The OASIS Group at U.C. Berkeley:
Research Summary and Future Directions

George Porter, Mel Tsai, Li Yin, Randy Katz
May 2003

Document Scope

This document is a forward-looking summary of the research and activities of the U.C. Berkeley OASIS group. Here we present a draft architecture for implementing application-specific "in the network" functionality within the coming generation of programmable network elements (PNEs). Our goal is to provide a description of our architecture in sufficient detail to make it suitable for external review and feedback, particularly by our research sponsors.

To set the stage for our proposals, we present an overview of the considerable research opportunities that exist for PNEs. We then review the state-of-the-art in the constituent technologies of programmable networks and network appliances, in terms of commonly encountered services and applications.

A key component of our research is exploring the right way to program PNEs, from a system of interacting PNEs down to the programming and configuration of a single PNE. We present a new framework for programming and developing applications for a single PNE and discuss the most relevant issues.

To illustrate the challenges of embedding new functionality into a network context, we present our a case study analysis of the performance of iSCSI, an emerging standard for mapping the SCSI storage device command set onto the IP protocol stack. The interaction between network transport, particularly congestion control, flow control, and reliable delivery, has considerable implications for the ability of applications to sustain high utilizations for network-attached storage devices.

We include our proposed programmable network testbed and our detailed experimental plan, including our plans for prototype implementations of additional services and applications in this context.

We conclude with our expectations as to the research results we hope to generate. In particular, we believe there are opportunities to better understand the interaction between control and data planes, the partitioning of functionality between end points and the network, and new approaches for achieving higher levels of reliability, safety, and performance in network-intensive applications.

Table of Contents

1. Introduction 3
2. Research Opportunities 3
3. Applications That Benefit From Computation in the Network 5
4. Existing Network Appliance Applications 6
   - *Traffic Monitoring, Shaping, and Usage Accounting 6*
   - *Layer 4-7 Switching 6*
   - *Firewalls 7*
   - *Intelligent Storage 7*
   - *Intrusion Detection 8*
   - *Network Overlays 9*
5. A Programming Model for PNEs 9
   - *Goals of the Single-PNE Programming Model 9*
   - *Background: Software Routing Tookits 10*
   - *Towards A New Single-PNE Programming Model 12*
   - *Overview of the Programming Model 13*
   - *Tackling the State Management Issue with Packet Tags 15*
   - *Summary of The Single-PNE Programming Model 16*
6. A Case Study: iSCSI 17
   - *Part 1: Interaction of Flow Control Mechanisms 18*
   - *Part 2: Impact of Packet Loss 23*
7. An Active Networking Testbed for Storage Apps 24
   - *The Alteon-iSD 24*
   - *Using the Alteon-iSD for Intelligent Storage 25*
   - *Future Directions for the Testbed 26*
8. Proposed Experimental Plan 27
   - *A Monitoring and Measurement Infrastructure 27*
   - *The Single-PNE Programming Model on a Linux Platform 27*
   - *Distributed State Maintenance among Device Ensembles 27*
   - *Cooperative Caching System to Support WAN-Connected SANs 27*
   - *A Peer-to-Peer Accelerator and Enforcement Point 28*
9. Conclusion and Summary 28
10. References 28

1. Introduction

Programmable network elements are network routers that perform complex, diverse, and configurable operations on packets in the routing fast path with minimum latency. PNEs are characterized by being able to classify, infer, and act upon packets using stateful information from both the control and data planes. PNEs represent a paradigm shift from today's fixed-function network routers to a more dynamic and application-aware packet processor that can interact and cooperate with other PNEs. The promise of PNEs is not only to accelerate and enhance existing network-intensive applications, but and also to enable new types of in-the-network functionality with increased reliability and reduced cost.

Today we see the emergence of so-called *network appliances* that indicate the trend towards PNEs. These include traffic shapers, server load balancers, layer 4-7 switches, web caches, SSL accelerators, firewalls, and intrusion detection systems. Each performs more extensive packet-level filtering and computation than traditional routers. Network appliances are enabled by two main factors–advances in hardware capabilities and the intelligent recasting of network applications and services into the network itself. Many of these applications were once relegated only to servers, which offer a high degree of flexibility and programmability (terms that we will further develop later), but have relatively low performance. In particular, PNEs offer fast-path computation and memory bandwidth that far exceeds a server's capabilities. On the other hand, a pure hardware solution would offer the highest performance, but the least flexibility. Thus, network appliances (and, in the future, PNEs) represent a balanced trade-off between flexibility and performance.

We expect the hardware architectures of many future PNEs to be based on a mix of general-purpose embedded processors, memories, and some special-purpose ASICs. This mix is similar to the one found on most existing network appliances. However, spurring the development of PNEs is the advent of *network processors* [1]. Network processors are high-performance programmable chips that aim to replace today's ASIC-based packet processors in routers and switches. In general, network processors are single-chip multiprocessors with specialized instructions, memories, and coprocessors that are optimized to perform programmable routing functions. They offer high degrees of parallelism and flexibility, strong L2-L7 filtering and computation, and promise to shorten the design cycle of future PNEs because they are off-the-shelf solutions. In some respects, the goals of network processors and PNEs are converging and complementary.

2. Research Opportunities

The OASIS group is interested in several aspects of PNEs and has performed background work in this arena over the last several months. We recognize several very interesting and important research questions for PNEs that we have begun to tackle. To succeed we must definitively consider the following questions:

What are the defining characteristics of an application that can benefit from network programmability?

Not every network application can be successfully cast into the PNE framework. As the hardware capabilities and software programming models of PNEs evolve, a clear

understanding of what applications can benefit from network programmability is important. Section 3 gives an introduction to our current thinking in this area.

What are the common operations in existing network applications?

Because PNEs are expected to operate at very high data rates with low latency, it is important to identify and understand the common computations and classifications required by network applications. These operations form a benchmark set that drives the design of the PNE's low-latency fast path; in Section 3 we discuss some of these operations for some example application areas.

How do you manage distributed state information?

Routers and switches are generally required to make only local decisions based on packet header information, which increases data parallelism and eases system complexity by allowing packets to be processed out-of-order and independently. However, there are many applications (for PNEs in particular) where the decision-making process is order dependent and/or driven by aggregated state information from previously identified packets. Moreover, the problem is further complicated when the state information is distributed across multiple nodes. We believe that understanding how to collect and use distributed state information is one of the most difficult design challenges for PNEs.

How do you successfully distribute functionality in a network of PNEs? What is the right system-level programming model of a network of PNEs?

We expect that applications for PNEs will require a system of PNEs working together to implement the desired functionality. For example, a geographically-distributed web cache might be implemented with several collaborating PNEs, each serving redundant copies of web pages for load balancing and performance purposes. In such distributed applications, it is important to consider how distributed state is aggregated and how application functionality is partitioned between the various PNEs in the network. In other words, not only should we focus on the programming model of a single PNE, we need to understand the system-level programming model of an ensemble of PNEs and the best way to partition distributed applications onto the underlying network elements and end points.

What is the right programming model for a single PNE?

While the system-level programming model of a group of PNEs is very important, it is also vitally important to get the software programming model of a *single* PNE correct. Indeed, the single-PNE programming model directly impacts reliability, flexibility, programmer productivity, and cost. Moreover, it is well known that the software interface presented to the programmer is often an afterthought when developing a device.

To develop the right single-PNE programming model, we must consider its functionality. The basic functionality of a PNE can be described as "sense, infer, and act on packets." As an example, a PNE might be responsible for monitoring traffic patterns to detect a DDoS attack. Here the PNE collects data about packets that flow through it (sense), continuously analyzes the collected data to detect an attack (infer), and allows or drops certain flows if an attack was detected (act).

To implement this sense/infer/act functionality, the programming model of a PNE must efficiently allow the programmer to (1) implement intricate filtering and classification of packets in an intuitive way, (2) make complex decisions based on aggregated state, and (3) modify, forward, or route the packet in unrestricted ways. The programming model should also encourage reliability and robustness of implementation–errors in software and/or configuration are the most crucial factors that influence the reliability and cost of a network-intensive application. As a secondary goal, the programming model must allow efficient exploitation of the underlying hardware to achieve high performance.

Section 5 gives an overview of our research and development in programming models for PNEs.

How do you quantify the flexibility and reliability of a PNE?

The market for PNEs will not be driven solely by high performance. Rather, the real importance of PNEs is their promised reduction in total cost of ownership, which is achieved because of a PNE's flexibility to support new types of functionality and combinations of existing functionality, combined with high reliability; something not always achievable with today's routers and servers. We believe that it is important to develop ways to quantify flexibility and reliability. Without an understanding of what makes one device more flexible and reliable than the next, we cannot develop new PNEs with better programming models. Unfortunately, quantifying such attributes is not simply a matter of determining the number of applications a PNE can support or how the system's mean-time-to-failure.

We have only begun to understand this important question–more research and discussion is required.

3. Applications That Benefit From Computation in the Network

We believe the primary characteristics of an application that are enhanced by casting some or all of its functionality to the network are:

- The endpoints are heterogeneous. In other words, the application or service has no single point where filtering or computation is natural or logical.
- When the filtering or computation accesses every bit in each packet. In such cases, the data rates become too high for servers, and the computation rate is too high for traditional routers. By separating a lightweight-filtering component from more heavyweight computation, data of interest can be isolated and further analyzed.
- The application is not fully general purpose, i.e., not exclusively relegated to a server. Or, the application can be partitioned into a general-purpose part and a specialized part that is suitable for casting into network-based functionality.
- When distributed state is involved, such as bandwidth monitoring or DDoS attack information, and the state must be quickly aggregated for analysis. Managing this aggregation and state maintenance is a challenge for architectures based on PNEs.

4. Existing Network Appliance Applications

The previous section described the characteristics of applications that can benefit from in-the-network computation. Admittedly, we cannot foresee tomorrow's "killer application" for PNEs. However, while we expect programmable network elements to make inroads in many types of applications, we believe the beginning of the PNE application roadmap starts with existing applications for network appliances. One force driving the evolution of PNEs is the need to consolidate functions of several network appliances into one box, for savings in both management cost and rack space. This "all-in-one" approach might be the first step towards realizing the programmability and reconfigurability of PNE boxes in the near future; such boxes will need this to implement different types of applications based on customer needs. Thus, when analyzing the requirements of future applications for PNEs, we believe it is useful to perform a taxonomy of applications for network appliances. This section provides such an overview.

Traffic Monitoring, Shaping, and Usage Accounting

Monitoring and controlling traffic flowing through specific points in the network is an important network application. This allows organizations to detect attacks, conform to service agreements, provide detailed and fine-grained customer billing, improve quality of service for a set of flows, and many other functions. However, future applications that use these facilities will require a level of sophistication and configurability not currently provided by most network appliances and routers. Indeed, these require complex classification and filtering to isolate specific flows and users. They need to keep statistics on potentially millions of flows while simultaneously analyzing groups of flows to detect patterns and conditions. Also, the PNEs that implement these facilities may cooperate and aggregate their statistics to implement the desired application(s). Another intriguing application of intelligent traffic shaping devices is restricting the flow of application-level data, rather than simply TCP byte streams. For example, it is advantageous to identify a flow of application-level requests and restrict their flow to improve server performance and avoid overdriving clients.

Layer 4-7 Switching

While traditional network routers and switches forward packets based on layer 2 and layer 3 headers, an increasing number of applications require examination of any header fields up to

Filters	Actions
• Increment/reset counters when packets arrive for a given TCP Flow • Identify command packets in a flow • Identify resources associated with a command packet	• Delay (or drop) packets to trigger TCP's congestion control mechanisms • Buffer packets • Drop certain command packets while delivering others • Send out a summary report to other PNEs • Incorporate learned summaries from other PNEs

Table 1. Traffic Shaping and L7 Switching Filters and Actions.

layer 7 (the application layer). Packet classification at this level is more complicated than required by L2/3 routing, and care must be taken to avoid increased processing latency and buffering. Example filters and actions for layer 4-7 switching (and also traffic shaping) are shown in Table 1. Applications that use layer 4-7 information are often complicated by the need to reorder and examine the headers of several packets in succession (e.g., due to fragmentation) before a forwarding decision can be made. Again, this highlights the importance of state management in complex protocol processing.

Existing applications that examine and modify application-layer headers include HTTP server load balancing, web caching, TCP offloading, application proxies, intrusion detection, SSL acceleration, and several others.

Firewalls

Firewalls generally perform two types of functions: packet filtering and network address translation at layers 2-4. Firewalls act as a barrier between an internal "secure" network and the external "insecure" network, applying rules and limiting access to the internal network. External access to the internal network is usually restricted to TCP sessions initiated by a host on the internal network. Thus, firewalls must keep records (state information in tables) of these TCP sessions to route properly. Firewalls must be able to examine the network and TCP headers for classification purposes; if application-level firewall rules are utilized, then the firewall must also be able to examine and understand that part of the packet. Software-based firewalls are often built with rule sets encoded as an ordered sequence. By sequentially traversing these "chains" of rules, a firewall ensures that the rule priorities and ordering are preserved. The disadvantage of encoding rules like this is that it becomes difficult to parallelize rule classification over a set of different rules.

Filters	Actions
• Src/dst IP address • Protocol • Flags • Src/dst port • Interface name • ICMP type • SYN packets • Type of service	• Redirect • Accept • Deny • Deny and return ICMP • Queue • Address translate

Table 2. Firewall Filters and Actions

Table 2 lists some example classification filters and actions common in firewalls.

Intelligent Storage

Storage Area Networks (SANs) consist of a collection of SCSI disks connected together through a fast interconnect, typically Fibre Channel. SANs provide highly reliable access to shared data in an organization or enterprise. They do not, however, inherently offer a rich set of storage-related

services to the devices connected to the fabric. For example, storage virtualization is the technique where a set of physical disks is presented to end-users as a set of virtualized disks. An *intelligent storage director* performs this virtualization. A storage director allows the SAN to better utilize storage resources, ease reconfiguration, implement server-free backup and recovery, increase security through access control, and reduce downtime.

One recent SAN development is the emergence of iSCSI, a new protocol designed to transport SCSI data over TCP/IP. Although iSCSI is an alternative to Fibre Channel, it will not replace FC and thus there is a need to bridge the two in some environments. Thus, in addition to storage virtualization, intelligent storage directors also provide such bridging.

To implement the desired functionality, storage directors must examine fields in both the TCP headers as well as iSCSI or FC headers. Storage directors must also extensively use collected state information to make forwarding and packet modification decisions at very high data rates.

Intelligent storage directors are currently being deployed by vendors such as Cisco and Brocade.

Filters	Actions
• iSCSI commands (and CDBs) vs iSCSI data • Logical Unit Number (LUN) • Logical Block Address (LBA) • Authentication Mechanism (CHAP, etc)	• Route packet • Replace LUN or LBA with result of hash table lookup • Block (Drop) request in case of masking • Redirect packet to one or more interfaces

Table 3. Intelligent Storage Filters and Actions.

Intrusion Detection

One of the most difficult challenges in networking is detecting, stopping, and preventing security breaches in the network. Intrusion detection methods attempt to monitor and detect a large number of different attacks. Current intrusion detection devices are significantly complicated by the large timescale in which data must be collected and monitored; even tremendous memory and processing capacity is sometimes not enough to detect certain attacks. With the increasing numbers of sophisticated and distributed attacks, the need for reliable intrusion detection is more apparent than ever. Furthermore, given their distributed nature, it is important that information about an attack is collected from multiple places. Combining observations from multiple perspectives allows network operators to better identify an attack. It is difficult or impossible to do this at an isolated point.

The ideal intrusion detection system requires all the properties of an ideal PNE: strong filtering and classification, decision-making based on a huge body of collected local and non-local state, high reliability and flexibility, and reconfigurability in the field.

Network Overlays

The protocols that underlie existing networks are resistant to change. The Internet protocol has gone largely unchanged for over 20 years. Thus, when implementing a new network application, one cannot assume that the underlying protocols can be changed. Network overlays are applications that attempt to emulate changes to the network's underlying routing mechanisms to implement desirable new properties. Network overlays also include dynamically assembled networks of hosts, often called peer-to-peer networks. Overlay networks and peer-to-peer networks include application-layer multicast networks, globally persistent storage systems, and commercial file-sharing systems such as Gnutella and Kazaa.

OASIS is currently investigating whether network overlays can benefit from a PNE-based infrastructure. Most existing overlay applications will probably not require the high bandwidth and computation rates provided by the PNEs we envision, but one can foresee new overlay applications that will.

5. A Programming Model for PNEs

As discussed in Section 2, there are essentially two different programming models that should be discussed for PNEs: the programming model for a network of collaborating PNEs, and the programming model for a single-PNE. In some respects they are related, because the programming model of a single-PNE must provide services and support for communication with and control over other PNEs. In other words, in some respects the single-PNE and system-level programming models are one in the same.

At this point OASIS does not have a complete proposal for the system-level programming model. Our investigations start with related work in distributed systems, databases, and multiprocessor systems (e.g., message passing, shared memory, consistency models). Our model is built "bottom-up," in that our initial investigations into network monitoring and event capture and inference will motivate primitives at PNEs to enable those services. From these, higher-level services and applications can be built. This bottom-up approach is outlined more thoroughly in our Experimental Plan (see Section 8).

Although we do not yet have an intuition for the system-level programming model, we have been developing a single-PNE programming model for some time. This single-PNE programming model meets several important goals, and was designed after examining the strengths and weaknesses of existing models for PC-based software routers. This new model offers an architecture-independent, high-level method for writing applications that also helps utilize the underlying hardware in an efficient way. This section presents an overview of our model. For brevity, references to the "programming model" refer to the single-PNE programming model in this section, and such comments do not necessarily apply to the yet to be specified system-level programming model for PNEs.

Goals of the Single-PNE Programming Model

The major goals of the programming model are summarized as follows:

- The ideal programming model should provide a high-level, abstracted environment for writing network applications in an architecture-independent way. The abstraction increases designer productivity by preventing her from dealing with the low-level details of the underlying architecture. Being architecture-independent, the software becomes portable to a

new architecture that supports the same interface. An abstracted design environment also reduces mistakes and bugs because the programmer spends less time worrying about details and more time on system-level application issues.

- The ideal programming model should support a compilation or configuration path that is portable to a number of hardware platforms. This path should maintain portability yet also achieve the required performance levels.

- Perhaps most importantly, the ideal PNE programming model should support a very diverse set of applications and services. To do so, the programming model must implement a variety of application primitives and also the models of computation that most network applications require.

Background: Software Routing Tookits

Hardware routers have command-line based management interfaces that allow easy configuration and setup of the router. However, routers are mostly fixed-function devices that generally cannot be reprogrammed to implement new applications and services without a significant redesign. Thus, one cannot look at current practices in routers to find the ideal programming model for tomorrow's next-generation PNEs. To find the ideal programming model, we get our inspiration from current research in *software routing toolkits*, which we refer to as "extensible routers."

Extensible routers [2] are software tools that support the design, simulation, and quick prototyping of network applications for servers, routers, or network appliances. The most prominent example is MIT's Click Modular Router language [3], based on Eddie Kohler's Ph.D. By stringing together a series of pre-written network *elements* in the Click router description language, a network routing application can be built with useful functionality. (Currently, the Click element database contains over 250 elements.) The Click runtime environment integrates this router application into the network stack of a Linux machine, allowing users to physically implement the network application on commodity hardware with relative ease.

Unfortunately, current-generation extensible routers have shortcomings. Scout [4] and Click, for example, are relatively fine-grained approaches that require users to implement network applications with simple packet-processing components. There is a significant start-up design time when a user wishes to implement or extend a real-world TCP/IP router application, because these approaches do not supply "out-of-the-box" router functionality by any means. We argue that almost any network application can be formulated an extension of a basic router, and thus starting with a "blank slate" is inefficient for any application of moderate to high complexity.

For an illustration of the fine-grained and low-level nature of Click, Figure 1 shows the Click implementation of a basic IP router with just two Ethernet ports. It contains over 38 elements and 50 connections between elements. Syntactically, the connections between elements are relatively inflexible in the face of changes. This is due to the topological and netlist-style fashion in which Click routers are built: adding or removing functionality to a design can be time-consuming in the Click language, especially when adding or removing the element(s) requires a series of other changes to accommodate the elements. In addition, this 38-element router does not include many important router functions commonly found on commercial routers. These functions include filtering rules, support for Ethernet VLANs and VLAN tagging, spanning tree algorithms, QoS or bandwidth limiting, higher-level routing protocols such as RIP or OSPF, and the list goes on. Thus, a user who wishes to add functionality *on top* of a basic router must first implement the router, which could easily require more time than the application design time. In fact, Click's element database does not even include most of the aforementioned features. Also, note that the

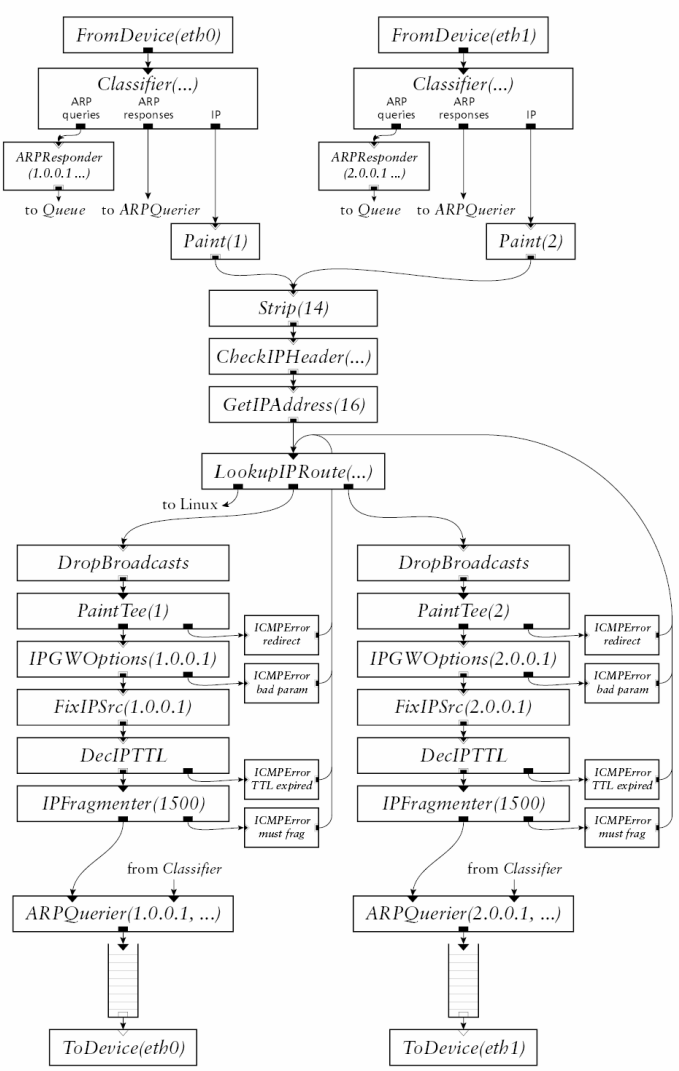

Figure 1. Basic Click IPv4 Router. Source: [3]

design in Figure 1 has only two Ethernet ports, whereas many commercial layer 2/3 routers and switches have up to 16 or more ports. Adding more ports will cause the Click language description (in terms of elements and connectors) to grow dramatically, especially if each port gets its own set of particular policies and filters. Yet, it is reasonable for a router to have different policies on every port.

Click's element database lacks many important items that are required to implement a wide variety of applications, despite constant additions to the database by various developers. Although Click is fundamentally "flexible" (because new elements can be added by the programmer by writing in C++), it does not have correct combination of flexibility *and* productivity required to meet the goals of the ideal PNE programming model. In other words, in its current form, Click cannot implement diverse, portable, high-performance, and reliable applications in an architecture-independent way.

Washington University's Router Plugins [5] is an extensible router that takes a slightly higher-level approach. Similar to Click (which extends Linux), this system is implemented as an extension to the NetBSD UNIX network stack. In this system, a network application is

implemented by loading "Plugin" kernel modules that perform forwarding, classification, and other operations on packets. A "Plugin Control Unit" (PCU) and the "Association Identification Unit" (AIU) control these kernel modules. The AIU is responsible for matching packets (using TCP/IP filter criteria) and are dispatched to the appropriate Plugin by the PCU. The actual hooks into the NetBSD network stack are provided by the "IPv4/IPv6 core," which is basically the interface software for the network hardware. The "Plugin Manager" is the NetBSD interface that loads and initializes the router application. Examples of router plugins include basic IPv4 longest prefix matching (for routing), packet classifiers, packet schedulers such as Deficit Round Robin (DRR), and IP options.

Although Router Plugins is a step in the right direction because it supports higher-level design (and thus increased productivity), it also has shortcomings. First, like Click and Scout, it does not come with a set of Plugins that support full out-of-the-box functionality that a commercial router might contain (VLANs, L7 classification, network address translation, port trunking, etc.). Second, default functionality cannot be achieved without writing significant new code in C++, which limits its flexibility and productivity combination. In short, Router Plugins and Click both suffer from similar problems.

Towards A New Single-PNE Programming Model

How does one achieve a high degree of flexibility while maintaining performance and programmer productivity? The Click language is very flexible because one can implement any desired functionality, but at the cost of significant development effort. On the other end of the spectrum, commercial hardware routers can be quickly configured (through their CLI), but they have limited flexibility. Thus, how does one achieve the flexibility of an extensible router such as Click with the ease of configuration and "high productivity" of a commercial router? We believe the ideal programming model for PNEs comes from answering this question.

Our new programming model uses the *generalized packet filter* as its basic primitive. Click and Router Plugins, on the other hand, use the "network element" as their basic primitive. We believe that the proper and careful selection of a basic primitive is critical to the success of the programming model, because the rest of the programming framework is built upon the basic primitive. If a programming framework is built upon a primitive that is too general and not domain-specific, the framework will not be ideal. Generalized packet filters perform a versatile set of network-oriented classifications and actions on packets, but are not necessarily geared towards general computation (although they can be modified to do so).

The idea behind generalized packet filters is best described by discussing "packet filters" in commercial routers. Nearly all commercial routers can apply filtering rules to packets traversing its ports. Usually, these filtering rules are described and instantiated in one centralized location on the command-line interface, and then selectively applied to the network ports as desired. In other words, an instance of a filter is created globally, then the programmer selects to which ports the filter instance is applied. A router cannot normally create an unlimited number of such filter instances; high-end routers often support up to several thousand filters. Filter instances are uniquely named with an index, i.e., "filter 0" through "filter 2048".

The basic operation of a packet filter can be described as "classify, then act if it matched the classification." Packet filters on most commercial routers can classify a packet based on fields in layers 2-4 on a packet. Then it performs some action if the packet matched. For example, a particular filter can be configured to "drop all packets with source IP address 10.0.x.x and TCP source port 80." Because filters are instantiated globally and uniquely named, changing the classification or action criteria of a filter instance will immediately propagate to any port on

which the filter was applied. Routers have a mechanism to determine the order in which filters are applied, because many different filters are often applied to a single port on a router. The most common mechanism is sequential execution: filters are processed sequentially from 0 up to the highest filter, e.g., 2048.

Packet filters are a powerful application building block in a "fixed-function" router. Very complex routing policies can be implemented using filters. For example, suppose a 64-port router is responsible for routing traffic between the various floors of a building. Each floor has different groups of users with different access to network storage, printers, servers, secure compute resources in offsite locations, and the Internet via a firewall. Describing such a router application in Click would be a complex endeavor. For every one of the 64 "ports" in the Click language description (represented by a series of elements), a different set of classification and action elements must be instantiated and inserted into the Click description. Based on past experience, this is very time-consuming and not amenable to frequent changes and updates in policies. With a commercial router, however, the network manager simply updates, removes, or adds new packet filters to the various ports with ease through the command line. This is one of the most important weaknesses of Click. Click is suitable for applications that require just a few network ports, but does not easily scale when the number of ports becomes large.

Packet filters, while powerful enough to implement complex policies on fixed-function routers, are not flexible enough to support the PNE applications. In our programming model we generalize the packet filter as an extension of the normal packet filter. For example, instead of matching packets based on TCP/IP (i.e., L2-L4) criteria, generalized packet filters can match packets based on *complex criteria* such as NAT forwarding table entries, packet annotations (i.e., tags), the HTTP URL, or any combination of L2-L7 fields. Furthermore, they do not need to have simplistic actions such as "allow" and "drop." Instead, they support other operations such as redirect, address translate, tag with state information, forward to the control processor, encrypt, compress, etc. Thus, the concept of a "generalized packet filter" is the same as a normal packet filter on a commercial router, but it supports arbitrary classifications and actions on packets to implement *any desired network functionality*.

Overview of the Programming Model

We believe that the generalized packet filter is a very powerful primitive for the PNE programming model. With just a *few types* (not hundreds) of generalized packet filters, very powerful and complex applications can be constructed with ease. For brevity, the programming model for PNEs is best illustrated with the example shown in Figure 2. The idea behind our programming model is to implement a router *virtual machine*. This virtual machine looks like a standard router architecture, but is highly configurable. The number of line cards, the backplane, the number and types of ports, etc., are configurable in this virtual machine. Once the programmer has created the basic aspects of the virtual machine, the router application is implemented on top of the virtual machine by customizing and adding generalized packet filters.

Figure 2 shows a pictorial representation of an example virtual machine in our programming model. In this model, all of the blue-shaded items are enabled "out-of-the-box." These items form a skeleton implementation of a commercial router, and can be tailored (through the command line) to fit the needs of the specific application. In the virtual machine, each network interface (port) is implemented by a including a router line card (two line cards are shown in Figure 2). These virtual line cards are connected by a virtual switched backplane. This organization closely models the architecture of a typical line-card-based router. To implement higher-level control protocols, a control processor (far right) is also conceptually added in this framework.

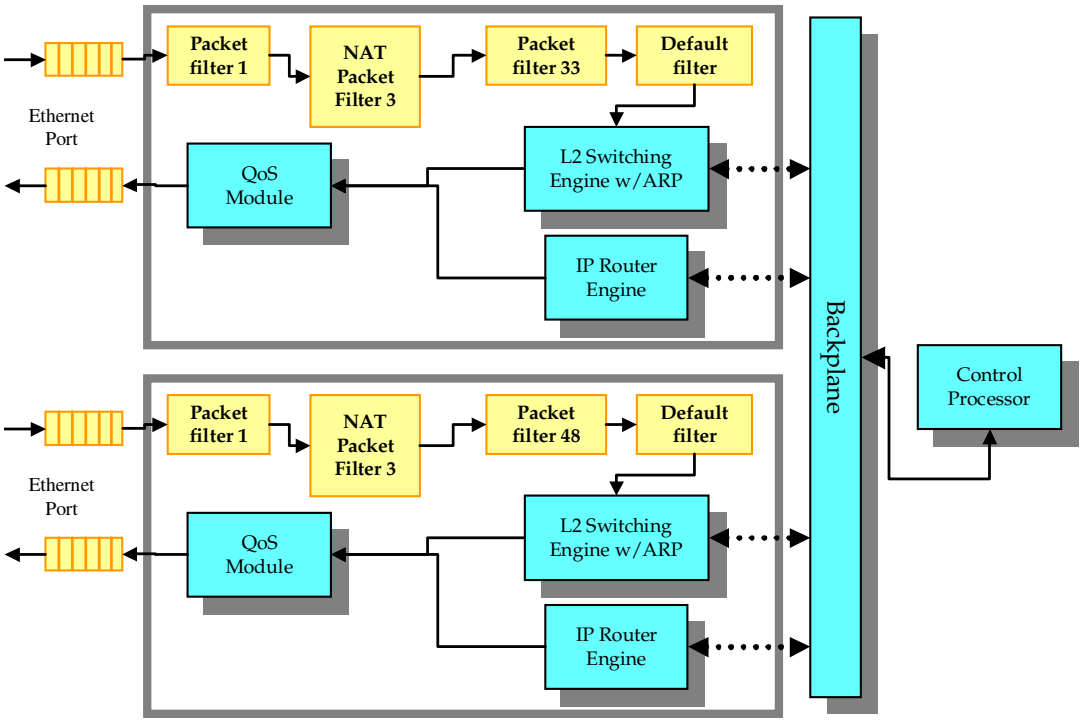

Figure 2. NAT example.

The programming model for PNEs is a "plug-and-play" environment for creating a router and writing applications for the router by adding and removing pieces of the virtual machine. Thus, not only is this programming model suitable for writing applications, it is also suitable for rapid prototyping of new hardware architectures and experimenting with hardware/software partitioning. (Although highly interesting, the advantages of this model for router design and simulation are beyond the scope of this document, but are being explored.) In addition, applications written in this programming model are portable to devices that export the same "virtual router" pieces through the CLI. This is possible because most applications will require customization only through the generalized packet filters.

One can see the ease in which a simple network router application can be built with this programming model. Within minutes (i.e., by setting some defaults for the blue-shaded items), a full TCP/IP router with a large number of ports can be implemented in software. The example in Figure 2 shows a simple network address translating (NAT) firewall. Normal access control rules related to the firewall can be implemented with standard L2-L4 packet filters (i.e., see packet filter 1), but the NAT functionality is implemented with "NAT Filter 3." This filter is a generalized packet filter that has special access to shared state (each instance of the NAT packet filter sees the state of all others in the router). It also performs an address-translate action if the packet meets the classification rules. A discussion of NAT is beyond the scope of this paper, but essentially the filter is responsible for matching ingress packets from an internal network, adding an entry into a flow table for each new flow, and performing the reverse operation on response packets from the public network.

The NAT example above illustrates an important aspect of the generalized packet filter: shared state. To properly implement NAT in Figure 2, the globally-defined "NAT packet filter 3" must properly share state among all ports in which it is enabled. This is because response packets can

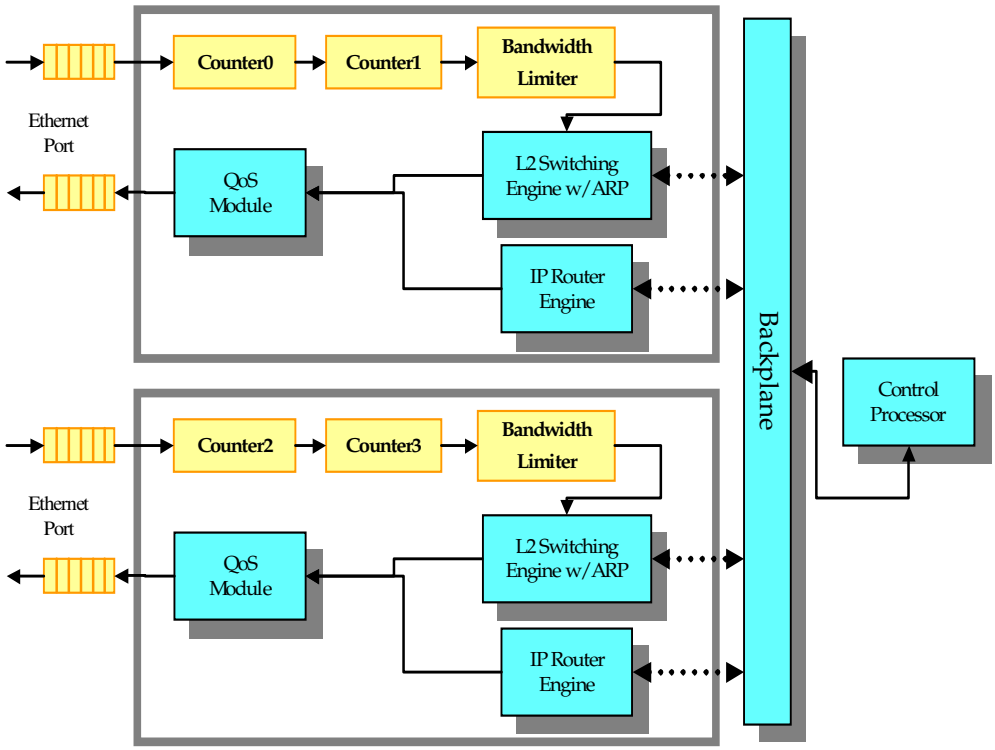

Figure 3. Traffic Shaping example.

arrive at the firewall on a different port than it originated, and different ports need to "see" the same translation tables. Thus, when actually implementing the NAT packet filter in this framework, the table-update information must be explicitly shared between the ports. State sharing can be performed via shared memory, message passing, or some other scheme. Depending on the actual target HW/SW implementation, the state-sharing mechanism may be an important decision. In general, many desirable generalized packet filters will require state sharing, and is an essential component of this framework.

Figure 3 shows a simple traffic shaping example implemented using this programming model. In this example, each of the two lines cards uses counters to monitor traffic flowing through the port. Counters are easily implemented by configuring standard generalized packet filters to classify certain types of traffic and then *tag* the packet with extra information when it matches. The "Bandwidth Limiter," also implemented by customizing a standard generalized packet filter, monitors each packet for the existence of certain tags and determines whether to allow or drop based on the ingress packet rate.

We have cast several more complex applications into this framework, but a discussion of them is beyond the scope of this document.

Tackling the State Management Issue with Packet Tags

Programming models such as Scout and Click were built to extend the abilities of a typical protocol-processing router. To a certain extent, the single-PNE programming model presented here also has this as a goal, but it aspires to do more. In particular, it aims to do more intensive packet processing on flows while maintaining complex state. In a sense, we argue that the single-

PNE programming model does the *simple* things *simply* (i.e., protocol processing, filtering, and routing), but provides underlying support for much more complex processing with state.

Dealing with per-flow state in a programmer-friendly and efficient way is difficult, and involves a lengthy discussion. Furthermore, we do not yet have all the answers. However, perhaps the most powerful mechanism in the single-PNE's programming model is the use of *packet tags*. A rudimentary form of packet tags is actually used in Click, but not exploited to its full potential.

Basically, any generalized packet filter can tag a packet with extra information, as described in the NAT and traffic shaping examples above. However, the fundamental mechanism of packet tagging is quite powerful. Tags are an additional instruction or data item that has been added by an upstream generalized packet filter. There is no implicit requirement on how or where the tag was added; it could have been from a filter on another line card in the router virtual machine, or from another PNE entirely (for now we leave aside security). Tags contain data such as counting or usage information on a particular flow, classification information such as IPv4 and TCP, or extracted application-layer information such as HTTP or iSCSI commands. They also contain instructions for control flow, too. For example, a packet might be tagged with an instruction that says "ignore all downstream filters of type X, Y, or Z" or "this is a low priority packet, drop if needed."

By judiciously building the single-PNE programming model using a virtual machine model, generalized packet filters, and a variety of useful tags, we believe that very complex packet processing is possible using this model. The shuffling of state between different components of the router virtual machine, or even between different PNEs, can take place using a tagging mechanism. It is possible to access tables and other shared state by creating generalized packet filters that add special tags. This is essentially a message-passing scheme, but a shared memory model could easily be emulated, or built from the ground up using generalized packet filters that allocate local and distributed state.

Packet tagging is a work-in-progress; much needs to be done to develop a methodology for understanding its effectiveness to tackle state management and other complex processing tasks.

Click and Scout could potentially use packet tagging to the extent shown here, but their architectures were not built from the ground up to support them, and their would require a major shift in the way programmers write applications for these software routing toolkits.

Summary of the Single-PNE Programming Model

The programming model presented in this section is a high-level and abstracted way to implement network applications in an architecture-independent framework. Architecture independence is achieved by using a virtual machine model that represents the components of a real PNE or router. Generalized packet filters are the key to productivity and flexibility in this model. Instead of writing new code for each function and dealing with the low-level details of the hardware, generalized packet filters are high-level application building blocks for network applications. Using just a few types of generalized packet filters, a diverse set of operations can be performed on packets simply by configuring them at the command line. We avoid the use of custom-written generalized packet filters wherever possible, as this limits the portability of the "code" for this programming model.

The use of packet tagging significantly extends the power and scope of generalized packet filters. We hope to show that packet tagging is an effective way to deal with per-flow state and complex control tasks that involve the coordination of multiple generalized packet filters.

This programming model is currently under development within OASIS, and a proof-of-concept prototype is being written to verify our claims. Also, as stated at the beginning of this section, we are currently investigating the relationship between system-level and the single-PNE programming models. We suspect that significant support for the system-level programming model must be natively included in the single-PE programming model in to be successful.

6. A Case Study: iSCSI

OASIS is interested in intelligent storage applications and their role in next-generation PNEs. An important part of our research is analyzing the effects of embedding new functionality into the network. This type of analysis helps us understand and drive the capabilities of PNEs so they can support the new functionality.

Wide-area storage networks are becoming increasingly important. They integrate storage with the network, providing the illusion of centralized storage for administrative purposes while distributing it for better survivability. iSCSI is emerging as the key protocol in this environment. It embeds SCSI commands into iSCSI PDU (protocol data unit) and transmits them over the network using TCP/IP. To illustrate the complexity of network and storage integration, we have performed a case study analyzing iSCSI and TCP/IP interactions. Its goals is to understand how:

- iSCSI, at layer 5 interacts with the TCP congestion control algorithm at layer 4;
- router queue size affects performance;
- packet loss affects sustained I/O performance;

iSCSI fundamentals is beyond the scope of this document, but an overview can be found in [6].

These questions help us understand the dynamics of embedding storage in the network and how to design algorithms that intelligently coordinate the storage and the network operations and sustain end-to-end throughput.

To perform our experiments, we developed a simulator by modifying ns-2 [7]. Within ns-2, the iSCSI initiator and iSCSI target are simulated as a TCP application. The protocol stack of the simulator is given in Figure 4. Using this simulator we investigated the performance impact of sending an iSCSI PDU (protocol data unit) over TCP versus accessing the disk directly. Because our goals are to analyze how iSCSI layer parameter can affect network congestion and how the system recovers when network is congested and packets are dropped, therefore, we ignore the disk operation and assume it is not the bottleneck. We also ignore issues such as the effects of local caching, disk operation reordering, intelligent disk out-of-order operation execution, the interaction of disk speed with network transmission, etc.

In the remaining discussion, the target application transmits a large file over the network using iSCSI. Since we assume that the disk is not the bottleneck, our results are not affected by whether the file is transmitted from computer to computer or from a computer directly to a disk. Because our goal is to analyze how the application behaves during network congestion and packet loss events, we focus on the congestion state and congestion recovery phase of TCP and its influence on the SCSI processing. We only consider the steady state network behavior when the network is experiencing packet loss and not transmitting at saturated speed. Therefore the maximum link bandwidth does not matter in our simulation. For any link bandwidth, we can always adjust other parameters to observe network behavior under network congestion. To speed up the simulation, we assume each link has a bandwidth of 1Mb/s. In this situation, disk is faster than the network

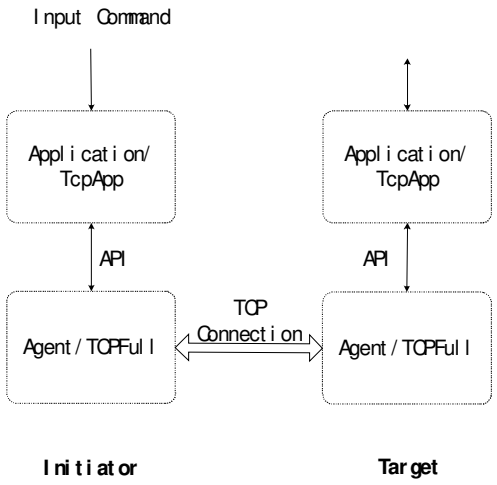

Figure 4. ns-2 Simulation of iSCSI

and the network is purposefully the bottleneck. The round-trip time between the initiator and the target is 34ms. The buffer size in the router is 20 packets of size 1500 bytes (30 kilobytes total) unless otherwise specified.

The simulation consists of two parts. First, we study the interaction of flow control mechanisms in iSCSI layer and transport layer. Our goal is to understand the impact of iSCSI layer parameter Maximum-Burst-Size and the router queue size on the performance. The simulation suggests that the Maximum-Burst-Size can affect the network congestion and congestion recovery behavior. In addition, a fixed Maximum-Burst-Size cannot guarantee a best performance no matter how big it is, and therefore an adaptive Maximum-Burst-Size based on the router queue size is more preferable for a better performance. Second, we investigate the performance impact of network loss rate to the overall throughput and the disk operation. It is shown that the loss rate has a big impact on the overall throughput. In addition, to avoid disk performance degradation caused by network loss, cache can be used to hide the network loss from the disk to a limited degree.

Part 1: Interaction of Flow Control Mechanisms

We investigate the interaction of flow control mechanisms in iSCSI layer and TCP layer. In iSCSI protocol, a session level parameter called iSCSI Maximum-Burst-Size controls the maximum SCSI data payload in bytes in a Data-in or a solicited Data-out iSCSI sequence. This is essentially the flow control mechanism in the iSCSI layer that avoids overflowing the application's buffer. At the same time, the TCP protocol uses its congestion window to avoid overflow of the TCP receiver buffer and network congestion. When iSCSI runs on TCP, the latter adapts to the network condition to avoid buffer overflow and congestion.

The maximum data that can be transmitted in each burst is determined by the minimum of the Maximum-Burst-Size in the iSCSI layer and the TCP congestion window size. Because the network traffic is dynamic and bursty, the queuing resources inside the network for different flows vary. As a result, the Maximum-Burst-Size that can fully utilize the network queuing resources and achieve the best performance is changing. If the application always has a large buffer available, the application might always allocate a big application buffer such that the iSCSI buffer will not be the bottleneck even if the network is running in its highest speed. However, it might be the case that the buffer in the application layer is shared by several sessions, in which

case the application buffer becomes a resource which needs to be allocated efficiently. In this case, the Maximum-Burst-Size can be set adaptively according to the queuing resources inside the network without sacrificing performance. In addition, the current design of the iSCSI protocol allows parameter tuning on the fly. Therefore, dynamically Maximum-Burst-Size tuning according to the network condition is possible in implementation. One thing to notice here is that there might be multiple iSCSI parameters that can be tuned to adapt to the network conditions such as the iSCSI PDU size, the first-burst-size, *etc*. The impacts of these parameters will be investigated in our future work. At this stage, we focus only on the Maximum-Burst-Size.

In the following simulations, we study how Maximum-Burst-Size and router queue size affects the performance. These simulation results agree with our suspect that a Maximum-Burst-Size tuning scheme adapting to the router queue size will benefit the performance. It gives some insight about possible solution to dynamically adjust the Maximum-Burst-Size, yet the design of Maximum-Burst-Size tuning algorithms needs more investigation.

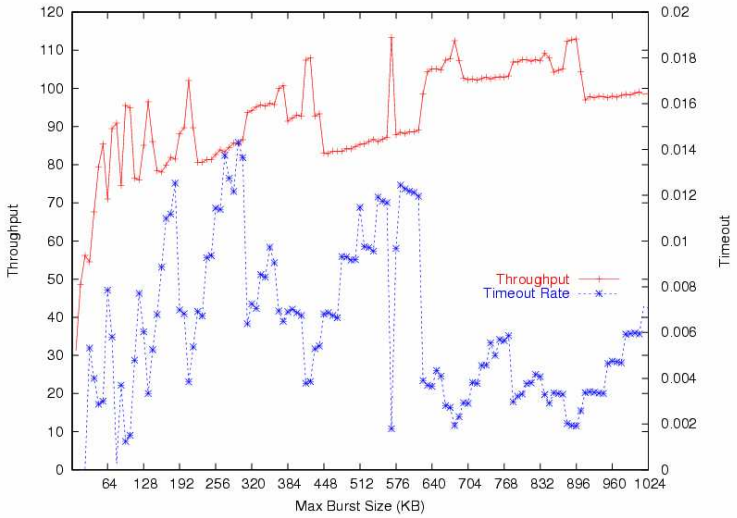

Figure 5. Impact of Maximum Burst Size

Part 1.1: Impact of Various Maximum-Burst-Size

To design the tuning algorithm, we need to first answer the question how Maximum-Burst-Size affects transfer rate performance. In the following experiment, we send a large file (400MB) over the connection under Various Maximum-Burst-Size configuration. In this part, we temporarily ignore the existence of cross traffic to understand the direct impact of Maximum-Burst-Size on the network congestion and overall performance even the network condition stays the same.

The top curve in Figure 5 shows the overall throughput. At the beginning of the curve, the throughput climbs fast. At around 32KB, the throughput stays around 80KB to 100KB with some fluctuation (±10KB). Though the throughput still increases, the improvement is small compared to the big jump at the beginning. Notice here, the link bandwidth is 1Mb/s in our simulation, the throughput in Figure 6 saturates around 110KB. To understand this behavior, we trace the queue behavior for each Maximum-Burst-Size and calculate the corresponding average packet drop rate. The packet drop rate is defined as the ratio of number of packet drop to total packets delivered. We zoom in the curve from 8KB to 200KB in Figure 6.

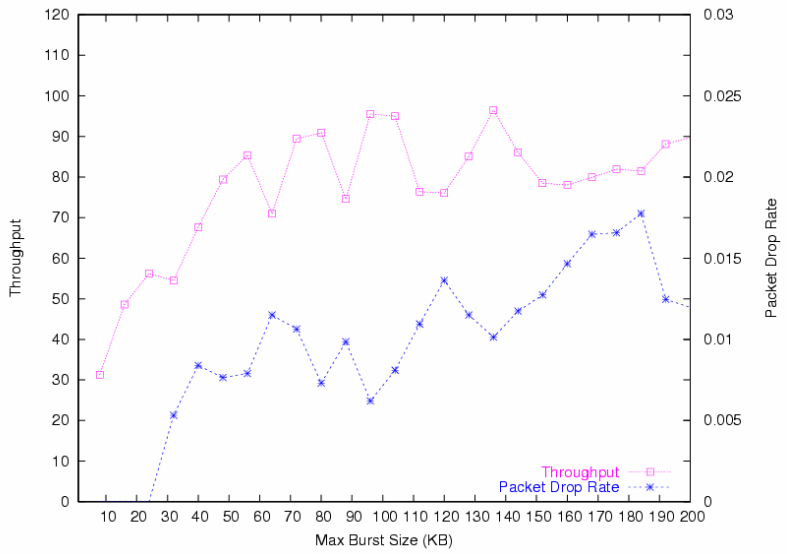

Figure 6. Impact of Maximum-Burst-Size (8KB-200KB)

As shown in the "Packet Drop Rate" curve in Figure 6, at the beginning of the curve, from 8KB to 24KB, increasing Maximum-Burst-Size will not cause any packet drops. Therefore, as a result, we can see a continuous increase in the overall throughput. However, when the Maximum-Burst-Size increases to 32KB, it starts experiencing packet drops because it is bigger than the router buffer size. As a result, the throughput decreases rather than increases.

Clearly in most cases when the packet drop rate increases, the throughput will drop. One exception is the 32KB and 40KB case. Although 32KB has a smaller packet drop rate, it has a worse performance than 40KB. This is because 32KB (≈22 pkts) does not have enough packets to trigger fast retransmission while 40KB (≈28 pkts) can trigger fast retransmission.

We can see from 5 that the fluctuation of packet drop rate causes the throughput to oscillate. The oscillation in throughput is very undesirable for both disk operations and network queue management. To understand why the packet drop rate oscillates, we plot the TCP traces for 64KB and 80KB in Figure 7 and Figure 8. As we can see from the figures, although 64KB has less data to send, it experiences more packet drops and timeouts than 80KB. The reason behind this is as follows: For 64KB, each iSCSI burst has 46 packets (one packet is used to send the WRITE command to the target). The TCP window starts from 1 and doubles in each RTT and reaches 32. Till now, only 15 packets left. With a 20 packets router buffer, no packet will be dropped. As a result, the congestion window increases to 47. When next burst becomes available, the sender will start transmitting all packets out. As a result, it overloads the network and causes a burst loss. While for 80KB (56 packets) case, it also starts from 1 and reaches 32. When the window size equals to 32, it still has 25 packets to be transmitted. Therefore, it will get 5 packets dropped and detect the congestion. As a result, in next burst data transmission, it will not cause new packet dropped. From the figure, we can see both window sizes will reach 22-23 and experience multiple packet drops again. 64KB has worse performance than 80KB because it does not have enough packets to trigger fast retransmission and results in more timeouts. In general, the fluctuation is caused by the interaction between the iSCSI layer flow control and TCP congestion control.

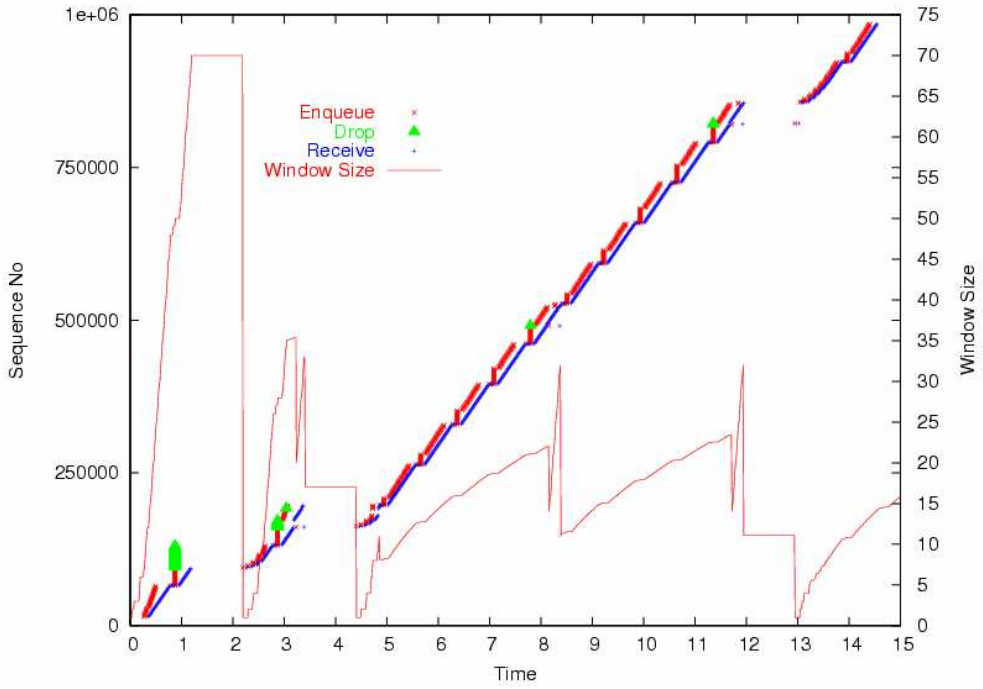

Figure 7. TCP Trace for 64 KB

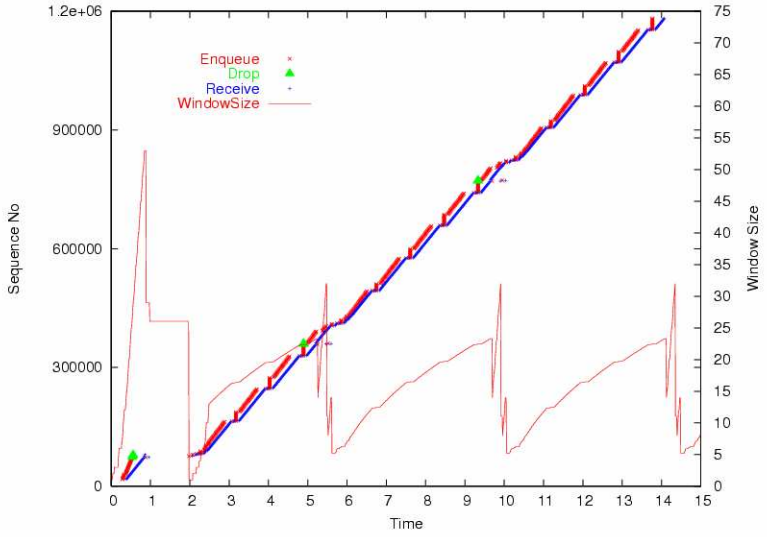

Figure 8. TCP Trace for 80 KB

Another interesting observation is that it seems the performance shows some periodical up and down behavior (Figure 5). Therefore, one possible solution to avoid the fluctuation is to set the Maximum-Burst-Size according to the periodical behavior. But more studies are required to confirm the periodicity.

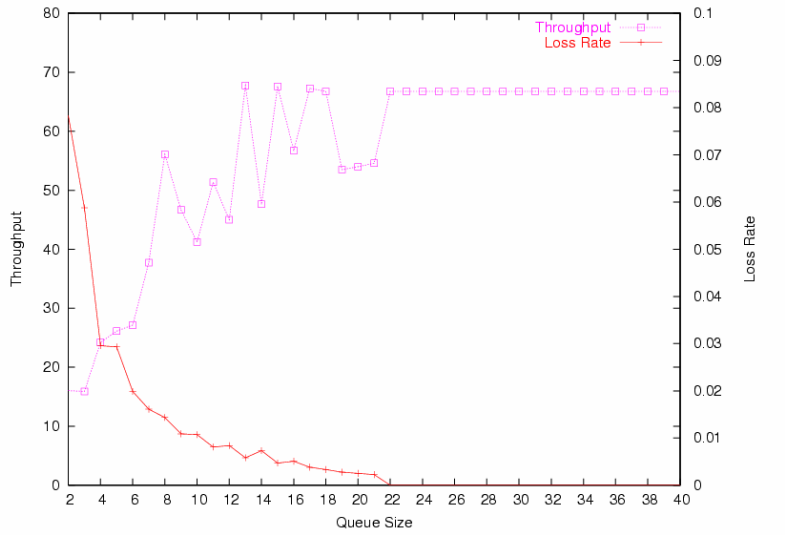

Figure 9. Impact of Router Queue Size

Part 1.2: Impact of Queue Size

In the previous section, we discussed for a given queue size how the performance is affected by the Maximum-Burst-Size. Here we discuss how performance changes with available router queue size. The available router queue size will be affected by cross traffic. In this experiment, we configure the Maximum-Burst-Size as 32KB and send 100MB across the network.

From Figure 9, as router queue size changes, the performance with the fixed Maximum-Burst-Size of 32KB has large variance. This suggests that we should tune the Maximum-Burst-Size according to the network condition rather than keep it fixed. In the second half of the curve, performance is stable, because for the Maximum-Burst-Size of 32KB with the router buffer larger than 21 packets (32KB), no drops occur and the bottleneck is the Maximum-Burst-Size.

We can draw the following conclusions. First, a bigger Maximum-Burst-Size does not necessarily imply better performance due to the interaction between iSCSI and TCP congestion control. Even if you set the Maximum-Burst-Size larger than the highest network capacity, it is still possible to achieve better performance using less application buffer. Second, because the network varies, a fixed Maximum-Burst-Size is not ideal because it either underutilizes the network bandwidth or results in unnecessary packet timeouts and worse performance.

However, all our previous analyses are based on simulation. To verify our results, our immediate next step is to perform similar study using the real testbed. If similar burstiness is observed in the testbed, the next question is how to design an algorithm which smoothly maintains a high throughput while embedding the storage into the network. One possible direction to go is that the sender can adaptively buffer data and adjust the sending rate based on the buffer capacity on the path adaptively. Some lessons can be learnt from streaming media where buffer is used to absorb burstiness and provide smooth video play. Another direction might be adjusting Maximum-Burst-Size based on the periodicity pattern. As stated earlier, our simulation results show some periodicity of the throughput. If we can capture the periodicity pattern, we can design an algorithm such that the throughput will stay at the peak smoothly.

Part 2: Impact of Packet Loss

In Part 1, we already see how the packet drop rate directly affects the overall throughput. In this part, we first simulate the performance for various link loss probabilities and then analyze how packet loss affects the I/O disk operation.

We first test the performance with a link loss rate varying from 0 to 0.2 and simulate the performance for 3 Maximum-Burst-Sizes. Figure 10 shows, as the loss rate increases, the performance will drop dramatically and converges to zero eventually. In addition, for various Maximum-Burst-Sizes, even if their performances differ a lot for small link loss probability, they will achieve same throughput as the link loss probability getting larger. This is because the TCP window is limited by packet drops and becomes the bottleneck.

Figure 10 shows the impact on the overall throughput for the whole file transmission. What is the impact of packet loss on disk operation? When we are transmitting I/O commands over the network, if a packet gets dropped, the TCP will stop delivering packets to iSCSI buffer until the packet is retransmitted. This means the disk will have no data to operate for some time. The disk will either stay idle or move its arm to process another session. In both cases, it will increase I/O command operation latency and reduce disk's efficiency. For example, for a disk with a 5ms seek time and 20MB/s sequential access throughput. If a timeout takes 500ms, staying idle for 500ms means we waste 10MB sequential data access. For fast retransmission, if the network is stable, the TCP congestion window will oscillate around the optimal window size (W packets). Fast retransmission will happen every $\frac{W * RTT}{2}$ second. If a fast retransmission takes f seconds, staying idle during the fast retransmission will waste $\frac{2f}{W * RTT}$ disk efficiency. If a disk will switch to another session during timeout or fast retransmission, two extra seek times are lost to switch to the new session and switch back from the new session. A disk cache can hide the packet loss to some degree. To hide the timeout from disk operation, the cache needs to at least store 500ms's data such that when TCP stops sending data to the disk, the disk can continue getting data from its cache. In other words, a 10MB's disk cache is required to hide single timeout for one application. How frequent timeout can happen without interrupting disk operation? Let's assume

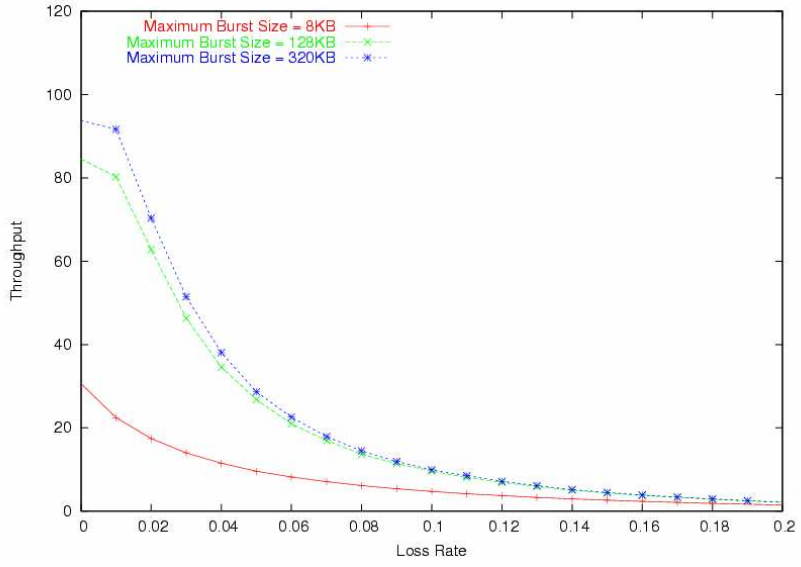

Figure 10. Impact of Loss Rate

the interval between two timeouts is X. Figure 11 shows the behavior of the cache.

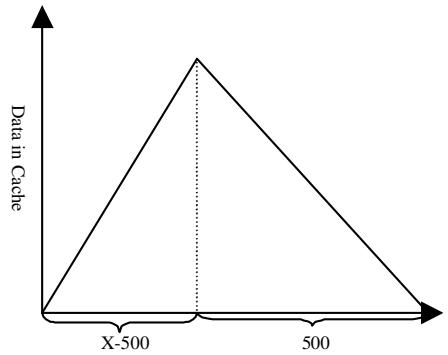

Figure 11. Disk Cache Behavior

As Figure 11 shows, to make timeout transparent to the disk, total data arrived within X-0.5 seconds must be larger than the data accessed in the timeout period. Assume the disk cache is large enough to hold all data arriving within X-0.5 seconds. X is derived using equation (X-0.5)**network-speed*> X**disk-access-speed*.

7. An Active Networking Testbed for Storage Apps

In 2002, we assembled a programmable networking testbed that allowed us to investigate its usefulness and programming model for storage applications. This section briefly summarizes it and the experiments we performed. It is located in Soda Hall on the UC Berkeley campus. Its layout is shown in Figure 12, and consists of a Passport 8600 router, an Alteon web switch and integrated service director (iSD), and several PCs used to host the storage applications. It is arranged so we can construct a client-server storage application in which all packets that flow between the client(s) and server(s) are first routed through the Alteon-iSD combination.

The Alteon-iSD

The Alteon-iSD is a commercially-available combination of a L2-L7 web switch (the Alteon) and a general-purpose PC (the iSD). We consider the Alteon-iSD to be a type of programmable network element. See Figure 13. Packets from user flows are classified using L2-L7 information on the Alteon. If a packet matches the filters, it is redirected to the iSD, which is able to perform arbitrary computation on the packet. Once finished, the iSD returns the (potentially modified) packet to the Alteon, which forwards the packet to its (potentially new) destination. In a sense, this combination looks like a router and server in one. However, what makes this approach unique is that the iSD can also update the filtering and routing tables of the Alteon at high speeds. High speed updates are achieved by a low-level control protocol between the Alteon and the iSD.

High performance is achieved by the Alteon-iSD as follows. Routers excel at making fast classification and forwarding decisions, but cannot perform computation on packets. On the other hand, servers can perform computation on packets, but cannot process packets at high data rates. The Alteon-iSD achieves “the best of both worlds” by taking advantage of *flow-based* routing. In many applications for PNEs, only the first few packets of a newly-encountered flow require significant computation. In server load balancing, for example, a new HTTP session may require some initial processing on packets in the flow (to make the load-balancing decision), but all subsequent packets are simply routed to the chosen HTTP server. The Alteon-iSD works this

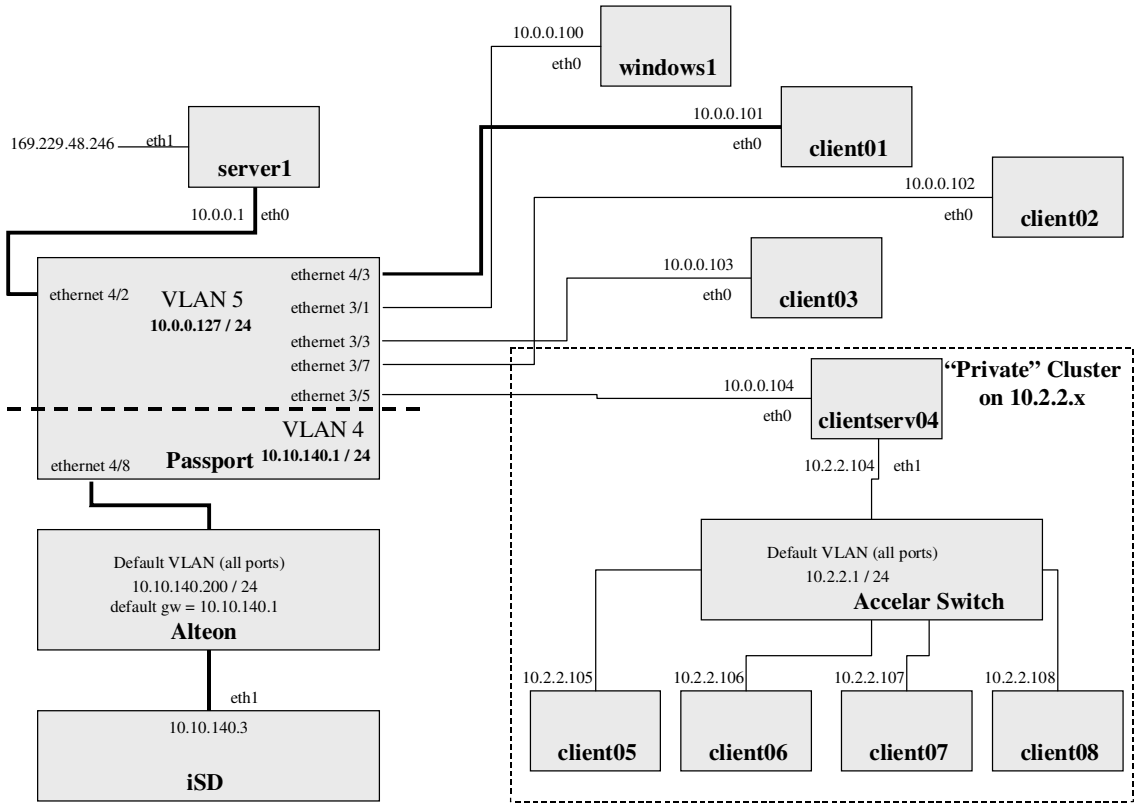

Figure 12. Layout of the Testbed

way. Ideally, the Alteon redirects only the first few packets of a *new flow* to the iSD, not every packet. The iSD, serving as the decision maker, might determine that the Alteon should (for example) redirect all packets belonging to this flow to a particular destination. Thus, overall performance is significantly increased because now the Alteon (the "fast" data path) can perform the address translation required for redirection. In summary, very high performance is achieved because routing and decision-making occurs on a per-flow basis, not on a per-packet basis.

Using the Alteon-iSD for Intelligent Storage

One of our main goals was to evaluate the use of the Alteon-iSD and its programming model for

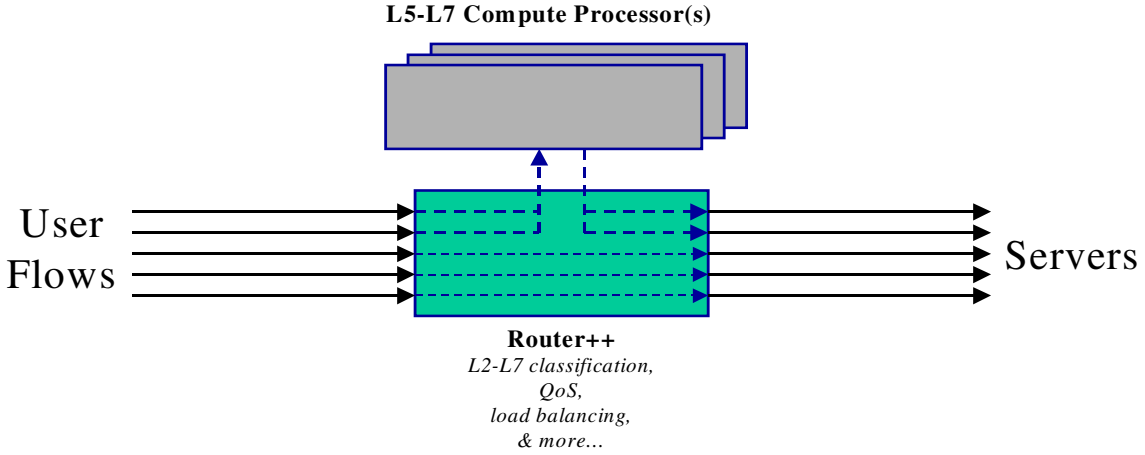

Figure 13. Basic Alteon-iSD Operation

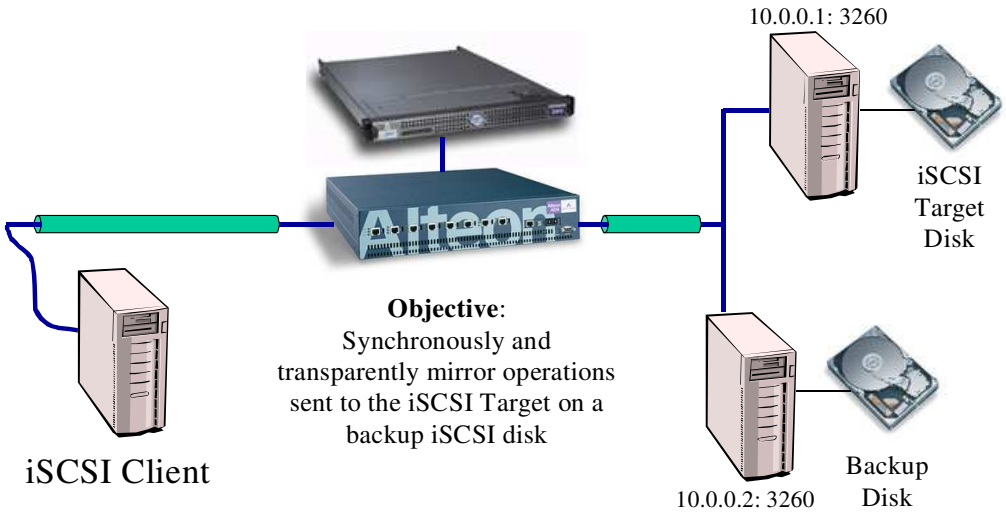

Figure 14. An iSCSI Synchronous Mirroring Example.

intelligent storage applications. Our experiment was to implement a simple iSCSI synchronous mirroring example on our testbed; see Figure 14. The iSCSI client, shown on the left-hand side of the figure, believes that it is accessing a remote iSCSI disk at IP address 10.0.0.1. However, by intercepting iSCSI traffic between the client and the target disk at 10.0.0.1, we are able to synchronously transmit the same iSCSI commands and data to a backup iSCSI disk at 10.0.0.2. In essence, all actions performed by the client on the iSCSI target disk are mirrored on the backup disk, and this is performed transparently to the client.

We exposed a limitation of the Alteon-iSD in this experiment. High performance is achieved only when a few packets from each flow are sent to the iSD. However, the Alteon has L7 filtering only for HTTP packets (it's a web switch). It can't distinguish between different iSCSI packets, or commands versus data, because the iSCSI protocol sits on top of TCP/IP at L7. The only way to classify iSCSI packets is based on their TCP/IP header, which is port 3260. Thus, the Alteon's sends traffic with destination port 3260 to the iSD, thus distinguishing between iSCSI and non-iSCSI traffic. A firmware update to the Alteon could include iSCSI-specific filters, which would have increased performance and functionality. The iSD was forced to examine every iSCSI command and data packet, so the performance degradation was high. Nonetheless, we found that the Alteon-iSD did not degrade performance when intercepting packets to perform synchronous mirroring. This is because the rate of data transfer between the client and iSCSI target disk was limited by the speed of the physical SCSI disk, and not the Alteon-iSD.

Future Directions for the Testbed

Our original testbed was constructed for the iSCSI synchronous mirroring experiment. However, we have equipment to expand its size and usefulness. In particular, we have another Passport 8600 router, three more Alteon web switches, and 20 more iSDs in a 1U form factor. Because they are PCs running Linux, the iSDs can also be used as generic client PCs and we plan to use them as such. We also have a PacketShaper device from Packeteer. This is a network appliance for traffic management that supports fine-grained management and monitoring of flows traversing its network ports. To support realistic storage experiments, we would like to extend the testbed with some storage-specific equipment. For example, we would like an Intelligent Storage Director and iSCSI devices such as TCP-offload engines (TOEs) and iSCSI host bus adaptors.

8. Proposed Experimental Plan

This section presents several next step experiments.

A Monitoring and Measurement Infrastructure

A critical role of PNEs is to monitor and measure network traffic. Their ability to store state—and communicate it to other PNEs—allows monitoring for error conditions indicated by data path events (and not just control path events). These could indicate, for example, the imminent failure of a disk in a SAN, a misconfigured BGP router that causes traffic loss from a subset of prefixes, or a distributed attack. Traditional servers are usually unable to perform such monitoring and decision-making at line speeds, especially considering that these network error conditions are infrequent and the detection may involve measurements at geographically distributed locations. We wish to discover the types of network monitoring and measurement facilities that are ideal for future PNE applications. Our first experiment will be to implement a monitoring and measurement infrastructure for our testbed (or possibly a modified version of our testbed).

The Single-PNE Programming Model on a Linux Platform

As a proof-of-concept and eventually a public software release, we are currently implementing the concepts of the single-PNE programming model on a standard PC running Linux. In other words, we are building the components of the router virtual machine architecture in a Linux runtime environment with a command-line based management interface.

Distributed State Maintenance among Device Ensembles

While maintaining per-flow and per-application state at a PNE is necessary for many network appliance applications, enabling new classes of applications will require inter-PNE communication. For example, in a SAN environment, the fast interconnect bridging iSCSI targets and initiators can be made up of numerous switches and routers. Applications that require information about the state of the SAN will need to coordinate data arriving at multiple PNEs. In addition to exchanging raw data, PNEs will need to selectively exchange subsets, summaries, and other condensed forms of their measurements. The remote PNE will need to aggregate those reports with its data to form a coherent view of the status of the system. We envision that this distributed state maintenance functionality will be used by higher level applications, and thus will need to adapt to its needs in a configurable and dynamic way.

Cooperative Caching System to Support WAN-Connected SANs

In this project, two SANs are connected with each other via WAN links (either private lines or paths through the public Internet). PNEs in the SAN fabric monitor iSCSI block requests and prefetch data blocks likely to be used in the future. These prefetched blocks are managed on local storage by the PNEs. Furthermore, the PNEs enable iSCSI initiators to cache data blocks from distantly located targets. By intercepting—and also redirecting—subsequent iSCSI requests to the local target, the overall storage system latency will decrease, and thus application performance will likely increase. To implement this, PNEs must intercept and understand iSCSI control messages. They must coordinate with each other to properly redirect cache hits to local disks. Lastly, they must ensure that all requests see a consistent and up-to-date view of the filesystem.

A Peer-to-Peer Accelerator and Enforcement Point

This project shares some similarity to the SAN-to-SAN caching and prefetching system. A peer-to-peer accelerator intercepts p2p control messages, and using topology-based information, redirects those messages to destinations likely to have the required information. In many access networks, e.g., DSL or Cable Modem, there are large numbers of sources and sinks of popular media content. By observing network traffic, and redirecting or otherwise responding to these, p2p accelerators can increase perceived performance for end clients.

A small but important modification to this scheme allows "do not download" lists to exist at the points of redirection. If a given query matches an entry in such a list, the request can be dropped. In this way, the PNE-enabled network acts as an enforcement point for restricting the distribution of p2p content. Since it is reasonable to assume that p2p traffic might be disguised or otherwise hidden as other network traffic, standard caching and transparent proxy mechanisms will not be effective to perform the acceleration and enforcement capabilities.

9. Conclusion and Summary

This paper presented an overview of the OASIS research group activities and goals. There are a large number of interesting and open questions about the future of PNEs. We have begun to address several of these questions and are motivated to answer the others. We feel that the understanding the properties of an ideal system-level PNE programming model is critical to success for PNEs due to cost and reliability concerns. We are also particularly interested in uncovering tomorrow's killer-applications for PNEs, although we believe that the near-term role of PNEs will be for consolidation of several existing network appliances into one box.

10. References

[1] N. Shah, "Understanding Network Processors," M.S.E.E. Thesis, Dept. of Electrical Engineering & Computer Sciences, Univ. of California, Berkeley, 2001.

[2] Yitzchak Gottlieb, Larry Peterson, "A Comparative Study of Extensible Routers," 2002 IEEE Open Architectures and Network Programming Proceedings, 51-62, June 2002.

[3] R. Morris et al. "The Click modular router." In 17th Symposium on Operating Systems Principles (SOSP'99), Kiawah Island, SC, 1999.

[4] Mosberger, D., "Scout: A Path-based Operating System," Ph.D. Dissertation, Department of Computer Science, University of Arizona, July 1997.

[5] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., "Router Plugins - A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers," in Proceedings of ACM SIGCOMM'98, September 1998.

[6] Internet Drafts: iSCSI. http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-20.txt

[7] The UCB/LBNL/VINT Network Simulator - ns(version 2). http://www.isi.edu/nsnam/ns