as follows. Section 2 gives a brief description of the design, goals, and architecture of TinkerNet. Section 3 describes the progress that has been made since the last paper. Section 4 describes our current set of laboratory exercises. Section 5 gives some anecdotal discussion of our use of TinkerNet and of the complexity and cost in deploying a new TinkerNet. Section 6 discusses how to acquire the relevant code and documentation. Section 7 acknowledges the systems that perform a similar function to TinkerNet and discusses the pros and cons of each. Finally, we conclude with our future plans in Section 8 and our current conclusions about the system in Section 9.

2 System Overview

At its core, TinkerNet 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 the their *node* reboot and rejoin the ready pool.

2.1 Hardware

TinkerNet, Figure 1, does not emulate network traffic generated by the students. All of the data they request to be sent on the Ethernet is sent on the network, as is, malformed Ethernet frames and all. As such, we have designed the system to function under the assumption that all traffic on the network that the students can send and receive is malicious. As such, every *node* in the network is connected to two disjoint Ethernet networks: one for student traffic (affectionately known as the *warzone*), and one for administrative traffic, *admin* (control messages, file transfers, etc). As such, each *node* needs two network cards.

However, that is the extent of the hardware demands for the *nodes* themselves. The *nodes* have an extremely limited processing requirement: at most they need to keep up with incoming network traffic. The size of the kernel that they execute is on the order of three to four megabytes. We are unaware of any student allocating more than a megabyte of memory for data during execution. Thus, 8 MBs of RAM is sufficient for a *node*, although it is difficult to find even throw-away PCs with less than 32 MBs at this point. 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.

2.2 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.

Due to the unique requirements of the project, only low-level pieces of the OSKit networking modules are included. This gives us access to packets coming in off the network adaptors, but does not include a full socket API. The kernel must be capable of dealing with two entirely different types of traffic, be able to respond to all control packets sent over the administrative network, and not crash, panic or lock up in any way due to errors in student code. Therefore, an implementation was required that would allow the network module code to continue operating even if student code had fallen into an infinite loop or deadlocked.

Fortunately, any OSKit kernel is fully preemptable. In our implementation packets are handled as tasks. When a packet is received by either NIC of a *node* an interrupt is sent to the kernel. Administrative packets, like requests for reboot or acknowledgement of debugging information, are handled immediately in the interrupt. Packets for the student kernel to handle are instead added to a task queue. When not handling an interrupt request, the *node* kernel spends its time processing packets, removing them from the queue and performing the appropriate actions. One of the primary benefits of this design is that even if the student code crashes, so long as the interrupt handler remains in memory the *node* will still be able to reboot and join the ready pool.

To aid in the debugging process students are given the use of a specially designed variant of the *printf* function, which we have dubbed *netprintf*. Students communicate with their kernel through the *node* controller daemon (*tinkercontroller*) residing on the *server*. To access the *node* in any way, *netprintf* relays student debugging information to *tinkercontroller*, which then records the data for each *node* in a temporary file. This file can then be accessed through *tinkercontroller* by either *tinkerboot* (the student interface) or *tinkeradmin* (the administrative interface).

Students are given a makefile, skeleton C code that prototypes all the event handlers necessary to make a Tinkernet kernel link, and a C file with the function prototypes. From this template students can access

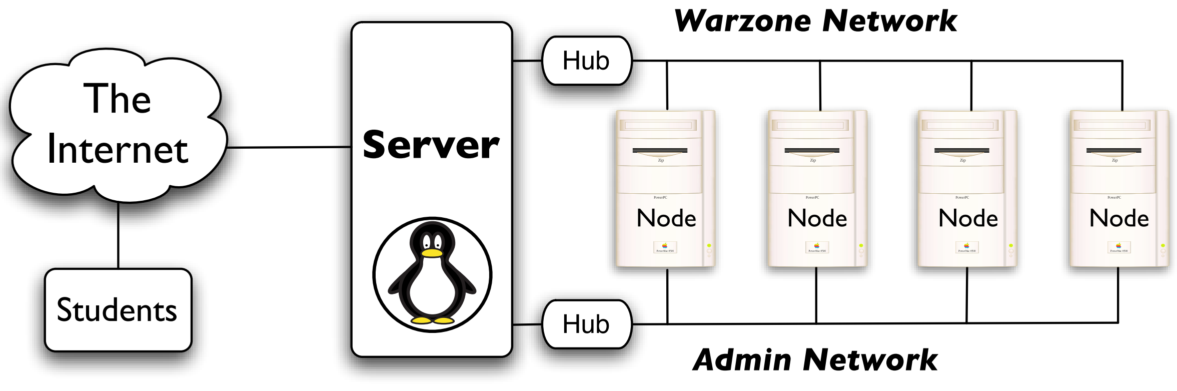


Figure 1: TinkerNet Architecture

any of the Standard C Library functions, as well as the *netprintf* debugging function.

2.3 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.

2.4 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).

2.5 Administrative Interface

The administrative interface, (Figure 3) *tinkeradmin* is a new addition to TinkerNet. It 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.

The *tinkeradmin* program connects to *tinkercontroller* over two different ports. One port is used to send status data and processes input. The other port receives commands from *tinkeradmin* and is authenticated to prevent malicious users from gaining control over the *nodes*.

3 New Features

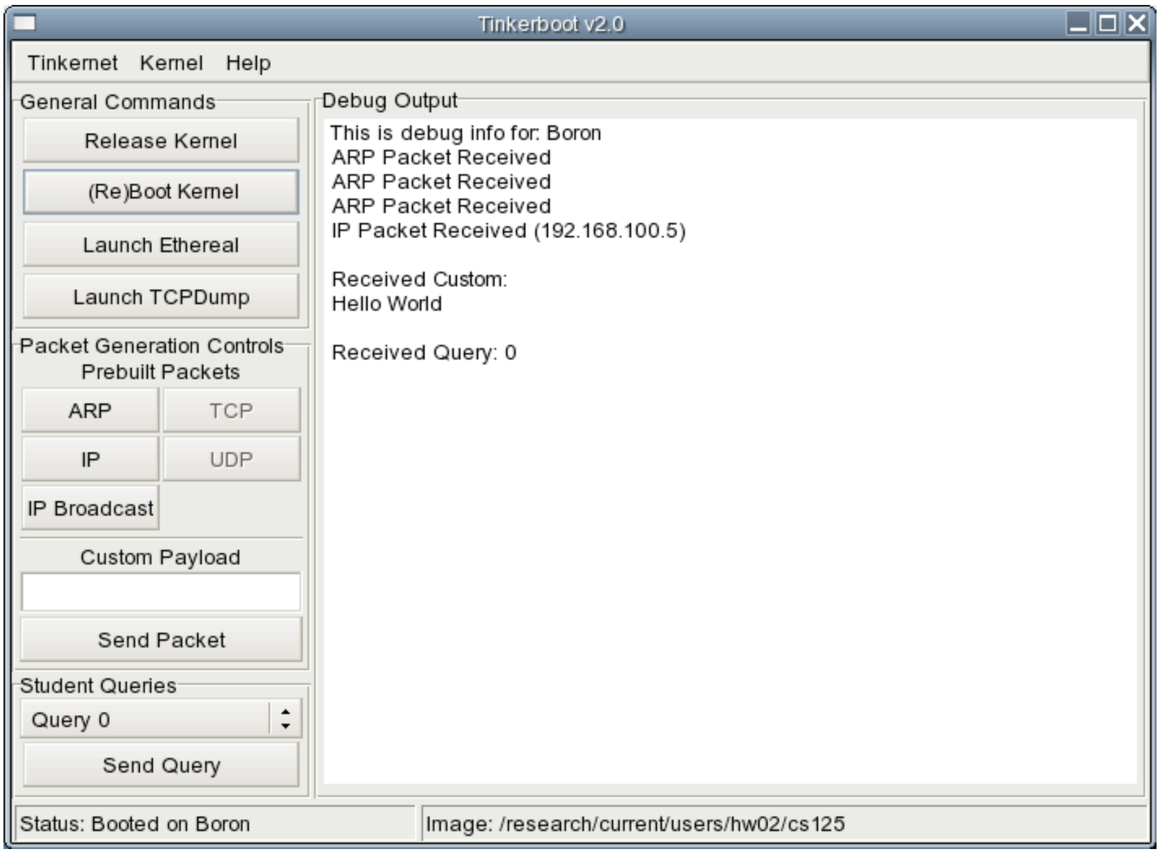
Besides *tinkeradmin* we have made a number of significant improvements to the system. There have been a handful of changes that improve the stability of the student kernels. We have updated all of the interfaces to be more usable and to use more modern GUI systems. Perhaps of most interest are two new features: security features and an administrative ability to add random packet loss on the network,

3.1 Security

One feature that was sorely lacking[2] in TinkerNet is security. The *nodes* are generally controlled by sending UDP packets to known ports for administrative functions like rebooting a *node* in use, sending bootup instructions, and acknowledging *netprintf* messages. Since there was no authentication, any student that knew the port numbers (which could be gathered by running *tcpdump*) could conceivably cause some annoyance by rebooting the machines of other students. We have encountered students that have discovered these port numbers, but we have never seen or heard of students actually being malicious. Although our students may be well behaved, building in some security measures seemed necessary before widespread deployment.

Security for TinkerNet is now accomplished in two ways: *tinkeradmin* replaces all of the old secret loopback services, and uses PGP[10] for authentication. Then, to prevent students from directly sending the control packets on the *admin* network, we added

[2]Although as far as we know NEVER exploited in six TinkerNet course offerings

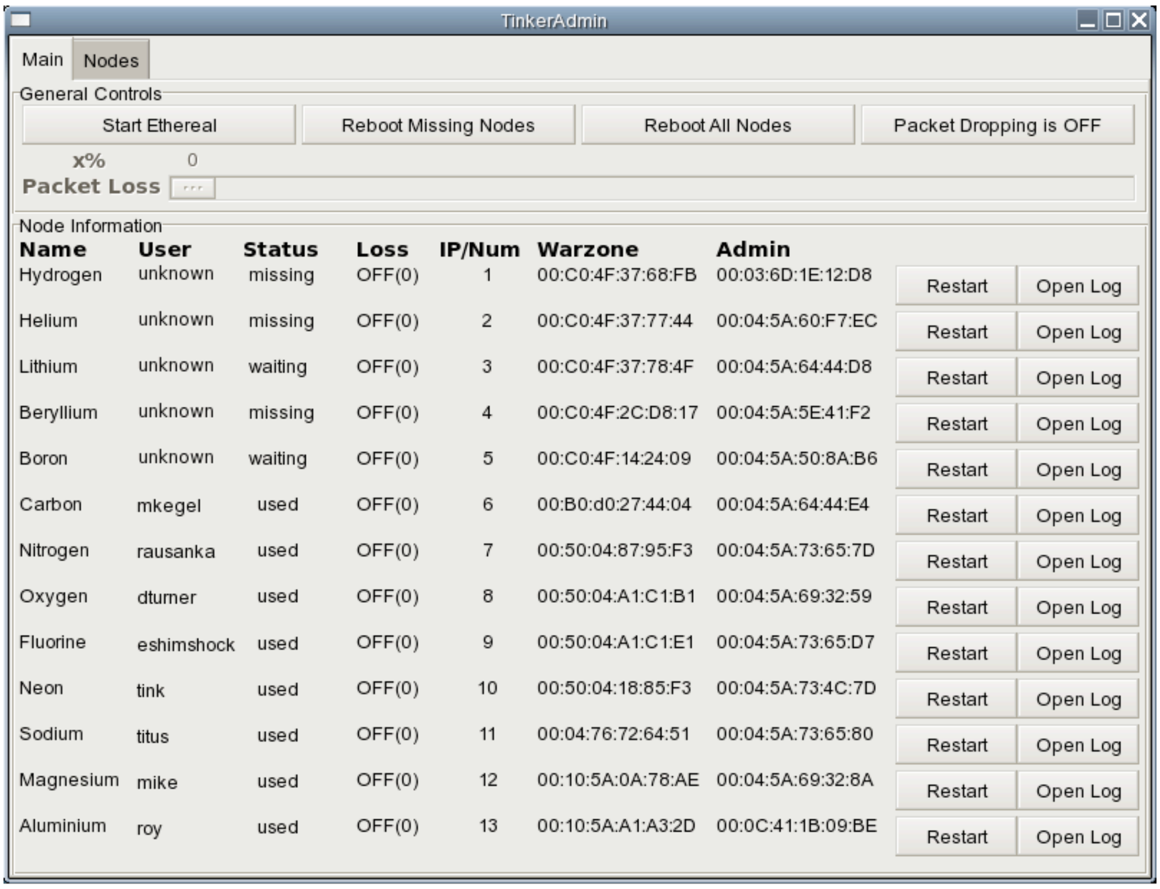


```
This is debug info for: Boron
ARP Packet Received
ARP Packet Received
ARP Packet Received
IP Packet Received (192.168.100.5)

Received Custom:
Hello World

Received Query: 0
```

Status: Booted on Boron

Image: /research/current/users/hw02/cs125

Figure 2: Tinkerboot Interface



Name	User	Status	Loss	IP/Num	Warzone	Admin		
Hydrogen	unknown	missing	OFF(0)	1	00:C0:4F:37:68:FB	00:03:6D:1E:12:D8	Restart	Open Log
Helium	unknown	missing	OFF(0)	2	00:C0:4F:37:77:44	00:04:5A:60:F7:EC	Restart	Open Log
Lithium	unknown	waiting	OFF(0)	3	00:C0:4F:37:78:4F	00:04:5A:64:44:D8	Restart	Open Log
Beryllium	unknown	missing	OFF(0)	4	00:C0:4F:2C:D8:17	00:04:5A:5E:41:F2	Restart	Open Log
Boron	unknown	waiting	OFF(0)	5	00:C0:4F:14:24:09	00:04:5A:50:8A:B6	Restart	Open Log
Carbon	mkegel	used	OFF(0)	6	00:B0:d0:27:44:04	00:04:5A:64:44:E4	Restart	Open Log
Nitrogen	rausanka	used	OFF(0)	7	00:50:04:87:95:F3	00:04:5A:73:65:7D	Restart	Open Log
Oxygen	dturner	used	OFF(0)	8	00:50:04:A1:C1:B1	00:04:5A:69:32:59	Restart	Open Log
Fluorine	eshimshock	used	OFF(0)	9	00:50:04:A1:C1:E1	00:04:5A:73:65:D7	Restart	Open Log
Neon	tink	used	OFF(0)	10	00:50:04:18:85:F3	00:04:5A:73:4C:7D	Restart	Open Log
Sodium	titus	used	OFF(0)	11	00:04:76:72:64:51	00:04:5A:73:65:80	Restart	Open Log
Magnesium	mike	used	OFF(0)	12	00:10:5A:0A:78:AE	00:04:5A:69:32:8A	Restart	Open Log
Aluminium	roy	used	OFF(0)	13	00:10:5A:A1:A3:2D	00:0C:41:1B:09:BE	Restart	Open Log

Figure 3: Tinkeradmin Interface

the restriction that administrative control packets are only processed if they are coming *from* a privileged port. Since non-root users cannot bind to such ports, and the *server* is not routing outside traffic into the *admin* or *warzone* networks, the only way to control the *nodes* is from *tinkeradmin*, and the only way to use *tinkeradmin* is to have access to the TinkerNet administrator's PGP key and the password needed to access it.

3.2 Packet Loss

One exciting new feature in TinkerNet is the ability for the administrator to configure the system to cause random packet loss. Within the *tinkeradmin* interface there is now an option for enabling packet loss. When enabled, a simple slider controls the percentage packet loss experienced on the network. This is implemented by communicating that percentage with the administrative portion of all booted *nodes*. When packet loss is enabled, the *node's* administrative code will compare the loss level with a random number and use this to determine whether or not to enqueue the packet for the student code to process or not. Thus all packets arrive at their destination, and we can enable this packet loss without complex hardware, but we can still produce the effect of having a lossy network. From the point of view of the student code, the packet never arrives. This allows for advanced end-of-semester labs that focus on protocols that work in the face of packet loss and adverse network conditions.

By default packet dropping is off, and can only be modified from within *tinkeradmin*. A default value for packet loss can be set in the configuration file for *tinkercontroller*, but we assume that most users will leave this at zero. Within the *tinkeradmin* interface, packet dropping can be toggled, and the amount of packet loss modified by a slider. Currently only integer values between 0 and 100 percent are accepted.

4 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 C[3], and then work our way up from 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 then 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.

4.1 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.

5 Current Deployment and Interest

Currently both Harvey Mudd College and University of California, Riverside have TinkerNet implementations. TinkerNet has been used in six networking course offerings with up to 65 students in a section. We have found that having one *node* for two students is an adequate size system, as students spend most of their time writing code and compiling, not testing. This can be further reduced to one *node* for every three or four students if students work on the project in pairs with a Pair Programming paradigm. Operationally, we have a set laboratory time when students must be present. These gatherings are monitored by faculty and tutors. But we also make the system available 24/7 with students able to email faculty and tutors with problems.

A large part of our recent activity was to create documentation such that other schools could implement TinkerNet. This documentation has been used by two undergraduates to build a new TinkerNet (*server* and 8 *nodes*). These students were unfamiliar with TinkerNet, either as users or administrators. They were only allowed to communicate by email with the TinkerNet team. The only documentation available to them was found on the TinkerNet web page. We used their comments and questions to improve the documentation, hopefully making it clear enough that anyone can create a TinkerNet.

The building of the new TinkerNet progressed through the following steps:

- Install and configure a Linux version on the *server*. This machine needs to have 3 network interfaces. Debian was used and this took about 3 hours.

[3]In 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.

- Build a rack of machines to act as *nodes*. Since each machine needs 2 network interfaces and various cables must be installed, this activity took the longest time, approximately 1.5 days.
- Create and install the boot disks on all the *nodes*. Given the detailed instructions in the documentation, this activity was done rather quickly, 2 hours.
- Install and configure all TinkerNet software on the *server*. This took about 3 hours.

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] [9] [18]. Comer's [5] networking laboratory description is similar to TinkerNet, but differs in significant ways. The most significant differences 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][9], 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 Conclusions

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]. Many different approaches to networking laboratories were discussed including vendor-specific laboratories, exercises on operational networks, and stand-alone laboratory environments. 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.

10 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 Engineering Education Annual Conference and Exposition, Session 2258*. ASEE, 2002.

[2] R. Chapman and W. H. Carlisle. A linux-based lab for operating systems and network courses. In *Linux Journal*, September 1997.

[3] W. A. Christopher, S. J. Procter, and T. E. Anderson. The nachos instructional operating system. In *USENIX Winter*, pages 481–488, 1993.

[4] D. E. Comer. *Operating System Design, The XINU Approach*. Prentice Hall, 1984. ISBN 0-13-637539-1.

[5] D. E. Comer. *Hands on Networking with Internet Technologies*. Prentice Hall, 2002. ISBN 0-13-048003-7.

[6] M. J. Donahoo and K. L. Calvert. *TCP/IP Sockets in C*. Academic Press, 2001. ISBN 1-55860-826-5.

[7] M. Erlinger, M. Molle, T. Winters, R. Shea, and C. 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] Ford, B., et. al. The Flux OSKit: A Substrate for Kernel and Language Research, October 1997.

[10] S. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly, 2002.

[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] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996. ISBN 0-201-63398-1.

[18] D. Nelson and N. Y. M. Teaching computer networking using open source software. In *ACM SICSE Bulletin*, volume 32. ACM Press, July 2000.

[19] L. L. Peterson and B. 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.