# “Faster Packet Forwarding in a Scalable Ethernet Architecture”

Daniel Pall

Princeton University

January 7, 2008

# 1. Introduction

Enterprise networks are medium-to-large networks which connect a few hundred or a few thousand end hosts to each other and to other networks, such as the Internet. The networks of a typical college campus and a large business would qualify as examples of enterprise networks. Providing reliable and efficient networking service while minimizing the need for network administration is a critical need in today's world where availability is a priority but the time and artifice of human network administrators is often at a premium. Some protocol must be used to serve as the foundation for these networks. However, the most obvious choices for this task (Internet Protocol and Ethernet) are both inappropriate; Ethernet cannot scale to large networks, and IP scales well but is overly difficult to configure effectively. The method that is used in practice (connecting small Ethernet segments with a core of IP routers) is better than either standalone protocol, but it still leaves plenty of room for improvement. A new Scalable and Efficient Zero-configuration Enterprise network architecture (SEIZE) has been designed to improve upon the current hybrid scheme by incorporating the best aspects of IP and Ethernet into a single unified protocol. During the course of this semester, I re-engineered SEIZE to use a faster packet forwarder that runs inside each switch's operating system kernel. This replacement combined with various other code-level optimizations resulted in a switch's unidirectional throughput increasing from 300 Mbps to 916 Mbps. This enhancement in processing speed makes the architecture much more feasible for the traffic demands of high-bandwidth enterprise networks.

### 1.1. Ethernet

Ethernet is a data link layer protocol in the networking protocol stack. It is practically ubiquitous – it has been very popular in various incarnations since its creation in the 1970s by researchers at Xerox Palo Alto Research Center.[1] Ethernet is a shared-medium data transmission protocol in which multiple hosts are connected to a common bus and packetized messages from one host to another are broadcast over this shared bus. Unsurprisingly, this very basic scheme is extremely inefficient, in that two hosts who try to send data simultaneously cause a collision on the shared line and both packets are lost and must be retransmitted. For this reason, pure Ethernet is only usable on small segments of a few dozen communicating hosts at most, when the probability of collisions caused by simultaneous transmission is low.

Ethernet is much more efficient when such small segments are connected by simple switches which forward traffic between segments. The switches perform fairly minimal processing – the only appreciable work that they do is accumulate address information about the sources of the messages that they receive into a forwarding table. This table is then is used to reduce the size of the broadcast domain for messages later sent to these known sources: in this case, packets are only broadcast on the destination host's subnet, rather than the entire Ethernet network. One convenient aspect of the Ethernet

design is that the network as a whole is self-learning; that is, the network devices (switches and end-hosts) do not need to be explicitly configured with any information about the network topology or addressing in order to function. Rather, the protocol functions correctly due to this simple but clever knowledge accumulation system on the switches.

In fact, Ethernet's self-learning capabilities are sufficiently well-designed such that the network itself requires no explicit configuration of any kind. It is a plug-and-play protocol – a user can connect any network device which implements the Ethernet protocol to the appropriate physical medium on an Ethernet segment and the existing network will self-learn so that the host and the network devices can communicate with each other. This is an incredibly convenient system: adding a host to the network requires no intervention or setup on the part of a system administrator. Installing the network infrastructure (switches) is also quite simple, since a switch requires no explicit configuration – its forwarding tables will be automatically populated by the accumulation system once traffic starts flowing through it.

A final aspect of Ethernet which enables this simplicity and convenience is the use of MAC (Media Access Control) addresses for host identification. MAC addresses are 48-bit globally unique addresses burned into virtually every network-capable device at time of manufacture (Peterson 119). The global uniqueness of MAC addresses ensures that no two network devices in the world share the same address, meaning that it is certainly safe to depend on these addresses as unique identifiers in a local enterprise network. The permanence of each machine's address is extremely desirable in terms of positive and persistent identification of each host. This greatly simplifies various aspects of network management, such as mobility, troubleshooting, and long-lived access control policies, because there is no way that a host can ever change its identity.[2]

The downside of Ethernet is that this significant convenience comes at the price of a serious compromise to network efficiency. Ethernet inherently relies on small-scale broadcasting (within segments) and also makes heavy use of whole-network broadcasting for common services like Address Resolution Protocol (ARP) and Dynamic Host Configuration Protocol (DHCP) to enable host bootstrapping. These messages are generally intended for only a single recipient, but because the communicating network devices do not have a complete view of the network topology, the messages must be flooded to all connected devices in order to ensure delivery to the single targeted device. Obviously, this is inefficient in many ways: an inordinate amount of (finite) network bandwidth is used, and each network device wastes clock cycles while responding to interrupts for arriving packets which are not intended for them.

Ethernet also has the disadvantage of using spanning trees to construct forwarding paths, and the resulting paths are generally a few hops longer than they ought to be. That is, use of the Ethernet protocol results in packet delivery over a path

which is not the shortest in existence between the source and the destination. Obviously, this constitutes another waste of network bandwidth and CPU time. In summary, Ethernet is an extremely convenient protocol due to its zero-configuration nature. However, largely due to the mechanisms which enable this convenience, the protocol cannot scale to networks of more than a few hundred hosts (Peterson 119). Therefore, it is not a viable solution for an enterprise network consisting of thousands of communicating computers.

**1.2. Internet Protocol**

Internet Protocol (IP) implements the network layer of the protocol stack, sitting above some data link layer protocol (usually Ethernet). Like Ethernet, IP can be used to deliver packets containing arbitrary data between different end hosts connected to IP routers. IP is a richer and more complex protocol, though, and this complexity allows for greater performance and efficiency at the expense of higher amounts of human network administration. This greater efficiency is exemplified, for example, by the lessened amount of broadcasting of packets as compared to Ethernet, and the fact that the packet delivery path for IP between two arbitrary end hosts is designed to be the shortest path available for the given network topology.

This efficiency comes at a price of increased demand on network administrators. In contrast to Ethernet's design principle of zero-configuration, IP is fairly demanding on network administrators – for example, it is up to the administrators to manually assign a 32-bit IP address to identify each host or to designate some protocol (usually DHCP) to automate this assignment process. Even with the help of DHCP for individual host assignations, network administrators are still forced to manually divide the addresses allocated to the entire network among its various subnets. This is not a trivial process, especially in cases of high host mobility between subnets (for example, mobile hosts commonly move between subnets in a wireless local area network (LAN)). In these situations, administrators must allocate IP address blocks based on worst case load, which is often much more demanding than the average case. This is a necessary step, though, or else the network administrators risk denying service to a network newcomer due to a lack of IP addresses on his subnet. Ultimately, IP's demand for static allocation of hierarchical IP addresses among subnets places a heavy burden on network administrators and leads to some degree of addressing fragmentation and inefficiency (Kim 1). Ethernet's use of the MAC address permanently assigned to each network interface card (NIC) in the network for routing purposes is much more attractive from the standpoint of minimizing the amount of network administration.

The transient nature of IP addressing also complicates the implementation of persistent access control policies throughout different parts of the network. In other words, it is hard to keep track of a host whose identity is constantly

changing (which is a very real possibility, especially in high-mobility networks such as wireless LANs). IP is an efficient protocol, but it is not feasible for a network of thousands of users – the high degree of human administration required (even with the help of DHCP and other tools and protocols) is prohibitive, due to the high degree of work required to simply keep the network running and also due to the probability (which rises with the size of the network) that some human configuration error will result in an interruption of service.

**1.3. An IP/Ethernet Hybrid: Connecting Ethernet Segments with IP Routers**

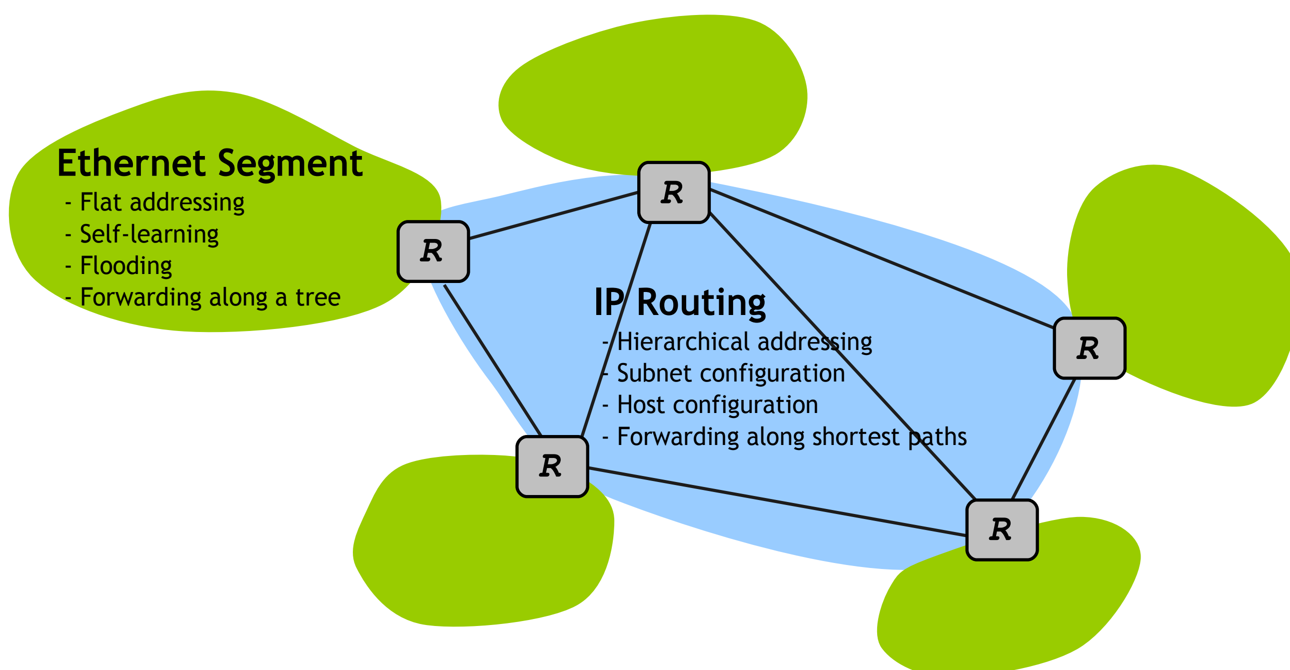


*Figure 1: Ethernet segments connected by IP routers*

As described above, neither IP nor Ethernet is a perfect solution as the single network protocol for an enterprise network. For this reason, today's enterprise networks use a combination of these two schemes. This hybrid implementation connects Ethernet-based subnets with IP routers. Then, simple broadcasting Ethernet is used for host-to-host traffic internal to each subnet, and IP routers are used for shortest path routing for communication between different subnets (Figure 1). This hybrid scheme is a reasonable plan, because Ethernet's inherent inefficiency is acceptable within small subnets of no more than a few dozen hosts, especially when this inefficiency allows for convenient plug-and-play behavior. Meanwhile, the use of IP on a macroscopic scale between Ethernet subnets ensures that inter-segment traffic is sent over shortest paths, which is an efficient use of network bandwidth.

It is by no means an ideal solution, though: host configuration required at the IP level can be automatically handled by DHCP, but coordinating this service between different subnets across the network is still a cumbersome task that a human network administrator must perform manually. This architecture also does not address IP's problem of addressing inefficiencies brought about by the need to split up the network's addresses among subnets whose user bases can grow and shrink dynamically as users move throughout the network. Fundamentally, this scheme is still undesirable because it uses transient IP addresses for the purposes of routing – there is no concept of a “permanent” identity of a particular host as that computer moves between different physical subnets of the network. This lack of positive, permanent identification complicates the challenge of maintaining consistent access control policies throughout the network. Certain protocols such as ARP and DHCP still make heavy use of broadcasting, but it is only over individual subnets rather than the entire network, so this is less wasteful than pure Ethernet in terms of bandwidth. The combination of the two protocols is preferable to a scheme of pure Ethernet or pure IP, but there is clearly room for improvement in terms of both efficiency and scalability.

## 2. SEIZE

SEIZE is a new network architecture designed to replace the imperfect Ethernet-IP hybrid currently used for enterprise networks. It was originally designed by Changhoon Kim and Jennifer Rexford at Princeton University. It can be viewed largely as an enhancement to Ethernet which mitigates the inefficiencies of that protocol while retaining the attractive simplicity of zero-configuration plug-and-play.

### 2.1. Principles

The goal of SEIZE is to combine the best properties of IP and Ethernet into a single protocol for use in an enterprise network. The key objectives of the protocol are efficiency and simplicity. Efficiency refers to minimizing the amount of wasted resources within the network; this can include everything from wasted clock cycles on individual hosts to link bandwidth wasted by sending a packet intended for a single recipient to multiple destinations. SEIZE also strives for simplicity in presenting a zero-configuration, plug-and-play scheme for the network – in other words, there should be no manual intercession by network administrators for regular events such as addition of end hosts to the network. Ultimately, SEIZE aims to combine the efficiency of IP with the convenience of Ethernet while remaining backwards-compatible with end hosts – a user should not need to install special software or drivers to make use of a SEIZE network.

### 2.2. Architecture and Mechanisms

SEIZE's design is largely dictated by its goals of efficiency and simplicity. For efficiency, SEIZE minimizes the extensive use of flooding and broadcasting which is utilized by Ethernet to enable communication with hosts whose network locations are unknown by the sender. Instead, SEIZE utilizes a clever scheme which reactively resolves the locations of unknown hosts using a distributed hash table as a directory system. This innovation obviates the need to flood packets to all neighbors to reach unknown destinations – an inefficient system that is found in pure Ethernet. Rather, SEIZE guarantees that unicast traffic is only actually sent to one host, even if this represents the first communication between those two hosts. Additionally, SEIZE eliminates the network-wide broadcasting that is used by Ethernet to bootstrap a new host into the network. Instead, a new host simply registers its MAC address with a special ingress switch and the ingress switch updates a hash ring that is shared between all switches in the network (Kim 4). Then, using the shared ring as a directory service, any switch in the network can locate the switch which services the newly-added host, and traffic can be sent to this ingress switch to be delivered directly to the host.

SEIZE's mechanisms to reduce broadcasting as compared to Ethernet helps SEIZE scale to huge networks much more effectively than Ethernet. At the same time, SEIZE retains Ethernet's convenient simplicity by using permanent and unchanging MAC addresses for routing. Network administrators can choose to assign an IP address to each host for application-level compatibility and reachability from external networks, but it is not necessary for routing purposes, and they need not be assigned in the hierarchical manner which complicates the hybrid IP/Ethernet implementation used in today's enterprise networks. The use of MAC addresses for identification also helps network administrators maintain consistent access control policies on a per-host basis throughout the entire network.

SEIZE retains the shortest-path routing of IP by using Open Shortest Path First (OSPF) routing between switches. This represents another design choice which borrows from IP in order to optimize efficiency. It represents an improvement over Ethernet, which is generally configured to forward packets across a single spanning tree which touches each switch in the network and does not contain any loops. Spanning trees are used for simplicity and to prevent packets from being endlessly duplicated through forwarding loops in the network, but they generally result in overly-long paths (e.g., too many hops) between any two switches in the tree. Shortest-path routing is a much more efficient scheme, in that it minimizes the number of switches involved in sending a given packet between a source and a destination, and it conserves the network bandwidth used for this transfer.

A final design goal for SEIZE was to avoid necessitating any changes to the end-host – in other words, no new programs or protocols should be needed on a computer that wishes to join the network. To this effect, SEIZE runs within

the existing network protocol stack and can be easily deployed on networks that support any Ethernet standard, such as IEEE 802.3 Ethernet and IEEE 802.11 Wireless LAN (Kim 2).

**2.3. Implementation**

SEIZE was first implemented by Changhoon Kim as a software router built from several different open source networking components and proprietary code written as extensions to these components. The entire network (consisting of a few hosts and special SEIZE switches connecting them) runs on general purpose desktop computer hardware: the hosts are completely unmodified and run the normal TCP/IP networking protocol stack, and the switches are configured with the routing tools depicted in Figure 2.

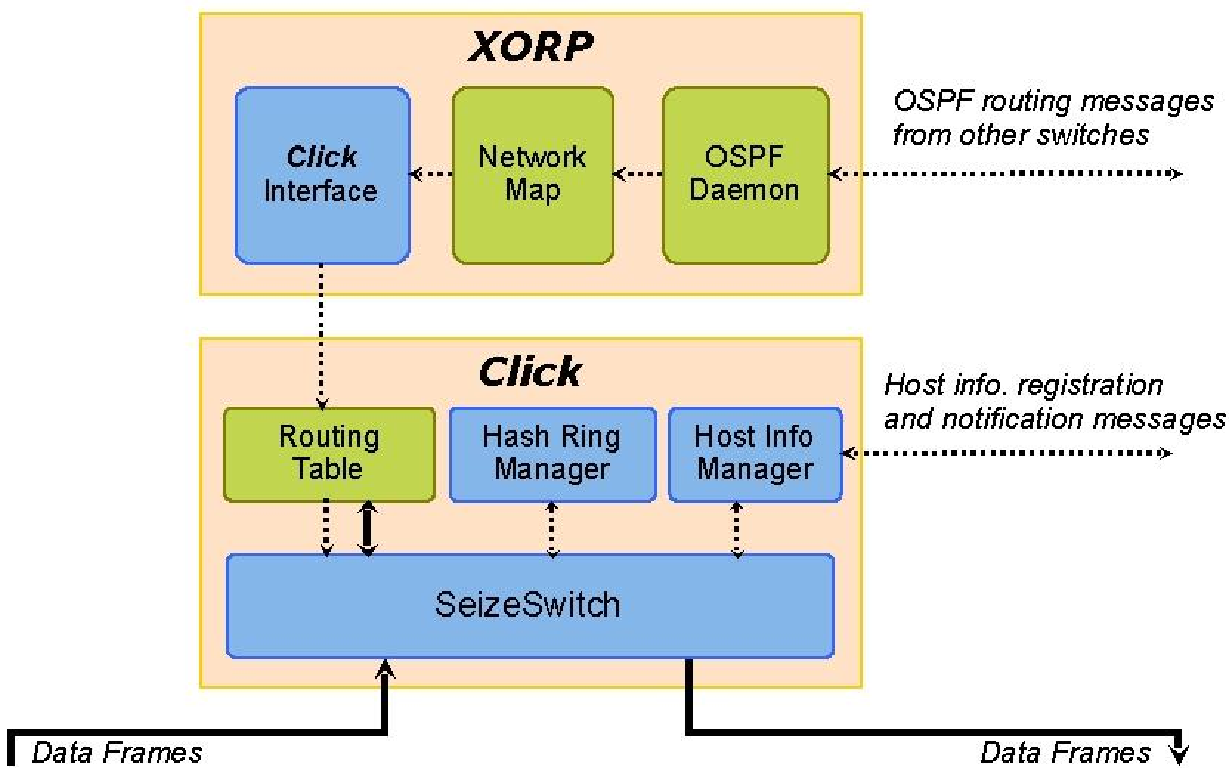


*Figure 2: A SEIZE switch consists of a XORP router which uses a Click packet forwarding engine.*

**2.3.1. XORP**

XORP (eXtensible Open Router Platform) is an open source software router that allows developers to utilize existing routing protocols or develop and test new algorithms for networking research.[3] It is implemented in C++ and is

focused on providing a stable and complete routing platform that is simultaneously open and accessible for developers who wish to change any aspect of its behavior. In this sense, it is very useful as a research tool for prototyping new networking technology, as it allows researchers to utilize regular desktop computers as customized routers, without necessitating specialized (and often costly) dedicated routing hardware.

XORP coordinates the higher-level aspects of a router, such as implementing routing algorithms (including common standards such as OSPF and RIP, or custom policies written by the developer) while leaving the actual low-level mechanics of forwarding packets with the switch's network interface cards to some other piece of software. This software is usually either the internal packet forwarding engine of the underlying operating system or some other hardware- or software-based forwarder. For example, XORP can use the Linux kernel itself as its packet forwarding component, or it can use a third-party software package such as Click.

The main purpose of XORP within the current SEIZE implementation is to execute Open Shortest Path First (OSPF) link-state routing between the different switches within the network. OSPF is a shortest-path routing algorithm commonly used with IP routers, and the benefit here is the same – no packets are sent over overly-long paths, thus conserving clock cycles of the networks' switches and bandwidth on the networks' links. For our project, we didn't write any extensions to XORP; rather, we simply used the basic XORP code with a customized Click forwarding engine.

**2.3.2. Click**

Click is an open source modular router project primarily developed at MIT and UCLA.[4] Like XORP, it is developed in C++ and is designed to be extensible and customizable by any developer who wishes to deploy his own code on the system. Click differs from XORP in that XORP focuses on implementing high-level routing protocols, but Click is used (by XORP or as a standalone module) to actually forward data at the packet level according to one of these routing policies. A Click router is composed of one or more different "elements", which are pieced together in a directed graph which emulates packet flow through the router system. There are dozens of default elements included with the standard Click distribution; some examples include modules traffic-control algorithms, priority traffic schedulers, round robin schedulers, multiplexers and demultiplexers, and so on. By arranging these elements in different configurations, Click is flexible enough to simulate many different types of network devices, such as IP routers, Ethernet switches, and innumerable variations of each of these devices with various customizations and behaviors.[5] Click developers also have the option of adding custom elements to the Click library to create specialized routing behaviors in networking projects. This is exactly how SEIZE is implemented – as a custom Click element called SeizeSwitch which is combined with a standard set of Click

elements inside a standard XORP software router.

The Click package consists of two distinct components: a user-space engine and a kernel module, each of which serve to interpret a user's routing configuration into an actual executing program which forwards packets according to the structure of the configuration and the definitions of the Click elements that it utilizes. The packet forwarding behavior generated by each engine is identical (that is, given the same router configuration, packets are routed in exactly the same manner by both the user-space program and the kernel module). The sole difference is that the user-space engine runs on a computer as a normal, unprivileged program, while the kernel module is dynamically linked into the kernel and therefore runs inside the operating system itself.

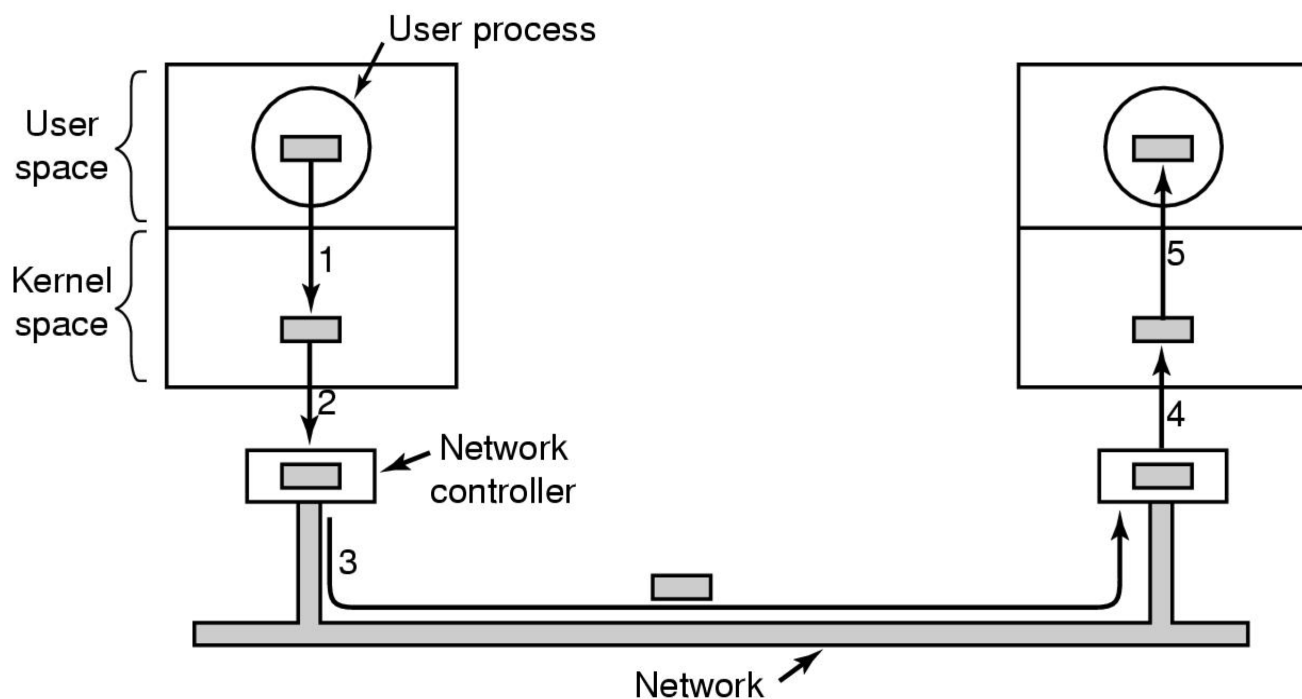


*Figure 3: A packet is copied many times between different parts of memory when a user-space packet forwarder is used. Image from Tanenbaum (2001) page 296.*

The kernel module is designed to forward packets at a significantly higher rate than the user-space program. This is due to the method that an operating system uses to write a data packet to the attached network (Figure 3). A user-level program has the packet in memory (RAM) and performs some system call, which traps to the kernel and copies the packet to some reserved buffer in the kernel's memory (also in RAM). The kernel then copies the packet over the system bus to some dedicated fast memory buffer located directly on the network interface card, which finally sends the bytes out onto the physical wire of the network.[6] This packet gets copied three separate times just to make it out onto the network once (that

is, we are assuming that the transmission occurs successfully and there are no retransmissions required). For a switch to actually forward a packet, it must be copied from the network card up to the forwarding engine in user-space (three copies) where it is examined and a routing decision is made – then the packet must be copied three more times to get back out to the network for further delivery.

A packet forwarder running in the kernel helps reduce forwarding latency by eliminating the copy operations between user memory and kernel memory. Moving the Click engine to the kernel reduces the number of packet copy operations necessary to forward a single packet from six to four. In today's computing environment where fast memory is often the limiting factor in computation speed, the opportunity to reduce dependence on system memory by reducing the number of times that the same data is copied cannot be ignored as an opportunity for optimization. Considering that a typical switch in a congested enterprise network might receive a few hundred thousand packets every second, it is clear that the use of a kernel-level packet forwarding engine might dramatically improve the performance of each switch. Therefore, my primary goal this semester was to replace the user-space Click engine with the kernel module, in hopes of improving the switch's packet throughput and making SEIZE more appropriate for a large-scale enterprise deployment.

**2.4. Evaluation Settings**

To get a sense of how SEIZE will perform in a real world situation, it is necessary to deploy the system on a physical network of computers and benchmark its performance in terms of data throughput.

**2.4.1. Emulab**

Emulab is a time-shared network testbed hosted by the University of Utah and usable free of charge for university researchers around the world. It is an extremely flexible and powerful testing resource, allowing users to essentially lease an arbitrary number of computers for a few hours, connect them to construct virtually any sort of network topology, and then test any experiment on these connected nodes. It was an ideal solution for testing SEIZE: it is simple to create switch/host topologies as simple or complex as desired and monitor traffic speeds and levels throughout the entire network.

A particularly useful feature of Emulab for my work this semester was the option which allows users to create and deploy customized operating systems on test nodes. Researchers can customize a particular computer with a modified operating system or a new suite of software and then run an Emulab process which essentially collects all of the information about that machine into a specially-formatted disk image. Then, this disk image can be automatically loaded onto clean machines in the future, without any further configuration necessary. This obviates the need to configure each machine individually for experiments. I used this feature of Emulab to create a customized version of the Linux operating system

kernel which was compatible with the Click kernel module necessary for high-speed packet forwarding.

**2.4.2. Testing**

The actual network architecture of SEIZE is analogous to that of Ethernet. Hosts are connected to switches via Ethernet cables, and the switches are in turn connected to each other to form a wired local area network. For my tests in Emulab, the topology was specified in a text file that used a simple configuration language to declare the existence of different nodes and connect them in a point-to-point manner. Below are two examples of simple network topologies that I used during the semester (Figure 4).

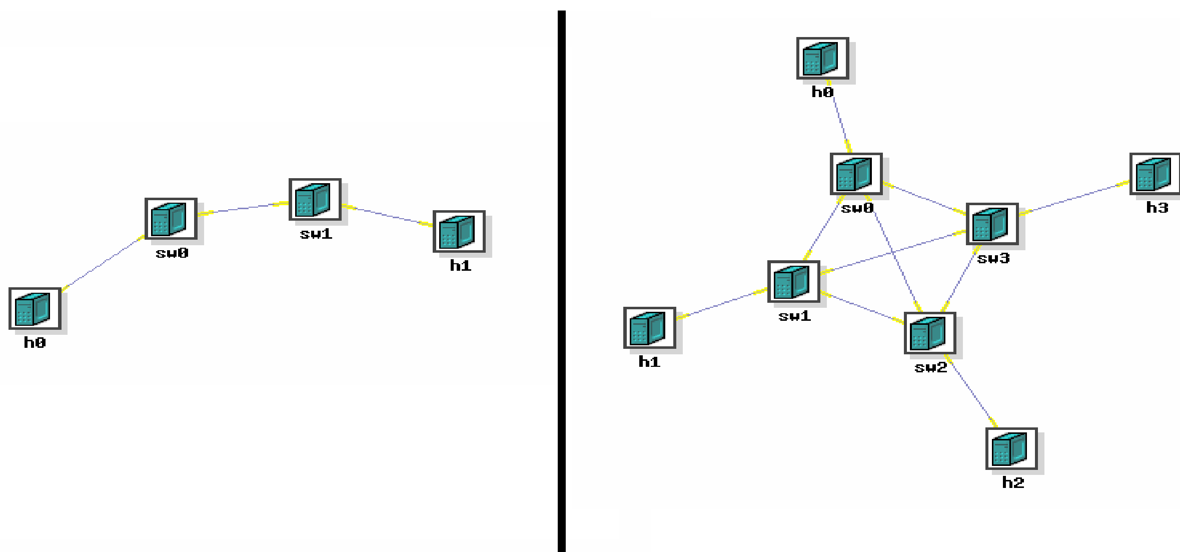


*Figure 4: Two separate network topologies. Nodes named with an "h" indicate hosts (end-users), and nodes named with "sw" indicate SEIZE switches. All links have a capacity of 1 Gbps.*

An Emulab experiment is “swapped in” and mapped to an appropriate set of connected nodes as defined by the experiment configuration file. Once each machine's boot sequence is complete, a sequence of setup scripts is run in order to prepare all hosts and switches for the experiment. At this point, the XORP router is invoked on each switch, which in turn instantiates the Click forwarding engine, and hosts are then able to send data to each other with any program that utilizes the TCP/IP networking stack. For testing purposes, this includes tools such as ping, iperf, and tcpreplay.[7, 8]

Each host is identified by its network adapter's 48-bit MAC address, just as in Ethernet. IP addresses are assigned to each host for convenience and usability during the testing process. The hosts are configured by the initial setup script so

that they are on the same subnet. As a result, each host tries to resolve the addresses of other hosts with local ARP requests, rather than sending packets to an external gateway, like a switch or router. These ARP requests initiate the SEIZE routing mechanisms, and SEIZE takes care of packet forwarding (replacing Ethernet) from this point onward.

# 3. SEIZE packet forwarding in the Linux kernel

The original SEIZE prototype was an effective demonstration of the potential of the architecture – each switch routed packets correctly and demonstrated a reasonably high packet throughput of 300 Mbps over a gigabit network. However, this throughput is insufficient to meet the traffic demands of a large enterprise setting. Furthermore, the prototype's data bandwidth of only 300 Mbps on a network whose maximum bandwidth is 1 Gbps was indicative of an inefficient use of network resources. For my work this semester, I replaced the user-level Click engine inside each switch with the Click kernel module. My hope was that shifting a significant portion of the SEIZE architecture into the kernel would result in a speedup that would enable the architecture's use in larger networks with higher bandwidth requirements.

### 3.1. Porting SEIZE from FreeBSD to Linux

My overall goal this semester was to replace the user-level Click forwarding engine inside the existing SEIZE switch prototype with an equivalent kernel module. However, the original SEIZE prototype ran on the FreeBSD operating system, and the kernel-level Click module for FreeBSD is somewhat embryonic at this point in time. Support for kernel-mode packet forwarding in Linux is much more mature and more widely utilized than the analogous FreeBSD package. For this reason, my advisors and I decided that porting the existing SEIZE architecture from FreeBSD to Linux was a critical step in improving SEIZE's performance with kernel-level Click. It also seemed like a generally good idea to move the software to Linux, in light of Linux's more sizable user base (both generally and particularly with respect to Click usage). This operating system port was not trivial, though: some of the challenges that I had to address are described below.

#### 3.1.1. Recompiling Click and XORP

The existing Click executables were compiled for FreeBSD, so I had to replace them with Linux executables. I decided to upgrade from version 1.5 of the Click software to the recently released version 1.6 to take advantage of various optimizations and bug fixes completed in the new version. In addition to generally compiling a Linux version of the user-space packet forwarder, I had to take special steps to compile and use the Linux kernel module, including a patch of the Linux kernel with a special Click modification package. This patch mostly fixes a few header files so that the C++ compiler accepts them, adds certain additional network functionality, and redefines “struct device” within the kernel to make it

compatible with Click.[9]

The patch itself was fairly minimal, but for its functionality to be recognized, the kernel had to be recompiled. This was a fairly complex process, due largely to the fact that my test network was located at a remote location at the University of Utah's Emulab network testbed. I eventually had to use a TTY program to gain direct serial-line access to the booting machine to enable the newly-installed modified kernel, as normal SSH access was only available after boot time. I used the disk image utility provided by the Emulab testbed to create a reusable image of the newly-created operating system with kernel-Click compatibility compiled into the kernel.[10] The final result was essentially a normal copy of Fedora Core 6 running the Linux 2.6.19.2 kernel with modified headers to allow kernel-Click to be dynamically loaded into the kernel at runtime. This operating system image has proven extremely useful during later testing, as it has alleviated the need to engage in the time-consuming process of repeatedly recompiling and reinstalling Click-enabled kernels on different nodes.

Compiling XORP for Linux was a much easier task in comparison. Unlike the kernel recompilation required for Click, I didn't need to take any special steps for XORP – a time-consuming but simple compilation at user level on the Linux platform was sufficient. I decided to upgrade from version 1.3 of the software to version 1.4 for the same reasons that I upgraded Click – the newer version was recommended by the developers as being more stable than version 1.3.

**3.1.2. Rewriting Configuration Scripts**

SEIZE uses a set of shell and AWK scripts to set up each machine for network communication. The scripts generally perform the roles of setting up each machine's interfaces and configuring the XORP router and Click forwarding engine in each switch. I had to rewrite significant portions of these scripts to accommodate the move from FreeBSD to Linux, mainly because these two operating systems utilize different directories for common networking resources and different commands to access this information. In these cases, the modifications I made were mostly changes of syntax – the functionality was retained in the FreeBSD to Linux switch, but the language necessary to enable that behavior underwent significant changes.

- **custom_init**: custom_init is a shell script which is run upon startup on each switch and host. It executes all of its commands as the root user, as this is necessary to perform most meaningful networking tasks in Unix-like systems. One example of a task performed by custom_init is disabling each machine's capability to route packets using its own internal kernel code (we want to route packets with the kernel, but using different code to do so – namely, the Click software). custom_init also creates a group called "xorp" for use by the XORP router processes, and it creates a mount point at /click which is eventually used to mount a virtual filesystem corresponding to the Click

router running on the switch. This virtual filesystem is analogous to the /proc filesystem found in Unix variants – it is written by the router itself and can be read by other users to get a sense of the router's performance at as it runs. The configuration can also be rewritten while the router runs, which enables dynamic reconfiguration of the router's behavior. Finally, custom_init sets some environment variables in anticipation of calling init_config if the current machine is defined as a switch and h_init if the machine is defined as a host.

- **init_config**: init_config is an AWK script called by custom_init that is only run on SEIZE switches. The input to the script is a description of the emulation network topology and a generator for the Click forwarding engine abstraction. The script generates a XORP router configuration and a configuration-generating script for the Click forwarding engine used by the XORP router. Essentially, this script inserts the address information about the machine and its interfaces (sent from the custom_init script) into a template for each of these configuration files.
- **h_init**: h_init is a shell script called by custom_init that is only run on hosts and not switches. It sets the host's maximum transmission unit (MTU) to 1466 bytes. This value is used because SEIZE switches encapsulate all packets inside an extra Ethernet frame and IP packet, which combine to require 34 bytes of header space, so the total maximum packet size in a SEIZE-enabled network is 1466 + 34 = 1500 bytes, which is the maximum MTU supported by an Ethernet-based network. This script also reconfigures its machine's interfaces to be compatible with the experiment.

### 3.2. Replacing user-level Click with kernel-level Click

Once the move from FreeBSD to Linux was complete, I had to take steps to add the kernel-mode packet forwarder to the SEIZE architecture to enable higher switch performance. A related secondary goal was to compile the Linux user-level Click module for comparison against both the FreeBSD user-level module and the Linux kernel module.

#### 3.2.1. Modifying Click configuration generator script and XORP configuration file

One problem that arose during the upgrade from user-level Click to the kernel module involved a difference in the way that the two forwarding engines handled control-plane packets for the OSPF routing protocol. SEIZE's network traffic can be divided into two categories: data-plane traffic (arbitrary messages between hosts, passing through one or more switches during the transit) and control-plane traffic (messages between switches which are used to update each switch's OSPF routing tables). For the system to function properly, Click must receive copies of all packets and forward them to other machines as appropriate, and the XORP process (which handles all OSPF routing) must receive copies of the control-plane packets only. When user-level Click was used as the switch's forwarding engine, each switch was configured so that

control-plane packets were echoed automatically to XORP (another user-level process). When the Click kernel module was substituted, however, it "intercepted" all packets – control-plane and data-plane – and did not forward the control-plane OSPF messages to XORP. To work around this problem, it was necessary to explicitly configure the Click engine to inspect the IP_PROTOCOL field of every incoming IP packet header. A value of 0x59 at this position indicates that the packet is an OSPF control message. The Click configuration was rewritten to include this check and send all OSPF control messages to the XORP process in user memory.

One relatively minor task that I had to perform was to update the XORP router configuration file so that it invokes kernel-mode Click instead of the user-space version. I added kernel-mode Click as the option enabled by default, because its performance is demonstrably superior than the user-level equivalent. For simplicity, future users can revert to user-level Click if desired simply by changing two lines of the file to enable user Click and disable kernel Click.

**3.2.2. Removing user-level libraries from custom SEIZE code**

A significant challenge in my effort to replace user-space Click with kernel-mode Click was the elimination of all user-space libraries from the Click code of the SeizeSwitch. It is illegal to utilize user-space library functions in code that runs in the kernel (the result is link-time and run-time errors), so I had to enact several workaround solutions which achieved the same results as the original user-space libraries without actually resorting to any user-level code.

**MD5 hashing libraries**

MD5 is a secure hash commonly used for encryption purposes and to fingerprint files for equivalence testing. It is considered a fairly strong hash and is often used for security purposes; its output space is a sizable 128 bits and collisions involving different hash arguments are minimal.[11] The SEIZE prototype used MD5 hashing with a switch's 48-bit MAC address as its argument. The result of this hash is utilized as a unique identifier – a fingerprint for that particular switch – which determines the switch's position on the distributed hash ring. Then, each SEIZE switch in the LAN can use this ring (which acts as a kind of directory service) to find the switch directly connected to a destination host whose location is unresolved. Finally, the lookup process is completed when this discovered switch reports the MAC address of the unknown host to the original requesting switch. This clever SEIZE lookup mechanism helps eliminate one of Ethernet's most glaring faults – the flooding of packets throughout the network when the destination host's address is unrecognized.

MD5's nature as a fairly powerful and cryptographically secure hash was not strictly necessary for its usage in SEIZE. Rather, the prototype basically needed a method of mapping a 48-bit MAC address onto a larger (128 bit) result space to be used as a directory service that could be utilized by each switch. In other words, SEIZE (using user-level Click

or kernel Click) requires nothing more than a decently random hash function which will consistently map MAC addresses onto the shared ring – not necessarily MD5. In fact, SEIZE's creators note that the use of MD5 is essentially overkill in this situation and that SEIZE would benefit from its replacement with a more lightweight hash which uses fewer CPU cycles during each invocation.[12]

The prototype used the MD5 package defined in md5.h and implemented in the lmd library in FreeBSD. Its inclusion in the Linux version of SEIZE with user-level Click (an intermediate step in my port) required the openssl/md5.h header file and the lssl library, since Linux uses the MD5 implementation used by the OpenSSL program. When I tried to use the same function for kernel-level Click in Linux, the kernel rejected the compiled module, since the OpenSSL library code was user-level and therefore inappropriate for the kernel. I was forced to examine alternate strategies for using this hash or a similar hash within the confines of the kernel.

On the advice of my advisor, I decided to move away from the secure MD5 hash and use a simpler, light-weight hash implementation. After examining some open-source hash functions, we settled on the Jenkins one-bit-at-a-time hash, which displays low collision rates and an effective avalanche effect, meaning that, on average, flipping a single bit in the hash input will change about half of the output bits.[13] Despite the fact that its implementation is extremely short (less than twenty lines in total), its effectiveness as a hash function is not substantially lessened compared to a more industrial-strength hash such as MD5. I was able to simply copy this function in its entirety as an inlined function in the SeizeSwitch implementation file. After adjusting the sections of the SeizeSwitch code that called these hash functions to reflect the new structure of the hash, the code was updated and ready for usage in the Click kernel module. There were no problems with compiling or linking against illegal libraries, since I eliminated outside linking completely by inlining the hash function.

**atoi( ): Array to Integer function**

atoi( ) is a widely-used library function utilized by the SeizeSwitch element to convert a number stored digit-by-digit in an array into an integer value that is more easily usable for common arithmetic and logical operations. For example, atoi( ) would return the value 398 if it was passed as an argument an integer array with elements [3] [9] [8]. atoi( ) is defined on most systems as a standard C library function in stdlib.h. However, the library in which the function is actually defined runs at the user level (just as with the MD5 implementation), so it is unavailable for the Click kernel module.

Rather than searching for a kernel-level implementation of atoi( ), I just decided to implement it within the SeizeSwitch implementation file, just as I had previously done with the Jenkins hash function. atoi( ) is a very simple function, and I was able to find a short and clean implementation of it on a code examples website. Again, this method of

inlining a function that was previously contained in a user-level library was a perfectly effective replacement policy.

**STL sort function**

The SeizeSwitch Click module utilizes the sort( ) function of the C++ Standard Template Library (STL) during the switch registration process (e.g., when a switch is added to the distributed hash ring). This is a very widely-utilized sorting method which encodes an optimized quicksort algorithm. sort( ) can be used to sort an array filled with virtually any type of C++ object using any comparison function which is passed to sort( ) via a function pointer. Once again, my efforts to compile kernel-mode Click with this function in place were unsuccessful. The code for sort( ) is significantly longer and more complicated than that of the atoi( ) function or the Jenkins hash, though, so I looked for an alternative to accommodate general-purpose sorting rather than simply inlining a copy of the code within the SeizeSwitch module.

I consulted the Click mailing list for guidance, and some Click users pointed me to the click_qsort( ) function defined in lib/glue.cc within the Click distribution. Essentially, click_qsort( ) is a re-implementation of STL sort( ) which is usable in Click projects both at the user-level and within the kernel. click_qsort( ) uses a different parameter structure than STL sort( ), but otherwise, the functionality provided is identical. Therefore, replacing STL sort( ) within the SeizeSwitch code was as simple as shifting around some of the arguments and calling click_qsort( ) instead of sort( ).

### 3.2.3. Modifying the basic Click code for compatibility with SEIZE

Adding the new SeizeSwitch element to the Click code base required that about 10 other Click source files be changed to reflect the new semantics of a SEIZE switch. These changes had already been made for SEIZE with user-level Click, so in some sense, there was already an existing patch for the Click 1.5 code base. But the job was complicated by the fact that I upgraded from Click 1.5 to Click 1.6 during the course of the semester, and patching the code for kernel-mode Click raised more challenges than did the original patch for user-level Click.

# 4. Results

The upgrades to SEIZE that I implemented this semester (namely, the usage of the Click kernel module and the lightweight Jenkins hash function) resulted in a significant increase in switch throughput compared to the performance of the previous SEIZE prototype, which utilized the user-level Click engine and the slower MD5 hash.

## 4.1. Hardware Configuration

Each switch and host in the experiment used identical hardware. Emulab's pc3000 nodes are fairly powerful workstations equipped with a 3.0 GHz Intel Xeon processor, 2 GB of DDR2 RAM, and a high-speed 64-bit / 100 MHz PCI-

X bus for communication between the CPU and the network card. Each machine also has several Intel E1000 Gigabit Ethernet network interface cards.[14] I used these network cards to connect switches and hosts with 1 Gbps links to create a network whose maximum one-way throughput between any two nodes was 1 Gbps. I used the two-switch, two-host linear topology shown earlier (Figure 4) to make experimental measurements, because most of Emulab's pc3000 machines have too few gigabit networking cards to support a full gigabit network using the four-switch, four-host topology. My use of a very simple network topology for these tests is reasonable because my tests focused purely on the maximum data-plane throughput of each SEIZE switch, rather than an aspect of its behavior such as control-plane scalability or response time to various network dynamics, measurements of which would require a more complex topology to deliver meaningful results. The ability to create a more complex, realistic topology would have been helpful for the SEIZE versus Ethernet test, but unfortunately Emulab simply did not have the necessary hardware necessary for a larger emulation.

**4.2. Testing switch throughput with iperf**

iperf is a networking utility maintained by the University of Illinois which helps measure maximum TCP bandwidth between two different hosts on a network. It is very convenient for the purposes of testing the maximum throughput of a SEIZE switch, because it essentially measures an arbitrary point-to-point bandwidth by sending massive amounts of data between the two machines (and thus through SEIZE switches) at a variable rate until it finds the highest such rate that is supported between the two hosts. The use of TCP ensures reliable delivery of messages, so the bandwidth results indicate the amount of throughput for SEIZE switches without any packet loss. I used a 256 KB window size and tested bandwidth over 10 second transmission periods after the TCP connection was established. For the simple linear topology, we ran iperf on the end hosts h0 and h1, so that each packet passes through two SEIZE switches sw0 and sw1.

**4.3. Comparing different versions of SEIZE: FreeBSD vs. Linux and User-Click vs. Kernel-Click**

The original SEIZE prototype (which runs on FreeBSD, uses MD5 hashing, and employs a user-level Click process as its packet forwarder inside XORP) yields a unidirectional throughput between h0 and h1 of 300 Mbps. A bidirectional test (e.g., a TCP stream between h0 and h1, and an identical, simultaneous stream sent from h1 to h0) resulted in a total throughput of 420 Mbps. These results are reasonable: the bidirectional test utilizes the network's potential bandwidth more effectively than the unidirectional test, but the improvement falls short of a full throughput doubling because the switches must share hardware resources such as CPU time and control of the I/O bus and the network interface card between two TCP streams rather than remaining dedicated to a single data stream.

Under the same conditions in Linux, SEIZE with user-level Click achieved much more impressive throughput

numbers: 495 Mbps for the unidirectional test and 605 Mbps for the bidirectional test. The reason for Linux's superiority is not immediately clear, but I hypothesize that it is due to the underlying TCP implementation of each operating system – Linux's implementation is probably more aggressive in trying to attain higher data rates than that of FreeBSD. This is likely due to the fact that the SEIZE prototype used an older version of FreeBSD (version 4.10-stable, released in May 2004) and the new platform used a comparatively newer release of the Linux kernel (version 2.6.19.2, released in January 2007).[15,16] This improvement gained by moving user-level Click from FreeBSD to a recent version of Linux was a pleasant surprise.

With user-level Click on FreeBSD established as the slowest implementation, a purely Linux-based comparison of SEIZE with user-level Click versus SEIZE running in the kernel was necessary. I tested the bandwidth with various packet sizes by limiting the MTU of each host machine and then running an iperf TCP test. The results were very satisfying: in terms of data throughput, SEIZE with kernel-mode Click vastly outperformed its user-level equivalent at every tested packet size (Figure A1). The bidirectional tests provided even stronger support in favor of permanently moving Click from user-level to the kernel – the gap in throughput between the two Linux versions of SEIZE was even greater when the network was more heavily loaded with two streams of traffic (Figure A2). This bodes well for future deployments of SEIZE in networks with greater numbers of hosts – it indicates that SEIZE with kernel Click is faster and more scalable than the version which uses user-level Click.

**4.4. SEIZE Bottlenecks**

We can gain further insight into the performance differences between user-level Click and kernel-mode Click by examining the same bandwidth numbers expressed as packets per second sent between hosts. The unidirectional and bidirectional packet throughput results are given in Figures A3 and A4. The plots are extremely revealing: user Click has a maximum packet throughput of about 42 kilopackets per second (Kpps), independent of both packet size and the number of streams passing through a particular switch. This firm limit indicates that there is some bottleneck in user-level Click's packet forwarding pipeline. The bottleneck might be the result of some hardware limit (such as CPU time, physical memory, I/O bus availability, or I/O interrupt rate) or a particularly time-consuming software operation within the SEIZE code (such as a slow and heavily-utilized hash function).

In Linux, user-level Click calls the same lightweight Jenkins hash as kernel Click and each Click engine runs on the same hardware. Therefore, this limit on each switch's packet forwarding rate in user-space Click (which was not reflected in the throughput results for kernel-mode Click) is probably not related to one of these factors. Rather, the user-level Click limit of 42 Kpps is possibly related to a limit on the speed of the copy operations that move packet data from the

kernel to the user-level Click process and back (as illustrated previously in Figure 3 in Section 2.3.2.). This operation (which is necessary for user-level Click but not performed in the kernel module) stands as the main functional difference between the two engines. Therefore, it seems reasonable that major differences in behavior between the two packet forwarders (such as this limit on packet forwarding rate) might be attributed to the extra copies between user memory and kernel memory present in the user-level Click program. The bottleneck also might be related to differences in the ways that I/O interrupts are generated and handled in kernel space and in user memory in Linux. Since the operations of sending and receiving single packets from the network card each generally require the use of interrupts (communication between the network card and the CPU), it seems reasonable that limits on interrupt generation or handling for user programs (not imposed on kernel modules) might result in this type of packets-per-second throughput limit.

To further examine this packet throughput limit experienced by SEIZE on user-level Click, I plotted unidirectional and bidirectional packet throughput results for simple Ethernet bridging (Figure A5). My use of Ethernet for this test has no particular motivation; rather, I simply wanted to test a different router configuration in user-level Click in order to determine if the limit on packets forwarded per second was unique to SEIZE or whether it might be observed for another configuration. The plot shows that a similar limit on packet throughput (again, independent of packet size or number of data streams) exists for a user-Click Ethernet switch, albeit at a different limit (about 68 Kpps, rather than SEIZE's limit of 42 Kpps). This result lends support to the possibility that the user-level SEIZE's packet forwarding rate limit is not the result of any SEIZE-specific software bottleneck, but rather that the limit is simply a result of the system architecture of user-level Click, which appears to be very vulnerable to a data-copying bottleneck.

Based on this packet forwarding rate limit experienced by user-level Click and its inferior data throughput as compared to the kernel module, it is inevitable that kernel-mode Click will serve as the core of SEIZE in future versions of the architecture. Therefore, a careful examination of the kernel module's performance and possible performance bottlenecks is in order. Currently, data throughput does not appear to be bound currently by CPU speed, I/O bus bandwidth, or availability of physical memory. Examination of the switch's run-time CPU statistics (using the Unix “top” command) shows that CPU utilization during a long-lived bidirectional TCP test is about 30 percent for large 1466-byte packets and only 11 percent for smaller 128-byte packets. This is a reasonably heavy load on the CPU, but the switches are by no means CPU-bound at this point, and a move to multi-core processing could further lessen the load on the CPU. Our test hardware used an extremely fast PCI-X bus for communication between the CPU and the network card on each switch – the total throughput for the bus was at least 6.4 Gbps, so this bus speed was not the limiting factor either, given our test network

network throughputs of less than 2 Gbps.[17] Physical memory can also be eliminated as a bottleneck, as it was virtually untaxed by the tests (less than 1 percent usage, according to "top"). Finally, our method of generating packets on hosts (using the user-level iperf program) does not presently appear to be a bottleneck either; still, as SEIZE matures and strives for higher speeds in the future, its testing mechanism should evolve to keep pace, perhaps by using a kernel module like pktgen for higher-speed packet generation.[18]

Since these components of the system seem unlikely to be the current bottlenecks for the kernel module, it seems possible that the currently bottleneck is the network resources of our testbed – specifically, the gigabit links between machines and the gigabit network cards that connect each switch and host to the network. The current results point to network utilization of over 91 percent (just short of total network utilization), which means that SEIZE might very possibly display improved performance if deployed on a higher-speed network. A careful examination of the data throughput and packet throughput results for kernel-mode SEIZE can help explain why link bandwidth is probably the current bottleneck.

For kernel-mode Click, packet throughput seems to be limited by two factors. The best packet forwarding rate (over 200 Kpps) is attained for packet sizes between 384 bytes and 768 bytes. Here, the bottleneck is probably related to hardware – it is possible that the interrupt rate for kernel module code is being taken to its maximum limit. For larger packet sizes between 896 bytes and 1466 bytes, the packets-per-second throughput graph becomes linear with a negative slope; this is indicative of a connection limited by link bandwidth. This hypothesis is strengthened by the observation that the data throughput at these packet sizes in Mbps (Figures A1 and A2) is basically constant at about 900 Mbps. Notably, though, within this large-packet range, the data throughput does rise slightly but steadily with packet size. This is due to the fact that the per-packet overhead becomes fractionally lower as packet size increases – that is, packet headers remain constant in size, while the data payload of each packet increases by 128 bytes at each test increment. This increasing data-to-header ratio results in higher data throughput at the highest packet sizes.

**4.5. Comparing SEIZE and Ethernet bridging**

For our test topology and configuration, SEIZE performed slightly worse than simple Ethernet bridging for both unidirectional and bidirectional streams as shown in Figures A6 and A7. At first gloss, this is a somewhat troubling result, as SEIZE is designed to offer higher aggregate throughputs than a simple Ethernet solution in a large enterprise network. However, it must be noted that our test environment hardly qualified as an enterprise network: it involved only two static hosts and one or two TCP streams, while a real network would encompass hundreds or thousands of hosts and likely thousands or tens of thousands of connections. This difference is critical in analyzing total throughput numbers, because

SEIZE gains a huge advantage over Ethernet by cutting down on broadcasting and flooding of control-plane packets – something that happens very often in networks with real-world network dynamics, but not at all in our static simulation.

SEIZE enjoys another significant advantage over Ethernet by using shortest paths for routing. This is a much more efficient use of link bandwidth and switch processing time than the single spanning tree used for packet forwarding by Ethernet bridging, because average host traffic passes through fewer switches and links en route to its destination, resulting a more efficient use of network resources and less per-packet latency. However, this advantage was not present in my experiments due to my use of a linear network topology whose spanning tree was identical to the shortest path between test hosts. In other words, the experimental topology was too simple to take advantage of one of SEIZE's main efficiency upgrades over Ethernet bridging. In a more complex and realistic network topology (specifically, one whose spanning tree encompassed one or more overly-long paths between switches), SEIZE would likely outperform Ethernet bridging.

In the absence of extensive control-plane traffic and a complex network topology, SEIZE is bested by Ethernet due to SEIZE's method of encapsulating all data packets within other packets for inter-switch transmission. While this mechanism is important for the system to function correctly, it also results in an extra per-packet overhead of 34 bytes that is used to encode the header data of each encapsulating packet. These extra headers result in SEIZE experiencing slightly higher loads on the network's physical resources (such as links and switch I/O buses and network cards) compared to Ethernet, which results in slightly lessened throughput.

With these limitations in mind, SEIZE's performance compared to that of Ethernet is quite respectable. At large packet sizes, SEIZE's throughput was over 95% of the equivalent Ethernet throughput for both unidirectional and bidirectional tests. This essentially tells us that, even in a linear network which is in some sense ideal for Ethernet bridging and too simple for SEIZE, our system still performs about as well as simple Ethernet bridges. Further tests should be designed to better simulate real-world enterprise networks: they should involve more switches in complex topologies and higher numbers of simultaneously-communicating hosts. These higher-complexity simulations would serve to better illustrate SEIZE's efficiency-oriented advantages over Ethernet – namely, superior control-plane mechanisms and shortest-path routing between switches. The likely result of these experiments would be a serious degradation in Ethernet's aggregate throughput and a much smaller hit to SEIZE's performance.

**4.6. Conclusions**

Moving the Click forwarding engine from user-level to the kernel resulted in a tremendous performance leap for the SEIZE networking architecture. In a 1 Gbps network, the unidirectional data bandwidth of 916 Mbps represents a

bandwidth utilization of over 91 percent, meaning that SEIZE is very efficiently using the network resources (gigabit links and network cards) at its disposal. In fact, the current limit on kernel SEIZE's throughput is probably the speed of these network adapters and links. The user-level SEIZE module is limited to forwarding about 42,000 packets per second based on some bottleneck probably related to user-Click's architecture and its need to constantly copy packets between user memory and kernel memory. Therefore, the user-level Click program seems to hold little potential for future versions of SEIZE. The Click kernel module is demonstrably more efficient and more convenient in practice, and it will serve as the core of SEIZE as the architecture evolves. An intriguing next step would be to deploy the current version of SEIZE on a network with 10 Gbps links and network adapters to try to determine if SEIZE would still be limited by link capacity or whether some other bottleneck would arise.

A final minor conclusion involves the new lightweight hash function and its effect on switch performance. Replacing the SEIZE prototype's MD5 hash function with the shorter and simpler Jenkins hash was viewed as an important step in removing a portion of the program which consumed an inordinate number of clock cycles considering its relatively simple task of mild randomization. The exact impact of this replacement was hard to measure, especially in light of the major change from user-level to kernel-level for the Click forwarding engine at the core of the SEIZE architecture. However, I did execute some tests which incorporated SEIZE's feature of ingress caching to compare with my large body of tests which did not incorporate caching. When caching is enabled, SEIZE's hash function is only invoked for the first few packets of a stream, rather than for every packet in turn, so these comparisons effectively helped me judge the effect that hashing has on a switch's performance. I noticed that the throughput numbers were identical for SEIZE when caching (and, by extension, hashing) was enabled and disabled. This leads me to conclude that the new Jenkins hash is certainly not slowing down the switches (since its utilization does not affect throughput numbers at all). This is a good indicator that the new hash function is successful in its attempt to use fewer clock cycles on each switch while preserving correct functionality; it is certainly not a bottleneck of the SEIZE system.

## 5. Future Work

Porting the SEIZE platform from a normal user-level program in FreeBSD to a kernel module within the Linux operating system represents a significant maturation of the architecture. Future researchers might explore many different avenues to further my goal of increasing switch throughput and reducing packet latency.

### 5.1. Multi-core processing

It would be interesting to try to take advantage of recent advances in multiprocessing to try to gain a speedup for each switch's routing process. Within the past few years, multi-core processors have become standard on almost all new computers, but we have not yet utilized anything other than uniprocessor machines for the SEIZE project. Running switches on multiprocessors without even modifying the SEIZE code base might gain significant performance boosts (namely, decreased CPU load and latency) due to better memory cache utilization and more clock cycles per second.

Another intriguing idea to get more out of multi-core is to experiment with multiprocessing optimization tools developed by Intel and other CPU makers. These tools are intended to help software developers design their programs to better take advantage of multi-core processors. This is usually achieved by explicitly dividing programs into independent threads of execution and then deploying different subsets of threads on each processing core. Intel also releases tools designed to profile the execution of multi-threaded programs – for example, such a tool can discover which segments of a large program are the using the most CPU time. Then, a developer might try to optimize this heavily-utilized code in order to gain an appreciable speedup in total execution time.[19]

SEIZE seems to naturally adapt to the multiprocessing philosophy. It is easy to envision, for example, a multiprocessing solution for SEIZE which assigns the handling of control-plane packets (like periodic OSPF routing updates) and the handling of data packets (regular host-host traffic) to different threads of execution, or even different dedicated CPUs. This separation of duties could be set up with minimal effort by inspecting each incoming packet and then routing it to one of several threads, and if set up properly with the help of the profiling tools mentioned above, it might result in some significant performance gains.

**5.2. Onboard Network Interface Card processing**

One particularly intriguing development route would be to explore the possibility of moving some of the processing of the forwarding engine onto the network interface card (NIC) itself. Locating microprocessors and some high-speed memory (such as DRAM and a few registers) on NICs is a growing trend in peripheral development (Tanenbaum 530-531). The reasoning behind this addition of computing power to the network card is to reduce the latency of constantly copying data back and forth between the adapter and the main CPU and memory unit of the computer. If some amount of processing can be performed on the network card itself, the switch will waste less time waiting for packet data to be transported between the adapter and the CPU. Therefore, a greater fraction of execution time would be used for actual processing and forwarding rather than simply waiting for copy operations to complete, and the overall throughput of the switch would potentially increase appreciably.

In some sense, this strategy is an extension of my work this semester: by moving the forwarding engine from user-level to kernel-level, I cut down on the number of times that an individual packet is copied while it travels through a SEIZE switch. Moving more processing out to the network card would further reduce the number of extraneous copies (from the NIC to the CPU), and potentially result in another speedup. One drawback is that the onboard memory and CPU on a NIC generally operate at a lower speed and capacity than a computer's main CPU and memory unit, so there is no guarantee that the overall performance would improve with heavier NIC computing utilization. Another route to explore would be the use of field-programmable gate arrays (rather than CPU and RAM) to perform this on-card processing; the use of FPGAs would probably alleviate speed concerns at the expense of slightly greater development complexity in moving the system to an FPGA-compatible implementation language.

**5.3. Using PollDevice elements in Click**

The Click distribution includes an element called PollDevice which is only compatible with the Linux kernel module. It is an optimized, hardware-specific version of the commonly-used ToHost element, which pulls packets off of the network card and pushes them to the other modules of the Click router. A PollDevice element is only compatible with switch hardware equipped with NICs that utilize the DEC Tulip fast Ethernet drivers or Intel E1000 gigabit Ethernet drivers (the element is designed to enhance NIC performance by utilizing driver-specific optimizations for these two drivers). Essentially, use of the PollDevice causes Click to check the network card for incoming packets more often than a similar Click configuration using a ToHost. Rather than depending on the NIC to send an interrupt to the CPU whenever incoming information is available from the network (interrupt-driven I/O), a PollDevice "polls" the card repeatedly, checking the card over and over on a very short interval to see if new any new data has arrived. The result is that individual packets lie idle on the NIC's memory for shorter time periods, there are fewer problems with packet overload and subsequent backup during routing, and each switch performs with higher throughput and less latency.

Currently, SEIZE does not utilize PollDevice elements, but it is clearly the next logical step in optimizing the performance of SEIZE with the Click kernel module. Conveniently, a portion of Emulab's available hardware actually uses the Intel E1000 gigabit Ethernet adapters for which PollDevice is designed. Therefore, experimenting with the PollDevice element should be as simple as replacing every instance of "ToHost" in the current Click configuration with "PollDevice." After SEIZE with kernel-level Click is more rigorously tested and benchmarked, the results should be compared against an identical version of SEIZE which uses PollDevice elements rather than ToHost elements. It would be very interesting to see the degree to which hardware-specific optimization affects the overall throughput of the forwarding engine.

# Appendix

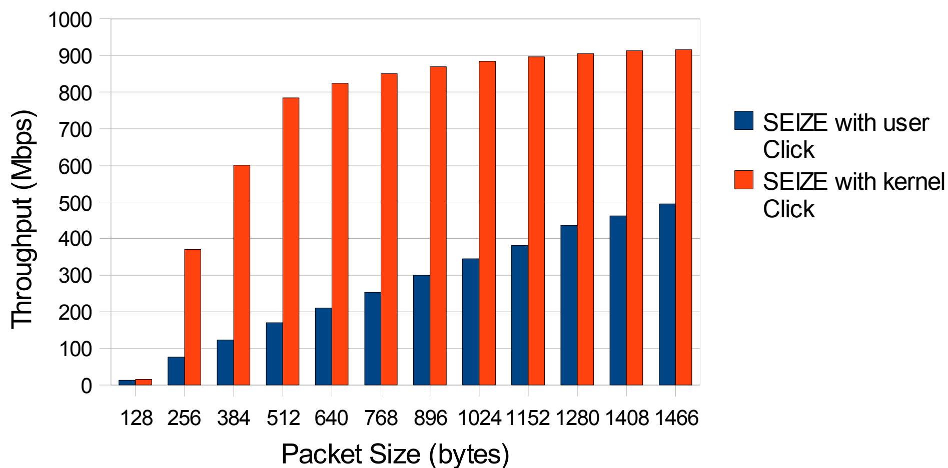


*Figure A1: Kernel-mode SEIZE outperforms the user-level version at every packet size.*

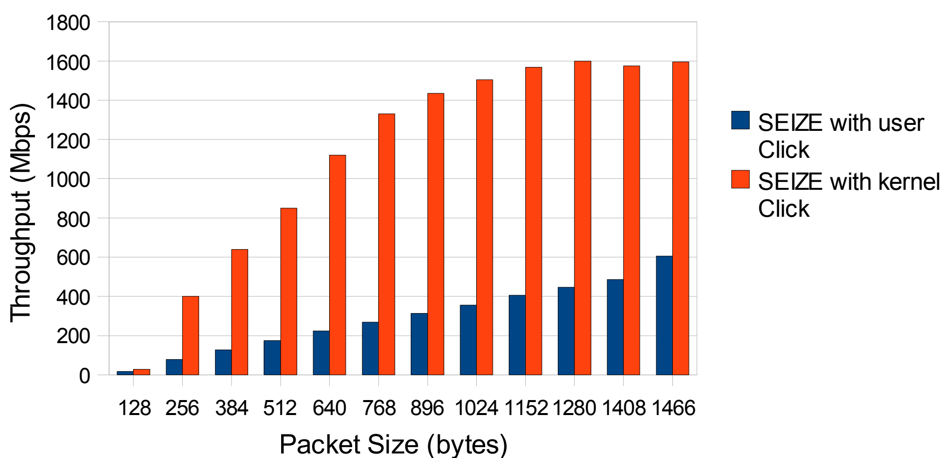


*Figure A2: Kernel-mode SEIZE scales to multiple streams of traffic effectively.*

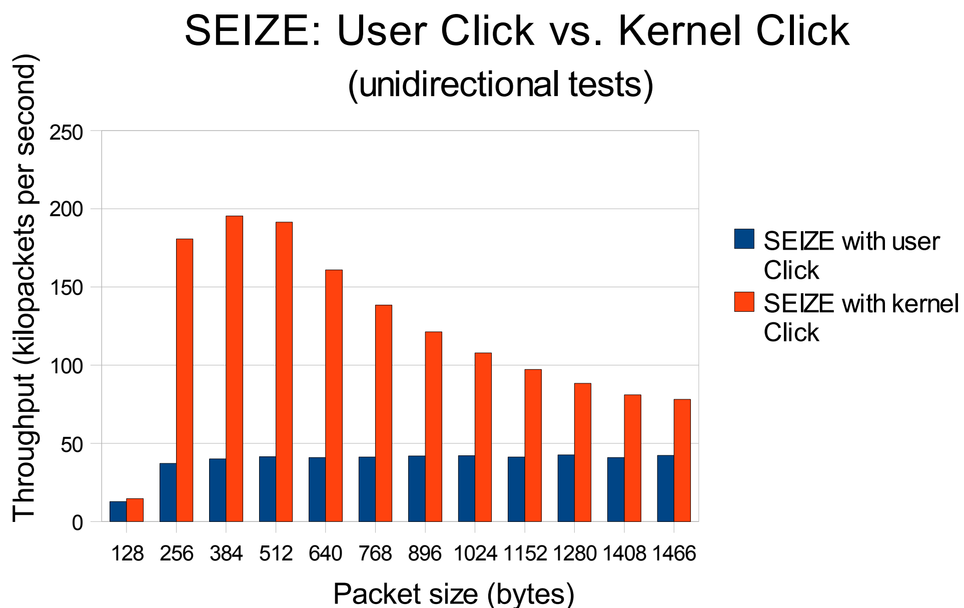


*Figure A3: User-level Click displays a throughput limit of 42 Kpps, independent of packet size.*

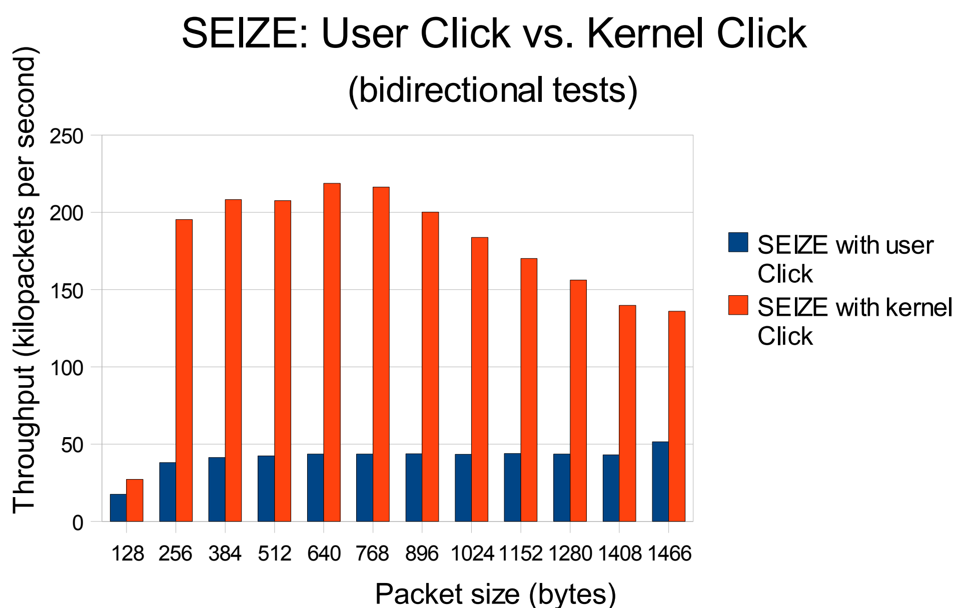


*Figure A4: The user-level SEIZE limit of 42 Kpps is still in effect, even with twice as much traffic.*

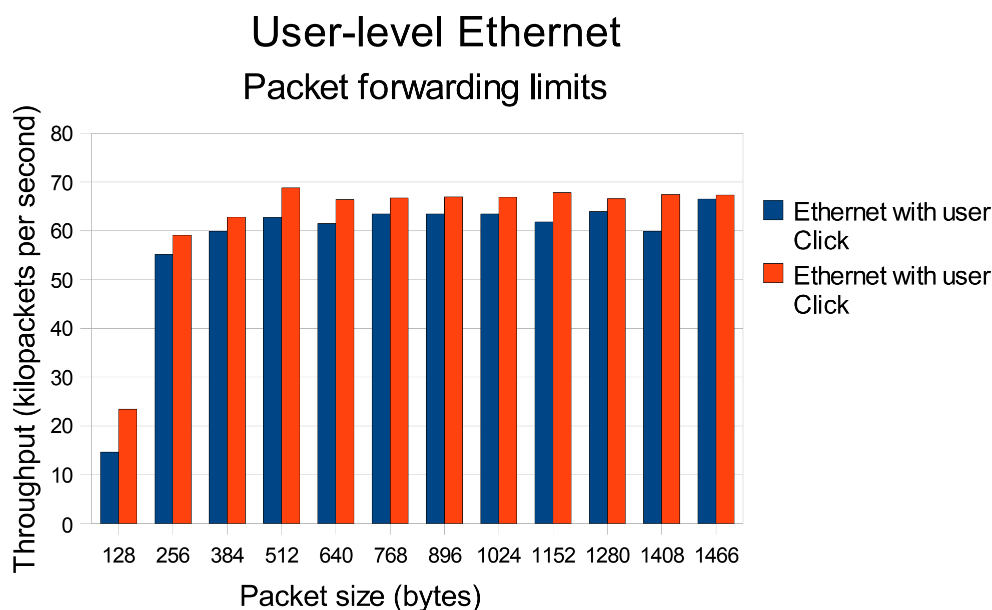


*Figure A5: A Click user-level engine configured to run the Ethernet bridging protocol displays an analogous packet rate limit of about 68 Kpps.*

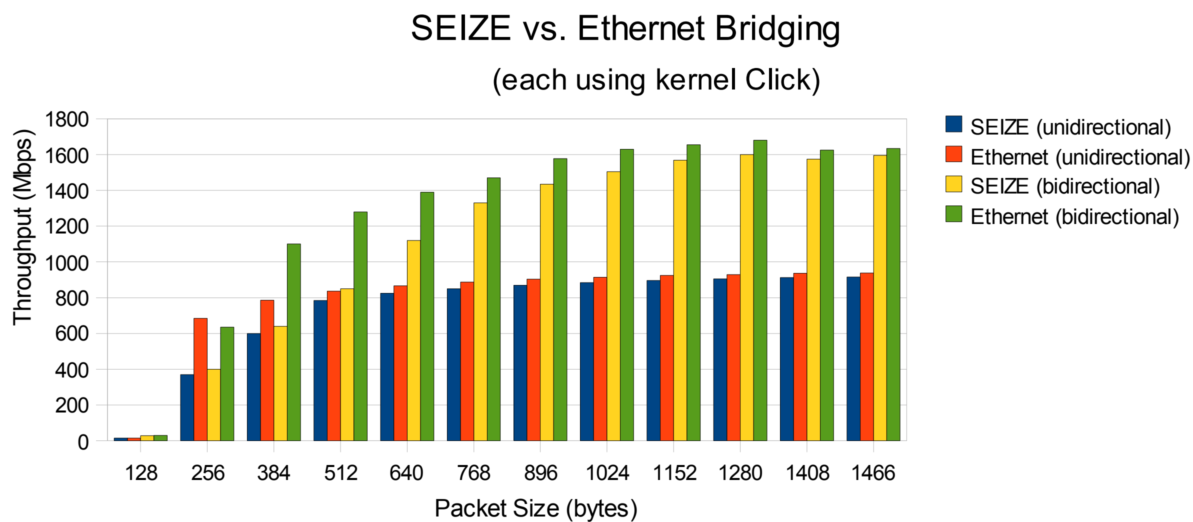


*Figure A6: For unidirectional and bidirectional tests, SEIZE demonstrates slightly worse performance than simple Ethernet bridging. This is largely due to the extremely simplistic test topology and lack of network dynamics.*

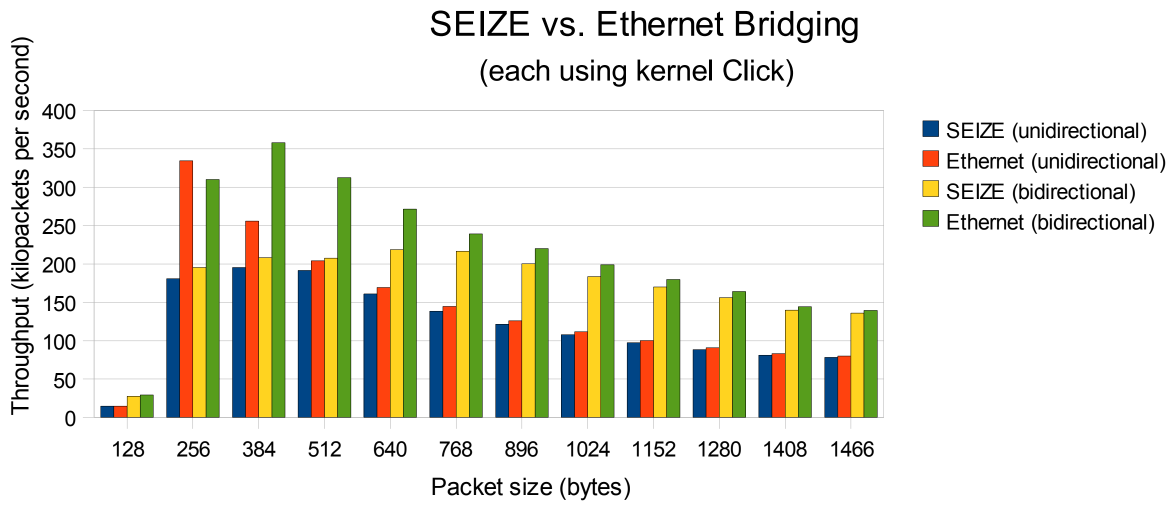


*Figure A7: When running in the Click kernel module, both SEIZE and Ethernet appear to be limited only by the link bandwidth of the network.*

## Acknowledgments

I was very fortunate to get lots of helpful assistance this semester as I worked on this project. I would like to especially thank my advisors Jennifer Rexford and Changhoon Kim, who were amazingly helpful at giving direction to my project and helping me out when I ran into technical problems. They also provided invaluable feedback during the process of writing this report. I'd also like to thank Eric Keller, who met with Changhoon and me to discuss the Click kernel module early in the semester. Members of the Click worldwide mailing list also made some very helpful suggestions when I got lost in the depths of Click; I'd particularly like to thank Eddie Kohler, Bart Braem, Beyers Cronje, and Joonwoo Park for their input.

I pledge my honor that this paper represents my own work in accordance with University regulations.

Daniel Pall

1 Peterson, Larry L. and Bruce S. Davie. *Computer Networks: A Systems Approach*. 3rd ed. San Francisco: Kaufman, 2003. 111-120.

2 Kim, Changhoon and Jennifer Rexford. "Revisiting Ethernet: Plug-and-play made scalable and efficient." Invited paper to appear in Proc. of IEEE Workshop 2007. 1. As found at http://www.cs.princeton.edu/%7Echkim/Research/SEIZE/LANMAN07/lanman07-seize.pdf.

3 "XORP Open Source IP Router." http://www.xorp.org/.

4 "The Click Modular Router Project." http://www.read.cs.ucla.edu/click/.

5 Morris, Robert, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. "The Click modular router". As found in *Operating Systems Review* 34 (5):217-231, December 1999. 5-8. http://pdos.csail.mit.edu/papers/click:sosp99/paper.pdf.

6 Tanenbaum, Andrew S. *Modern Operating Systems*. 2nd ed. Saddle River: Prentice-Hall, 2001. 296. Image found at http://www.prenhall.com/tanenbaum/cw_mos_2e/mos2e/ in JPG archive.

7 "NLANR/DAST: Iperf – The TCP/UDP Bandwidth Measurement Tool." The Board of Trustees of the University of Illinois. 2005. http://dast.nlanr.net/Projects/Iperf/.

8 "Tcpreplay." http://tcpreplay.synfin.net/trac/.

9 Click README file as found in Click 1.6.0 distribution. http://www.read.cs.ucla.edu/click/click-1.6.0.tar.gz.

10 "Emulab Tutorial: Custom OS." https://www.emulab.net/tutorial/tutorial.php3#CustomOS.

11 Anderson, Ross. *Security Engineering*. New York: Wiley, 2001. 103.

12 Kim, Changhoon, Matthew Caesar, and Jennifer Rexford. "Building Scalable Self-configuring Networks with SEIZE". Princeton University Computer Science Technical Report TR-801-07, October 2007. 12. As found at http://www.cs.princeton.edu/research/techreps/TR-801-07.

13 Jenkins, Bob. "A Hash Function for Hash Table Lookup." http://www.burtleburtle.net/bob/hash/doobs.html. As found on *Dr. Dobb's Portal*, September 1997. http://www.ddj.com.

14 "The New 'pc3000' Nodes". Emulab Knowledge Base entry. http://www.emulab.net/doc/docwrapper.php3?docname=pc3000.html.

15 The FreeBSD Project. FreeBSD 4.10-RELEASE Announcement. http://www.freebsd.org/releases/4.10R/announce.html.

16 Kernel.org archives. http://www.kernel.org/pub/linux/kernel/v2.6/.

17 PowerEdge 2850 Product Details. Dell, 2007. http://www.dell.com/content/products/productdetails.aspx/pedge_2850?c=us&cs=28&l=en&s=dfb.

18 Net:Pktgen. The Linux Foundation. http://www.linux-foundation.org/en/Net:Pktgen.

19 Domeika, Max and Lerie Kane. “Optimization Techniques for Intel Multi-Core Processors.” May 25, 2007. As found at http://softwarecommunity.intel.com/articles/eng/2674.htm.