An Educational Networking Framework for Full Layer Implementation and Testing

Keunhong Lee, Joongi Kim, Sue Moon
Department of Computer Science, KAIST
{khlee, joongi}@an.kaist.ac.kr, sbmoon@kaist.edu

ABSTRACT

We present the KENSv2 (KAIST Educational Network System) framework for network protocol implementation. The framework is event-driven to guarantee deterministic behaviour and reproducibility, which in turn delivers ease of debugging and evaluation. Our framework consists of four components: the event generator, the virtual host, the TCP driver and the IP driver. The two drivers are what students have to implement, and we offer to the students the drivers in the binary format for paired testing and debugging. We have developed a test suite that covers three categories of test cases: specification, paired, and logic tests. The framework logs packet transmissions in the PCAP format to allow use of widely available packet analysis tools. Those tools help inspecting logical behaviour of student solutions, such as congestion control. We have designed five step-by-step assignments and evaluated student submissions. With our automated test suite, we have cut down the number of TAs by half for the doubled class size from the previous semester, in total of 3 TAs and 49 students. We plan to continue using KENSv2 in our undergraduate networking course and expand the test suite.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol verification*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education, self-assessment*

General Terms

Design; Verification

Keywords

Educational Networking Framework; Full Layer Implementation; Automated Test Suite; Network protocols; TCP; IP

SIGCSE'15, March 4–7, 2015, Kansas City, MO, USA.
Copyright © 2015 ACM 978-1-4503-2966-8/15/03 ...$15.00.
http://dx.doi.org/10.1145/2676723.2677304.

1 Introduction

In our computer science discipline, hands-on projects challenge students to build systems they learn in class. These projects have always been an integral part of our curricula. Kurose and Ross supplement their textbook on computer networking with Wireshark[1] labs and programming assignments for network applications [1]. Wireshark labs help students to grasp quickly the workings of today's Internet via packet trace analysis without writing code. MYSOCK/STCP[2] have been used in undergraduate networking courses for students to implement the socket API and a simplified TCP connection mechanism to work over simulated packet losses and reordering. Clack [2] provides modular network stack implementations including TCP, and students observe TCP congestions in a graphical way. The popular network simulation tool, `ns2`[3], has accumulated an extensive set of protocol implementations in the past two decades, but its primary goal is to examine the protocol performance against other competing traffic rather than to provide abstractions for learning and implementation. While `ns2` operates strictly in a simulated environment, emulab[4] and ONL (Open Network Laboratory) [3] offer an emulated networking environment with ease of access and configurability. VNS [4] and its successor Mininet [5, 6] have empowered researchers with a container-based emulation environment that closely matches the performance of a real testbed with high fidelity.

For a complete learning experience, students should be able to implement and test the full protocol stack in realistic settings but focus on core networking features only. The assignment should avoid including extra burdens such as kernel programming or concurrency management. It is a common method to punch-hole a set of functions in educational frameworks for systems and let the students fill them as assignments. While it gives the instructor the power to choose at will features for programming assignments, it burdens the students with understanding of the entire framework and interaction of their own code with the rest of the framework. However, this punch-hole approach alone is not suitable for network protocols because of their *paired* and *asynchronous* nature—an implementation of a network protocol cannot operate and be tested without a counterpart. Thus an educational framework for network protocols should be able

[1]`http://www.wireshark.org/`
[2]`http://www.stanford.edu/class/cs244a/hw3/hw3.html`
[3]`http://www.isi.edu/nsnam/ns/`
[4]`http://www.emulab.net`

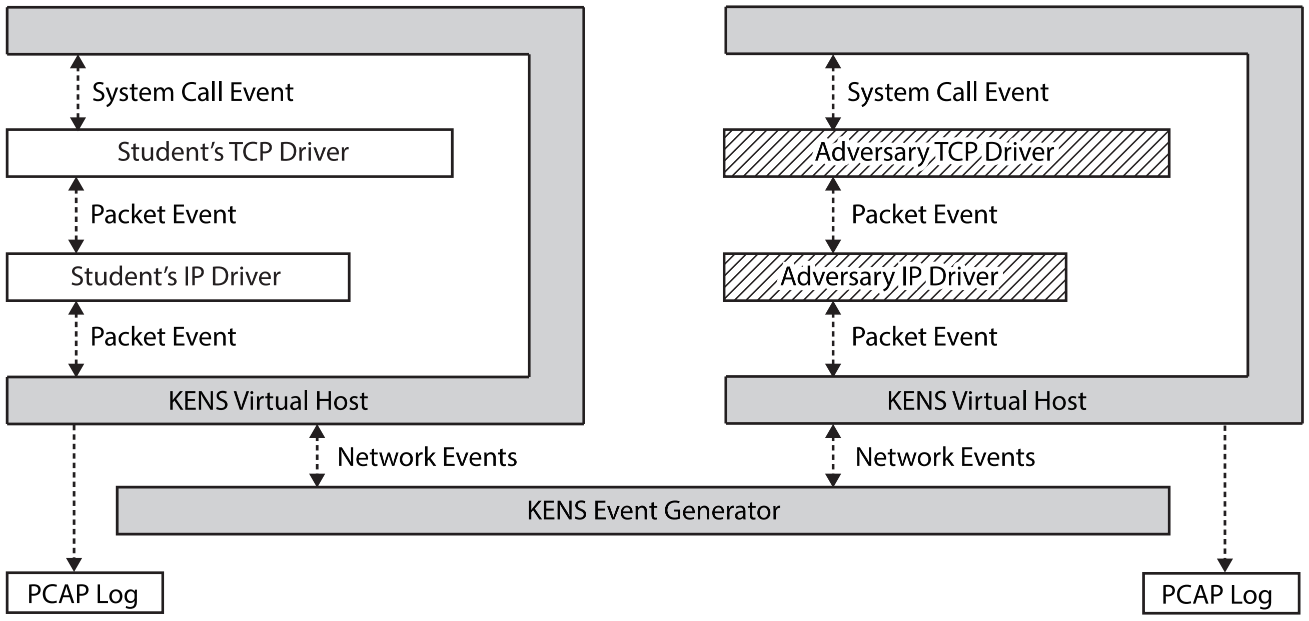

Figure 1: Overview structure of KENSv2 framework

to launch multiple instances of different protocol implementations for paired testing, in addition to punch-holed functions for individual implementations. The framework needs to work as a demonstration of protocol abstractions instead of a simple code template. Moreover, asynchronous executions over multiple hosts make the system unpredictable. The framework should be deterministic to ease tracking, debugging, and evaluation of students' solutions.

In this work we present KENSv2 (KAIST Educational Network System), a framework for students to implement TCP, IP, and routing protocols, and test against adversary implementations. Our framework is event-driven for deterministic behaviour and reproducibility. We have designed KENSv2, implemented it and have used it in an undergraduate computer networking course. In Section 2 we elaborate on our design decisions and in Section 3 present the framework overview. In Section 4 we describe our assignments, and in Section 5 evaluation results. We summarize our experience and lay out future work in Section 6.

2 Design Decisions

KENSv2 is an educational framework, and our design decisions are differ from that of general-purpose frameworks.

2.1 Adversary Implementation

A unique requirement of an educational framework for network programming is the need for adversary implementation. Most functionalities in network protocols are not one-sided but require a responding counterpart. For a student to have a working implementation of `connect()`, s/he must have a matching implementation of `accept()`. We provide these counterparts to allow students to use them for running their code under development. Since this is an educational framework, we should not expose the reference solution. Thus, the counterparts are in the binary format to hide the source code, while providing the required functionality.

Another motivation for adversary implementation is decoupling of the test code and implementation code for Test-driven development (TDD).

For example in Pintos [7], testing thread functionalities, running threads and checking the execution order are all independent from context switching and scheduling and can be performed in separate units. The source code of the testing suite is available with the framework and is independent of the source code being tested. On the contrary, the testing suite of network protocols follows the protocol logic and are not independent from the target implementation. The source code for the network logic should not be exposed to the students, but testing logic itself should be available to students.

2.2 Event-Driven Framework

Reproducibility is critical for an educational framework since students should have a consistent view on how things work [6]. Like other network simulation tools such as ns2, our framework offers a deterministic, reproducible environment that make evaluation straightforward.

To achieve this goal KENSv2 provides an event-driven programming model for layers and a discrete virtual clock. The event-driven model simplifies execution of multiple layers and multiple instances of layer implementation by multiplexing them in a single process. To avoid function calls that cross multiple layers from indefinitely blocking other events, we split a blocking call into two parts, raise and completion, to simulate asynchronous function calls. The virtual clock allows insertion of arbitrary network delay and losses while we run the simulation. We constrain all layer modules to register callback functions for network events. The event generator provides inputs from their upper/lower layers and and the timing information through the registered callback functions.

To deploy KENSv2 on real netweorks, it needs to adapt with existing userspace packet IO schemes such as `netmap`

```
struct kens_tcp_driver_t
{
   //system call mapping
   void (*startup)(kens_system_lib*);
   void (*shutdown)(kens_tcp_driver*);

   my_context (*open)(kens_tcp_driver*, int*);
   void (*close)(kens_tcp_driver*, my_context,int*);
   bool (*bind)(kens_tcp_driver*, my_context,
         const struct sockaddr *, socklen_t, int*);
   bool (*listen)(kens_tcp_driver*, my_context, int, int*);
   bool (*connect)(kens_tcp_driver*, my_context,
         const struct sockaddr *, socklen_t, int*);
   bool (*accept)(kens_tcp_driver*, my_context, int*);
   bool (*getsockname)(kens_tcp_driver*, my_context,
         struct sockaddr *, socklen_t *, int*);
   bool (*getpeername)(kens_tcp_driver*, my_context,
         struct sockaddr *, socklen_t *, int*);

   void (*timer)(kens_tcp_driver*, my_context, int);

   //automatically called by ip layer
   void (*ip_dispatch_tcp)(kens_tcp_driver*,
         struct in_addr, struct in_addr, const void *, size_t);

   //application link
   int (*app_dispatch_tcp)(kens_tcp_driver*, my_context,
         const void*, size_t);
};
```

Figure 2: Function prototypes of the KENSv2 TCP driver

[8], `psio` [9], and Intel DPDK (Data-plane Development Kit) [10]. The challenge here is to combine our event-driven scheduler with the polling loops used in modern userspace packet IO libraries for high-performance. We plane to add an adaptor that runs in polling mode and translates input packets to packet events.

3 KENSv2 Framework

3.1 Framework Overview

Our KENSv2 framework consists of the following four components: an event generator, virtual hosts, IP drivers, and TCP drivers. Figure 1 show the overall architecture of KENSv2. We have implemented our framework in C.

KENSv2 Virtual Host acts as an application layer to TCP and as the combined data link and physical layer to IP. The virtual host encapsulates all the interfaces that the student's code has with the framework. It is also capable of logging all network events in the PCAP[5] format. Students can easily visualize and evaluate the log by using Wireshark[6] or other packet analysis tools. This approach is also used in [11]. Like Web100[12], the virtual host pumps system call events to its TCP driver.

KENSv2 Event Generator feeds events to the virtual hosts. It functions both as a virtual application and the underlying network. It keeps track of a virtual clock and drives the simulation according to it. As the underlying network, it introduces network latency and packet drops. As a virtual application, it initiates system call events towards the TCP layer and they propagate through the TCP and IP layers back to the event generator. By running multiple virtual hosts on top of a single event generator, the framework can

[5]MIME type vnd.tcpdump.pcap. See `http://www.iana.org/assignments/media-types/application/vnd.tcpdump.pcap`

[6]`https://www.wireshark.org/`

```
typedef struct
{
   //access the local routing table
   uint32_t (*ip_host_address)(struct in_addr target);

   //send packet to lower layer
   int (*tcp_dispatch_ip)(struct in_addr src_addr,
         struct in_addr dest_addr, void * data, size_t data_size);
   //send packet to upper layer
   int (*tcp_dispatch_app)(my_context handle,
         const void* data, size_t data_size);

   //wake up sleeping 'accept' system call
   bool (*tcp_passive_open)(my_context server_handle,
         my_context new_handle);
   //wake up sleeping 'connect' system call
   bool (*tcp_active_open)(my_context handle);

   //current time in milliseconds
   int (*tcp_get_mtime)();

   bool (*tcp_register_timer)(my_context context, int mtime);
   void (*tcp_unregister_timer)(my_context context);
   //disconnect tcp-app linkage for a socket hanle
   void (*tcp_shutdown_app)(my_context handle);

}kens_system_lib;
```

Figure 3: System APIs. It shows the case for TCP drivers, but the IP drivers share the similar interface prefixed with "ip_" instead.

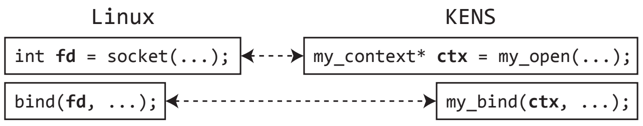

Figure 4: Context mapping between file descriptors and KENSv2 TCP contexts

simulate a network of multiple hosts running a set of applications.

Students' TCP and IP Drivers are the part that students should actually implement. The structure of driver is shown in Figure 2.

Adversary Drivers are binary counterparts for debugging and testing.

3.2 Network Driver Abstraction

Our driver abstraction provides a unified interface for network layer implementations. Drivers cannot access the system or other drivers directly but only through a set of API and utility functions. The API hides environment-specific details from the drivers; students can run their implementations on both real and testing environments seamlessly without modifying their codes. The TCP and IP drivers share the similar abstraction with minor differences depending on what their lower/upper layers are.

System API. Drivers have access to the system resources via the following functions. Figure 3 illustrates them:

- retrieve system clock
- register/unregister timer
- wake up blocked applications
- send data to lower/upper network layer

Network/System Event Callbacks. To interact with other layers and run timed operations, each driver registers a few callback functions:

- initialization/destruction event of the driver
- system call requests

- timer events
- data arrival from lower/upper network layer

Students receive a skeleton of a driver (either TCP or IP) and are assigned to fill the above callback functions. They can use the system API as they need.

To reduce the burden of managing unique numbering for process IDs and file descriptors, we also offer a context mapping scheme as shown in Figure 4, which internally maps each context data to individual TCP socket. The framework translates each file descriptor into the pointer of its context data. Drivers use this information to distinguish each socket from the attached context data.

3.3 Test Suite

At the moment we have implemented only the TCP test suite and are working on the IP test suite. The test suite consists of three parts: specification tests, pairing tests, and logic tests.

Specification test validates the input parameters and return values of the standard system calls. We check the return value and the content of the buffer. For example, the test checks if the TCP driver detects a collision with an existing port number when calling a **bind()** system call. The specification test also needs to repeat with different orders of system call events and packet events as the execution result depends on their execution order. To achieve that, our test suite enumerates every possible ordering. For example, Figure 5 illustrates two different cases for **accept()**: "returning already established connection" and "blocking until new connection has arrived".

Pairing test ensures that the code behaves in a matching manner with its counterpart because the communication may fail regardless of the specification test result. It tests if the data transmitted from one side is received correctly at the other side. For example, even though **write()** system call reports back the success of a transmission, packets may have been dropped or unrecognized in the lower layers or in **read()** implementation running at the destination virtual host due to bugs.

Logic test validates if the implementation conforms with the runtime requirements such as congestion control. Neither specification nor pairing tests covers correctness of context-dependent runtime behaviour because they are blackbox approaches confirming the execution results statically. For example, unsolicited or duplicate ACKs are dependent to the TCP protocol states and not straightforward to validate. We leverage the existing analysis tools for the logic test, which has reduced our development effort greatly at the same time making our framework easy to maintain. The framework generates the packet traces in the widely-used PCAP format. We use Wireshark since it provides context-dependent protocol checks such as advanced TCP filters verifying SEQ/ACK numbers. For simpler tests, more options are available—such as Packetdrill [13] providing packet-level assertions to verify constant fields and checksums.

4 KENSv2 Assignments

We are currently using the prototype of KENSv2 framework in a computer networks course for undergraduate students as a programming assignment. The goal of the assignment is to help the students to understand how actual network layers operate by requiring them to participate in TCP implementation.

```
-----------------------------------------------------------
Starting testListen


     CUnit - A unit testing framework for C - Version 2.1-2
     http://cunit.sourceforge.net/



Suite: testListen
  Test: __testListen_Accept_Before_Connect ...passed
  Test: __testListen_Accept_After_Connect ...passed
  Test: __testListen_Accept_Multiple ...passed
  Test: __testListen_Multiple_Interfaces ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      4      4      4      0        0
             asserts    145    145    145      0      n/a

Elapsed time =    0.000 seconds
testListen: progress = 4/4
-----------------------------------------------------------
```

Figure 5: Example result of a successful test result

What we expect the students to do are as follows:

- Manipulating protocol headers properly
- Demultiplexing flows from the lower layer
- Multiplexing flows from the upper layer
- Context management for demultiplexed flows
- Implementing functions in protocol specification

What we do *NOT* want the students to deal with are as follows:

- Hassles of managing multiple threads and protecting critical regions
- Issues on connecting their network stacks with other prebuilt network stacks
- Reinventing the wheel such as commonly used data structures (e.g. lists, maps)
- Platform compatibility issues

In the assignment we split the TCP layer implementation into a series of small tasks (sub-assignment) as shown in the following class schedule:

- (Week 1) Assigning own TCP context for each socket creation request
- (Week 2) Implementing basic accept/connect protocol
- (Week 3,4) Data transfer on reliable network connection
- (Week 5) Data transfer and connection establishment on unreliable network connection
- (Week 6) Implementing basic AIMD congestion algorithm

We gave students two weeks to submit each sub-assignment, and they had a week off between each assignment. As we did not provide any skeleton codes except the abstract function interfaces, students had to design their own TCP context data structure, demultiplexer, reorder window, etc. We left the implementation details to the students' discretion.

In spring 2014, 49 students participated in the project, supervised by 3 TAs. Students worked in pairs as a team to work on the assignments together. This was an incremental assignment and the final submissions consist of hundreds to 2K lines of code.

```
-----------------------------------------------------------
Starting testBind


     CUnit - A unit testing framework for C - Version 2.1-2
     http://cunit.sourceforge.net/


Suite: testBind
  Test: __testBind_Simple ...passed
  Test: __testBind_GetSockName ...passed
  Test: __testBind_DoubleBind ...passed
  Test: __testBind_OverlapPort ...passed
  Test: __testBind_OverlapClosed ...passed
  Test: __testBind_DifferentIP_SamePort ...FAILED
    1. testbind.c:244  - CU_ASSERT_EQUAL(err,0)
    2. testbind.c:245  - CU_ASSERT_TRUE(ret)
  Test: __testBind_SameIP_DifferentPort ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      7      7      6      1        0
             asserts     60     60     58      2      n/a

Elapsed time =    0.000 seconds
testBind: progress = 6/7
-----------------------------------------------------------
```

Figure 6: Example result of a failed test result

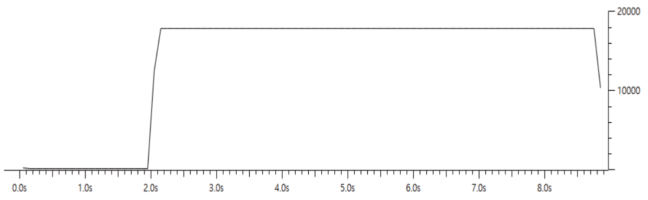

Figure 7: Transmission rate over reliable network

5 Evaluation of Students' Submissions

We had 49 students working in pair and collected 23 submissions for each assignment. In this section, we introduce how we evaluate the assignments using our framework.

5.1 Correctness of system call functionality

Each system call implementation is required to satisfy some functionalities. Some examples of functionalities we want to verify include:

- Finding existing TCP context with port number/IP address
- Checking port number collision while binding address to socket
- Returning correct socket address on getsockname
- Backlog management

Our test suite contains assert statements for system calls on various execution environment. We have placed virtual system events to cover every TCP state transitions. Also, we have tests that covers some complicated situations. Figure 6 contains corner cases of **bind()** system call. We verify the functionalities by checking whether the students' implementation passes all asserts.

5.2 Correctness of Data Transfer

The core function of TCP is to send data from one to the other. We verify it by comparing the sent data and the received data byte-by-byte. This transmission request is handled by a virtual user application that generates proper system call requests (socket-connect-write-close).

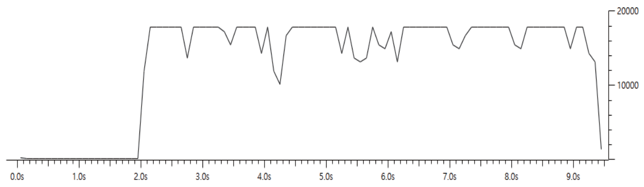

Figure 8: Transmission rate over reliable network (0.1% drop rate)

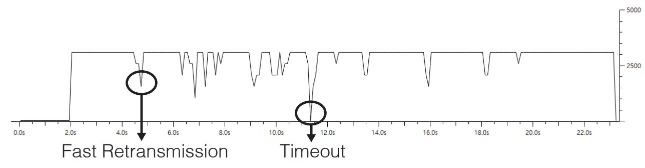

Figure 9: Bytes on flight over unreliable network (0.1% drop rate)

5.3 Correctness of Header Manipulation

We want to check the correctness of TCP checksum, whether a user has sent unseen ACK number, etc. Wireshark provides most verification algorithms we want. After the test application is finished, we analyze the generated PCAP log via Wireshark and check whether an implementation violates the TCP protocol compatibility.

5.4 Reliable Transfer over Unreliable Network

KENSv2 physical layer supports artificial drop/reorder/corruption of packets. The framework allows the verification of TCP behaviours on unreliable connections by simply turning on the unreliable transfer mode. Our Week 5 assignment is simply announced as passing same test suite while unreliable transfer mode is turned on. Students can observe the realistic dynamics of Bytes Per Second under these conditions.

5.5 Implementing congestion control and its verification

Students can implement any TCP congestion control algorithms as KENSv2 provides the virtual system clock and registration of timers. We offered students to implement TCP Reno algorithm[7]. Without any packet drops, the network I/O throughput looks flat like Figure 7. If there are some packet drops, the network throughput looks like Figure 8. However, this does not imply that AIMD congestion control is working. To distinguish AIMD congestion control from multiple retransmissions (though both supports reliable transfers over unreliable network), we have applied advanced TCP filters from Wireshark. These filters compute and compare SEQ/ACK numbers and allows us to determine how many bytes are on flight and not ACKed. Figure 9 illustrates the halved window size on fast-retransmission and the slow start phase on timeout. Wireshark filters also detect fast-retransmissions and timeouts. Currently we have manually observe the macroscopic behaviour of congestion control because the desired windows size is implementation-specific, for example, depends on the initial window size and round-up/round-down choices. However, we plan to adopt TShark (Terminal version of Wireshark) to automate parsing the filter results and evaluation of the logic test.

[7]`http://tools.ietf.org/html/rfc6582`

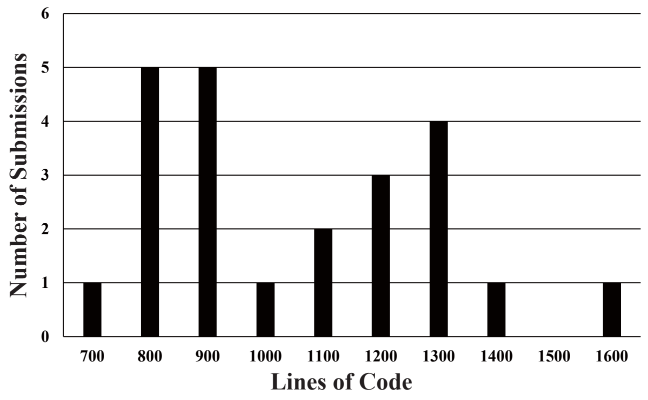

Figure 10: LoC distribution for the final project

5.6 Diversity of Submissions

Until the end of the semester, we had 23 submissions on total. Figure 10 shows the distribution of LoC[8] of all submissions. The shortest solution had 692 LoC and the longest had 1588 LoC. This high variation of LoC shows that our abstraction is general and embraces diverse solutions by students, *i.e.*, freedom of design. Also, the amount of assignment is about a thousand LoC, which is suitable for a single semester assignment.

6 Summary and Future Work

KENSv2 is an educational framework for transport and network layer implementation that supports:

- Realistic Protocol Implementation
- Incremental Development
- Automated Test Suite
- Packet Level Inspection

Our improvements are based on eight years of accumulated lab experiences. In Spring 2014 we are already experiencing benefits from the improvements. We could run TCP implementing assignments over 49 students having 3 TAs for management, which is halved from the previous semester. We plan to release the refactored framework as an open-source software and encourage other universities to try KENSv2 by this year.

We have the following extension ideas for KENSv2. The first is to extend our automated test suite to cover the IP layer. The current version has a functional interference and a reference implementation of the IP layer including packet fragmentation and reassembly, but lacks automated test suites. The second is to redesign KENSv2's data link layer as a generalized adaptor that can receive/transmit packets from/to various underlying packet IO schemes, such as userspace packet IO libraries or the native Linux kernel. The third is to add an IPv6 layer. We expect that IPv6 will become a standard suite for educational frameworks as its adoption is going to accelerate in the near future and the IPv4 address space is being exhausted. Finally, though our system call abstraction for blocking calls can represent the blocking behaviour of both **connect()** and **accept()**, the test code itself does not look like real-world TCP applications. We have a plan to extend our system call abstraction layer to cover every system call with a unified interface.

[8]The number of lines are measured with CLOC. See `http://cloc.sourceforge.net/`

7 Acknowledgements

KENSv2 is based on KENS (KAIST Educational Network System) developed by Junehwa Song and his students (`http://nclab.kaist.ac.kr/kens`). The original motivation and design of KENS has been a great inspiration. We are grateful to Yejin Park and Junhyun Shim for their help with writing. This work was supported by the Basic Science Research Program by the National Research Foundation of Korea (NRF) of MSIP (2014R1A2A1A01007580).

8 References

[1] James F Kurose and Keith W Ross. *Computer networking: a top-down approach featuring the Internet.* Pearson Education India, 2005.

[2] Dan Wendlandt, Martin Casado, Paul Tarjan, and Nick McKeown. The clack graphical router: visualizing network software. In *Proceedings of the 2006 ACM symposium on Software visualization*, pages 7–15. ACM, 2006.

[3] John DeHart, Fred Kuhns, Jyoti Parwatikar, Jonathan Turner, Charlie Wiseman, and Ken Wong. The open network laboratory. In *ACM SIGCSE Bulletin.* ACM, 2006.

[4] Martin Casado and Nick McKeown. The virtual network system. In *ACM SIGCSE Bulletin.* ACM, 2005.

[5] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *ACM SIGCOMM HotNets.* ACM, 2010.

[6] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[7] Ben Pfaff, Anthony Romano, and Godmar Back. The pintos instructional operating system kernel. In *ACM SIGCSE Bulletin*, volume 41, pages 453–457. ACM, 2009.

[8] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[9] PacketShader I/O Engine. `github.com/PacketShader/Packet-IO-Engine`.

[10] Intel DPDK (Data Plane Development Kit). `https://dpdk.org`.

[11] Marko Lackovic, Robert Inkret, and Miljenko Mikuc. An approach to education oriented tcp simulation. In *SoftCOM 2002: international conference on software, telecommunications and computer networks*, pages 181–185, 2002.

[12] Matt Mathis, John Heffner, and Raghu Reddy. Web100: extended tcp instrumentation for research, education and diagnosis. *ACM SIGCOMM Computer Communication Review*, 33(3):69–79, 2003.

[13] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX Annual Technical Conference*, pages 213–218, 2013.