

Layer 4/7 Switching and Other Custom IP Traffic Processing using the NEPPI API

Bell Laboratories, Lucent Technologies
600 Mountain Ave
Murray Hill, NJ 07974
USA

Abstract— Layer 4/7 switching refers to making use of TCP header information and payload content when processing packets within a switch. The manner in which the layer 4/7 information is used depends on the type of switched traffic (e.g. web traffic), and the desired functionality (e.g. load balancing). Implementing layer 4/7 switching efficiently on PC-based hardware running a general-purpose OS requires additional support from the OS. We have developed such support and an API for the Linux OS. This paper shows how layer 4/7 switching applications can be implemented using this support. In particular, we discuss the implementation of TCP redirection, transparent caching, and URL-aware switching.

Keywords— Layer 4/7 switching, programmable packet processing, active extensions, URL-aware redirection.

I. INTRODUCTION

Layer 4 and 7 switching services are becoming increasingly important as the use of web server farms and web caches becomes more widespread. Devices which provide layer 4 services can use TCP or UDP transport layer information (e.g. port numbers found in TCP/UDP headers) in making packet forwarding decisions. Switches which include layer 7 services can make use of application-layer information (i.e. packet payload) as well. The emergence of products providing layer 4 and 7 switching is a relatively recent phenomenon which is mostly fueled by the need to improve the performance and reliability of Internet services through the use of web server clusters and web caches.

The terms layer 4 and layer 7 switching are used to describe devices with a wide spectrum of capabilities. A layer 4 switch may simply be a switch which can be configured to direct all traffic with particular destination TCP ports to a particular network port. For example, it may switch all traffic going to port 80 (used for HTTP traffic) to a particular port on the switch where a web cache is attached. This may be done to provide transparent web caching. More sophisticated layer 4 switches may provide additional functionalities such as network address translation (NAT), load balancing among servers or caches, server

fault tolerance, and configurable quality of service for different kinds of traffic.

Layer 7 switches add the ability to use information found in the payload of packets in order to provide even more sophisticated capabilities. For example, the URL found in HTTP GET requests can be examined to determine whether an image is being requested. If so, all packets belonging to the TCP connection corresponding to this request can be switched to a server which is optimized to deliver images. Hence, URL-aware switching can be performed. Another common use for parsing the content of HTTP requests at the switch is directing requests for un-cacheable content (e.g. results of a CGI script) to an origin web server instead of a web cache, thus eliminating unnecessary load on the cache. Other examples of layer 7 capabilities include converting complete URLs to absolute URLs in GET requests to support transparent web caching with standard proxy caches (see [4] and later in this paper), and offloading encryption processing for SSL traffic from servers by performing it at the switch instead [17].

Layer 4/7 switches perform tasks which are related to specific applications that produce the switched traffic. Hence, the switches have to evolve along with the applications and the ways in which the applications are used. Also, many layer 4/7 functionalities are non-standard, non-trivial, and bear more resemblance to capabilities offered by software running on general-purpose computers than the capabilities of traditional routers and switches which are dedicated to standard layer 2/3 tasks. As a result of the need to support the more complex and rapidly evolving layer 4/7 capabilities, new switch architectures are appearing in which powerful CPUs are included. Some of these new switches contain CPUs dedicated to ports on the switch for rapid packet manipulation, a main CPU for management and configuration, and various hardware components for accelerating common functionalities such as table lookup and packet queueing. Other layer 4/7 switches are based on a PC architecture with one powerful main CPU which performs all the tasks. Naturally, the latter devices may support fewer ports, but they often provide

a richer set of features.

In addition to facing the hardware challenges posed by layer 4/7 switches, the developer of such switches has to devise a software architecture which will facilitate the implementation of the rather complex functionality of the switch. Our interest was in distilling some common primitives which could serve as building blocks for implementing many potential layer 4/7 services. Such primitives could be implemented efficiently and then be used to simplify the development of the switch. We chose to develop our layer 4/7 infrastructure on a PC architecture running the Linux OS, and we chose to implement our layer 4/7 primitives within loadable Linux kernel modules. Other choices of hardware and software could be made. In fact, the layer 4/7 building blocks described in this paper could potentially be implemented to run in dedicated port processors or even in ASICs.

The software infrastructure which we developed is called NEPPI (Network Element for Programmable Packet Injection). The main focus of this paper is on the implementation of layer 4/7 switching capabilities using NEPPI. A description of NEPPI and its API is also included. More information about NEPPI and a description of an earlier version of it can be found in [2]. A NEPPI-based network element will typically be placed in front of clients or servers, or at the edge of enterprise networks. Hence, we use the term *programmable gateway* to describe such a network element.

II. ARCHITECTURE AND PACKET PROCESSING

The main component of NEPPI is called the *dispatcher*. The dispatcher implements the primitives used for building layer 4/7 switching services. The dispatcher is a loadable kernel module. The services themselves are implemented as *gateway programs* which are typically kernel modules as well. NEPPI allows gateway programs which run as user-level processes as well as gateway programs which execute as kernel modules. In some instances, the functionality of a layer 4/7 service is divided between both a user-level program and a kernel module. An example of such a service is the URL-aware switching service which is described in section IV-C.

Gateway programs communicate with the dispatcher for the purpose of obtaining, manipulating, and injecting packets. Gateway programs send the dispatcher rules that specify the properties of packets on which they wish to operate. Such rules may include IP address ranges for the source and destination, TCP/UDP port number ranges, etc. Based on the rules obtained from the gateway programs, the dispatcher generates packet filter rules which are sent to the Linux packet filter. An arriving packet which trig-

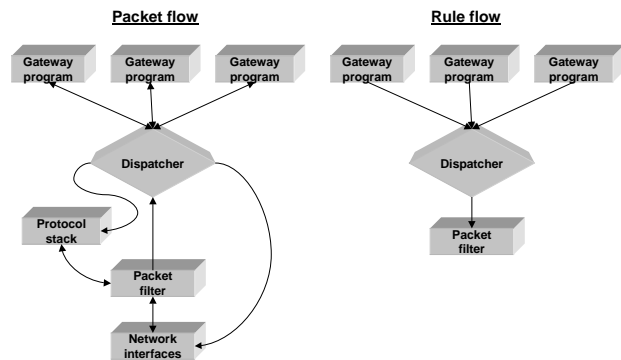


Fig. 1. Packet flow and rule flow in NEPPI

gers a rule is sent from the packet filter to the dispatcher which either sends it to the requesting gateway program, or manipulates it in accordance with a manipulation rule specified by the requesting gateway program (such manipulation rules include address translations, TCP sequence number changes, and TCP window size changes.) It is currently required that no overlap exist among the sets of rules submitted by different gateway programs. Packets generated or modified by the gateway programs are sent to the dispatcher which in turn modifies them further (if they match a manipulation rule), and injects them into the network. In the case where the destination address of a packet is the switch itself, the dispatcher will hand the packet to the protocol stack instead of the network interfaces. Figure 1 illustrates the flow of packets and rules.

A useful feature is the support for redirecting packets to a TCP port. Gateway programs may request the dispatcher to redirect certain packets to a TCP port on the switch. Such packets will be sent to the local protocol stack even if the destination IP address on these packets is not that of the switch. A user-level program which listens to the redirection TCP port will receive this data and will even be able to determine the original destination for the data. This feature is used mainly when the switch performs layer 7 services in transparent mode. A service which uses this feature is described in section IV-C.

Packets are processed at up to three stages as shown in the flowchart of figure 2. A packet which does not match any of the rules submitted by gateway programs is processed in the standard way by the protocol stack. Other packets are given to the dispatcher. The dispatcher may manipulate a packet by carrying out operations requested earlier by gateway programs (these operations are address translations, TCP sequence number changes, and TCP window size changes). Alternatively, the dispatcher may forward the packet to the requesting gateway program for processing. Once the packet is processed by the gateway program, it is received by the dispatcher which may

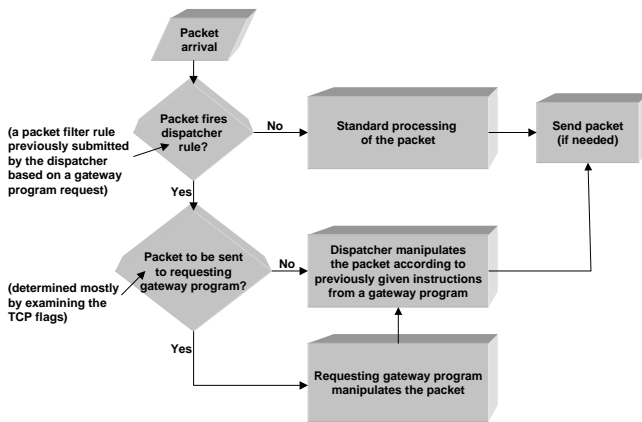


Fig. 2. NEPPI packet processing

further process it before re-injecting it into the network.

One of the goals of this architecture was to perform packet header manipulations which are generic and which occur on a large number of packets (such as address translations and TCP sequence number changes) at the dispatcher, while performing manipulations (including payload modifications) which are more specialized and which occur only on isolated packets at the gateway programs. Since gateway programs may reside in user space, packet processing at the gateway programs may incur significantly more overhead than packet processing at the dispatcher. In such a case, the NEPPI architecture will provide a “fast path” for most packets (the ones processed by the dispatcher), and a “slow path” for only few packets (the ones processed by the gateway programs). An interest in a potential implementation of NEPPI on a hardware-based switch provided further motivation for studying the separation of the packet processing functionality into two tiers: a specialized tier implemented in software which examines and manipulates relatively few packets, and a more generic tier implemented in hardware which manipulates a large number of packets in accordance with instructions provided by the software tier. Alternatively, in the case of an intelligent switch with port processors and a central processor, the dispatcher functionality may be supported by port processors, while the gateway programs execute in the central processor.

The decision on whether a packet should be forwarded to a gateway program or whether it should only be processed by the dispatcher is based mostly on the TCP flags within the packet. In order for the reasons for this to become clear, it is important to understand the structure of a typical gateway program. Gateway programs start by declaring *flows* of packets on which they wish to operate. A flow is defined by ranges of source and destination IP addresses, and source and destination TCP/UDP

ports. Note that many TCP connections may belong to a single flow. A gateway program may request to receive all packets within a flow, or only packets which have certain TCP flags set (such as SYN, ACK, or FIN). Other packets within the flows declared by the gateway program will be processed by the dispatcher according to instructions given by the gateway program (if any); such packets will not be forwarded to the gateway program. As an example of this approach, consider a gateway program which performs redirection. When declaring the flows, the gateway program will request all packets with a set SYN or FIN flag, and no other packets. By examining the SYN and FIN packets, the gateway program can detect the beginning and end of individual TCP connections within the flows that it declared. Once a SYN packet is received by the gateway program, a redirection decision can be made. The gateway program submits an address translation instruction to the dispatcher followed by the SYN packet. From that point onward, all non SYN or FIN packets belonging to this TCP connection will be manipulated solely by the dispatcher. Since the gateway program monitors the FIN packets, it can detect the end of a TCP connection, and instruct the dispatcher to remove the corresponding address translation.

In addition to address translations, the dispatcher may perform TCP window size translations which may be used for bandwidth management purposes, and TCP sequence number adjustments. These are needed when the gateway program modifies the payload of packets in a way that results in a packet size change. A gateway program for redirecting active FTP traffic is an example of such a program. Since an ASCII IP address and port number information are carried within the payload of some packets, redirection may result in a need to perform packet payload modifications which may result in a need to adjust TCP sequence numbers on all the following packets within the TCP connection.

III. PROGRAMMING INTERFACE

This section provides an overview of the NEPPI programming interface which is used by gateway programs. A description of the programming interface of an earlier version of NEPPI appeared in [2]. This earlier version was geared toward user-level gateway programs, and it did not include support for a number of features which are available in the current version. These features include a hash table implementation, ghost port management, delayed cleanup of gateway program data, and traffic redirection to a TCP port on the switch.

Hash tables are often needed by gateway programs for storing and retrieving data related to individual TCP con-

nections. NEPPI provides support for a hash table indexed by a key composed of connection information (source IP address, source port number, destination IP address, destination port number). NEPPI also provides an efficient way for managing ghost ports which are replacement port numbers for the original port numbers. An example of a gateway program which would use this is a program which performs full network address translation (full NAT). The source IP address on packets coming from the client is changed to the address of the switch, and the packets are sent to the server. When packets arrive from the server, they carry the switch IP address as the destination address, so the switch has to change the destination address to that of the client so that the client accepts these packets. However, the switch cannot know which client should be the target of a packet since that information is now lost. A solution for this problem is for the switch to change the source port in packets arriving from the clients as well as the source IP address. The source port is changed to a ghost port which is unique for each original client and source port pair. The packets arriving in response from the server will carry this ghost port as the destination port. Based on this ghost port, the switch can look up the original client and source port, and perform the appropriate translation on the packets going from the server to the client. NEPPI provides support for the allocation and reuse of ghost ports. We do not elaborate any further here on the support for hash tables and ghost ports. A description of the other recently added features appears below.

Gateway programs are typically implemented as loadable kernel modules. They communicate with the dispatcher (which is also a loadable kernel module) by calling functions exported by the dispatcher. NEPPI supports user-level gateway programs as well. In this case, the dispatcher acts as a device driver, and the gateway programs communicate with the dispatcher by performing reads, writes, and ioctls on this device. The API used by the programmer of a gateway program is identical for both kinds of gateway programs. A NEPPI application may even consist of a user-level program as well as a loadable kernel module. An example of such an application is URL-aware switching with TCP splicing (see section IV-C) where a user-level proxy performs the URL-aware switching, while a loadable kernel module performs the TCP splicing. NEPPI provides support for this scenario as well by providing a library which is linked with the user-level program to provide for communication between the user-level program and the loadable kernel module. A call to a function in this library will be performed, for example, by the proxy program for URL-aware switching after it accepts a TCP connection from a client and establishes

a TCP connection to a server. At this point, the proxy is ready to request the TCP splicing loadable kernel module to splice the two connections together. This library is not discussed further here.

We now elaborate on the most significant parts of the NEPPI programming interface. We start by describing the core data structures and proceed to describe the most important functions.

A. Structures

The core structure definitions for NEPPI are shown in fig. 3, and a description of the structures is now presented:

A.1 `flow_desc_t`

This structure describes the properties of a packet flow requested by a gateway program. `src`, `src_mask`, and `src_ports` contain the source IP address, an address mask designating the significant bits, and the source TCP/UDP port range for the packet flow. `dst`, `dst_mask`, and `dst_ports` similarly describe the destination IP address, address mask, and destination ports.

`protocol` specifies the IP protocol.

`inv_flags` contains the logical or of flags which specify which of the flow properties should be inverted (for example, we may wish to request all packets which do not come from the specified port range.) The following flags can be used: `IP_FW_INV_SRCIP` (invert the source addr/mask), `IP_FW_INV_DSTIP` (invert the destination addr/mask), `IP_FW_INV_SRCPT` (invert the source ports), `IP_FW_INV_DSTPT` (invert the destination ports), `IP_FW_INV_PROTO` (invert `protocol`).

`mark` lets the gateway program assign marks to packet flows so that it can distinguish between packets belonging to different flows just by looking at the marks included with the packets it receives from the dispatcher. A mark must be unique—different flows cannot have the same mark.

`type` can be set to one of three values: `INPUT`, `FORWARD`, or `OUTPUT`. This value determines the stage in the Linux firewall operation when the packets belonging to the flow will be intercepted.

`flags` is the logical or of constants (`F_FIN`, `F_SYN`, etc.) denoting TCP flags. If one of these flags is set in a packet within the flow, the dispatcher will send the packet to the gateway program. In addition to a logical or of constants denoting TCP flags, there are two other possible values for this parameter: `F_ALL` which requests all packets (regardless of the settings of their TCP flags), and `F_NE` which requests all packets with a non-empty payload.

A.2 trans_desig_s

Defines translations to be performed by the dispatcher on behalf of a gateway program. The translations apply to all packets within the flows requested by the gateway program which contain the source IP address, source port, destination IP address, and destination port which appear in the **orig** field of the address translation. The fields of **trans_desig_s** are:

addr_trans: defines an IP address and TCP/UDP port translation from **orig** to **new**. All packets within the flows requested by the gateway program which carry the source IP address, source port, destination IP address, and destination port defined in **orig** will be modified to carry the values defined in **new**. Note that values for the **orig** part of the address translation must be specified by the gateway program for all translations, even for translations which do not include an address translation. **orig** always serves to designate the packets to be translated.

seq_trans, ack_seq_trans: declares TCP sequence number or TCP acknowledgement sequence number translations. These fields declare a starting sequence number (**start_seq_num**), an offset (**seq_offset**), and a flag (**add**) for specifying whether the offset should be added or subtracted. The sequence number in packets carrying a sequence number larger than **start_seq_num** is incremented or decremented by the specified offset. Similarly, the acknowledgement sequence number may be adjusted. Currently, the dispatcher maintains a history of one sequence number (and acknowledgement sequence number) translation. Hence, if the starting sequence number and sequence number offset are modified by the gateway program, the previous values will be kept by the dispatcher for the purpose of applying this older translation to older packets which may still arrive.

win_trans declares a TCP window size adjustment which can be used to perform simple bandwidth management.

which specifies which translations are being declared. It is the logical or of constants corresponding to the translation types (ADDR_TRANS, SEQ_TRANS, WIN_TRANS).

trans_id points to a value provided by the dispatcher. It is used to refer to the translation when it needs to be removed.

B. Function Calls

The function calls used by gateway programs to interact with the dispatcher are as follows:

- **int neppi_startup(process_packet)**

Every gateway program that uses NEPPI is expected to implement a **process_packet()** function. When the gateway program is initiated, it uses the **neppi_startup()** function

```

struct flow_desig_s {
    u_int32_t src;
    u_int32_t src_mask;
    u_int16_t src_ports[2];
    u_int32_t dst;
    u_int32_t dst_mask;
    u_int16_t dst_ports[2];
    u_int16_t protocol;
    u_int16_t inv_flg;
    u_int32_t mark;
    char      type;
    u_int8_t  flags;
};
struct conn_s {
    u_int32_t src_addr;
    u_int16_t src_port;
    u_int32_t dest_addr;
    u_int16_t dest_port;
};
struct addr_trans_s {
    struct conn_s orig;
    struct conn_s new;
};
struct seq_trans_s {
    u_int32_t start_seq_num;
    u_int32_t seq_offset;
    char      add;
};
struct win_trans_s {
    u_int16_t win_offset;
    char      add;
};
struct trans_desig_s {
    struct addr_trans_s addr_trans;
    struct seq_trans_s  seq_trans;
    struct seq_trans_s  ack_seq_trans;
    struct win_trans_s  win_trans;
    u_int8_t            which;
    void                *trans_id;
};

```

Fig. 3. NEPPI structures

to register itself with NEPPI and passes a pointer to the `process_packet()` function that it has implemented. When a packet is to be forwarded to the gateway program, the dispatcher calls this function with the packet as a parameter. The gateway program can then process this packet in any customized way and return it back to the dispatcher. `process_packet` is a pointer to a function which has three parameters:

process_packet(u_int32_t flow_mark, char *ip_pkt, bool *do_trans)

In the first parameter, the dispatcher provides the mark of the flow to which the packet belongs (see section III-A.1). The second parameter points to the IP packet. The value pointed to by the third parameter determines whether the packet should be further manipulated by the dispatcher once `process_packet()` returns. Before `process_packet` returns, it should set this value to true if it is desired that the dispatcher perform further translations on the packet if it matches a previously declared translation. The value should be set to false if the packet should be sent as is.

- **int neppi_start_flow(struct flow_desig_s *flow_desig)**

This function is used by the gateway program when it wishes to request a new packet flow from the dispatcher. The parameter specifies the properties of the packet flow as described in the previous section.

- **int neppi_stop_flow(u_int32_t flow_mark)**

Removes the flow whose mark is specified by the parameter from the set of flows requested by the gateway program.

- **int neppi_send_packet(char *ip_pkt, bool do_trans, u_int16_t rport)**

Sends out a packet. The first parameter points to the IP packet. The second parameter determines whether the dispatcher should perform translations on this packet if it matches a previously declared translation. The third parameter specifies a local TCP port to which the packet should be sent. The value of this parameter is typically zero which means that the packet should not be redirected to a local TCP port. See section II for a description of this feature.

- **int neppi_start_trans(struct trans_desig_s *trans)**

This is used by the gateway program to declare a new translation or to update an existing translation. The translation on the relevant packets will be performed by the dispatcher on behalf of the gateway program.

- **neppi_stop_trans(void *id)**

Removes a translation whose identifier is pointed to by `id`.

- **neppi_delayed_cleanup(void *data, void (*cleanup_f)(void *), int sec, void *cleanup_id)**

This function is used by the gateway program to request that the data pointed to by the first parameter be cleaned `sec` seconds in the future by calling the function pointed

to by `cleanup_f` with a pointer to `data` as a parameter (i.e. `cleanup_f` is scheduled to be called on `data` when a period of `sec` seconds passes.) `cleanup_id` points to an identifier of this cleanup operation (this value is provided by `neppi_delayed_cleanup()`.) `cleanup_id` can be used to cancel the cleanup request using `neppi_stop_cleanup()`. Some data related to TCP connections should be kept until a certain amount of time passes (such as 2MSL—see section IV-A). This function is useful for cleaning up such data at the right time.

- **neppi_stop_cleanup(void *cleanup_id)**

Cancels a previously scheduled cleanup operation whose identifier is pointed to by `cleanup_id`.

IV. IMPLEMENTING LAYER 4/7 SERVICES

In this section, we show how a number of layer 4/7 services were implemented using NEPPI.

A. Layer 4 switching of TCP traffic

Layer 3 (or IP) routers make routing decisions based on IP addresses alone. Layer 4 switches still route packets based on IP addresses, but they also use information contained in the TCP/UDP header to make routing decisions. TCP/UDP headers contain protocol specific source and destination ports. Well-known applications run on designated port numbers (e.g. FTP servers run on TCP port 21, and HTTP servers run on port 80.) Layer 4 switches can handle and redirect requests to different services differently by looking at the TCP/UDP port numbers on the IP packets. A layer 4 switch manages a number of back-end servers which provide a certain service. Normally, the switch is assigned a virtual IP address which is registered with the DNS server. Clients make a request to this virtual IP address which is then redirected to an appropriate server. There are a number of products available in the market that perform layer 4 redirection. These include Alteon Networks' ACEdirector, Arrowpoint's Content Smart Switch, Foundry Networks' ServerIron Server Load Balancing Switch, Cisco Systems' Local Director (and Distributed Director), and several others [10], [11], [12], [13], [14].

We have used NEPPI and its API to implement a layer 4 switch gateway program which acts as a virtual server. The gateway program uses the `neppi_start_flow()` function call to instruct the dispatcher to forward to it packets that are destined to TCP ports for well-known services such as FTP (port 21), telnet (port 23), rlogin (port 513), rsh (port 514) and HTTP (port 80). Specifically, the dispatcher is instructed to only forward packets that have the SYN flag or the FIN flag set. The `flow_desig_s` data structure is set up appropriately to accomplish this. The forwarding of

packets from NEPPI to the gateway program is through a call-back function (see the description of `neppi_startup()` in section III-B. When the SYN packet for a TCP connection is thus received at the gateway program, it selects a physical server to serve this request based on some load balancing scheme. The server selection could be based on the header information on the SYN packet; for example, requests from a specific set of clients could be sent to a specific set of servers. The gateway program then instructs the dispatcher to redirect further packets on this connection from that client to the chosen server and the packets that come back from the server to the client after performing necessary address translations on the packets in both directions. This is accomplished by setting up the `trans_design` structure appropriately by specifying the original and new addresses and calling the `neppi_start_trans()` function for packets flowing in each direction separately. For the packets going from the client to the server, the dispatcher is instructed to change the source IP address on the packets to that of the virtual server IP address, and the destination address to that of the chosen physical server. For the packets going from the server to the client, the dispatcher is instructed to change the source address to the virtual IP address and the destination address to that of the client. This way, it will appear to the client that it is receiving a response from the virtual IP address to which it sent the request. Thus, full NAT is required to be performed by the dispatcher. IP checksums are recalculated by the dispatcher after the full NAT is performed. Performing full NAT allows the physical servers to be distributed anywhere on the wide-area network and does not constrain in any way their location relative to the location of the switch. The dispatcher manages the connection from beginning to end and provides a fast path for the packets that are not seen by the gateway program (in this case, all packets except the SYN and FIN packets).

The gateway program needs to keep connection information so that it can request the dispatcher to remove the corresponding address translations when the FIN packets are received. This connection information is maintained in a *connection control block (CCB)*. Library functions are provided as part of NEPPI for hashing, searching, updating, and cleaning up the CCB data structures. The library functions provided for cleaning up the CCBs merit attention. Two functions `neppi_delayed_cleanup()`, and `neppi_stop_cleanup()` facilitate cleaning up the CCBs and hence the connection state at the gateway program. When FIN packets are received at the gateway program, the connection information cannot be cleaned up immediately since a 2MSL time-out period is specified in the TCP requirements. When the FIN packets going in both

directions are observed at the gateway program, it calls the `neppi_delayed_cleanup()` function with a 2MSL time-out period, a CCB pointer, and a call-back destructor function pointer as parameters. This instructs the dispatcher to call the destructor function referenced by the function pointer after the 2MSL time-out expires and pass back the CCB pointer so that the connection information can be removed at the gateway program. At this time, the gateway program also instructs the dispatcher to remove the address translations belonging to this connection using the `neppi_stop_trans()` function call. Sometimes, the connection information can be cleaned out immediately. This can be done, for example, when an RST packet is observed going from the client to the server or vice-versa. For this purpose, the `neppi_delayed_cleanup()` function can be used with a time-out value of zero. The gateway program should use the `neppi_delayed_cleanup()` even if the connection state is to be cleaned up immediately. This is because NEPPI guarantees that the call-back function passed as a parameter is executed atomically and interrupts are disabled during that time. If the gateway program removes connection state by itself, it is possible that data structures get corrupted. The gateway program also uses the `neppi_delayed_cleanup()` function to clean up connection state for connection attempts which fail during the initial three-way TCP handshake. When a SYN packet is received at the gateway program, it creates a CCB and calls `neppi_delayed_cleanup()` now with a TCP connection time-out as a parameter. If the connection request is unsuccessful within the time-out period, the connection information is removed through the call-back destructor function. If the connection request is successful, the `neppi_delayed_cleanup()` function call that was made should be undone. This is accomplished through the use of the `neppi_stop_cleanup()` function. Calling this function removes the request for delayed cleanup of this connection at the dispatcher.

B. Transparent switching of HTTP traffic to proxy caches

The discussion above illustrated how NEPPI can be used to perform header manipulations and forwarding of packets on a fast path. In this section, we discuss a layer 7 service where payload modifications on IP packets are performed and how NEPPI is used to support this functionality.

Proxy caching is used to decrease both the latency of object retrieval and traffic on the Internet backbone. In general, web browsers have to be configured such that requests are channeled through a proxy. Recently, a number of cache server products which support transparent caching appeared in the market [15], [5], [16]. Using these prod-

ucts, object requests can be served by the cache in a way which is transparent to the web browser (i.e. no proxy setting is required.) We refer to these products as *network caches*. These need to be located on the route that the client request takes going from the browser to the origin server, or a Layer 4 switch needs to intercept the requests and redirect them to the network cache. In case a Layer 4 switch is used, the network caches are either connected to the switch or within one hop of the switch. Proxy servers (which we refer to as *proxy caches*) handle requests differently from network caches in the following way. When the browser is configured to use a proxy cache, the browser first makes a TCP connection to the proxy cache and then sends an *absolute URL* of the required object. The absolute URL includes the origin server name and the path to the required object from the document root at the server. The absolute URL, including the origin server name, is required because the packets that go from the browser to the proxy contain the proxy's IP address as the destination address. If only a *complete URL* (where the path provided does not have the origin server name but only the path relative to the document root) is provided, the proxy will not know the origin server name or IP address.

When a browser is not configured to use a proxy, it retrieves an object as follows: the browser first makes a TCP connection to the origin server and then sends only the complete URL. A complete URL is sufficient as the connection is already established with the origin server and it is understood that the origin server serves documents by parsing URLs relative to its document root. The Layer 4 switches in the market, which are designed to redirect transparently such client requests, blindly redirect them to the caches, which means that the HTTP GET requests still carry complete URLs. Network caches act in promiscuous TCP mode where the packets can still carry the origin server's IP address as the destination. Hence, the network cache will have the origin server's IP address and it can make a connection to the server to retrieve an object if it is not available locally. A complete URL is sufficient in this case. But a standard application-level proxy cannot expect this feature to be provided on the system that it is run on. Thus, to make standard proxies support transparent caching, we have implemented a transparent HTTP switch gateway program which extends the complete URL to an absolute URL by processing packets at the IP level. Further details on this technique can be found in [4]. Our goal here is to show how the NEPPI API is used to accomplish payload modification at the switch.

The gateway program uses the **neppi_start_flow()** function to request the dispatcher to forward to it SYN and FIN packets as well as packets that contain a non-empty

payload which are destined to port 80 (the standard HTTP port). When a SYN packet is received, the HTTP switch selects a proxy cache as the target for a transparent redirection of this request. At this time, the **neppi_start_trans()** function is used to specify that all other packets (mainly ACK packets) be processed on the fast path (i.e. at the dispatcher) by specifying proper address translations. When the packet containing the HTTP GET request is received from the client (this will be the first payload packet from the client), the gateway program processes this packet and transforms the complete URL to an absolute URL by prefixing the origin server IP address to the complete URL¹. The origin server IP address is available as the destination address on the IP packets. Given that the payload has been modified, the TCP checksum has to be recalculated. The **neppi_inet_csum()** function provided by NEPPI is used by the gateway program to calculate this checksum.

To make this URL transformation transparent to the endpoints, IP and TCP header changes are required. Because the length of the IP packet that carries the GET request is now increased, the total length field on the IP header of this packet is increased by an offset. In addition, the TCP header contains sequence numbers (*seq*) and acknowledgment sequence numbers (*ack_seq*) that need translation. The *seq* on the TCP header indicates the byte number of the first byte on this packet going from the sender to the receiver over the TCP session, while the *ack_seq* indicates the byte number of the next byte that the sender expects to receive from the receiver. For all packets after the GET packet(s) that go from the client to the server, the *seq* needs to be increased by an offset ($lengthof(\text{absolute URL}) - lengthof(\text{complete URL})$) so that the *seq* matches the byte number of the byte that the server expects to receive from the client. Similarly, on all packets starting with the acknowledgment to the GET packet that go from the server to the client, the *ack_seq* needs to be decreased by the same offset so that the *ack_seq* matches the byte number of the byte that the server should expect the client to send in the next packet following the GET packet. Performing the above changes to the header makes the TCP endpoints unaware of the change to the GET packets.

The gateway program uses the **neppi_start_trans()** function in order to specify the *seq* and *ack_seq* modifications that need to be performed on subsequent packets flowing in either direction. The **trans_desig_s** structure is set up with the offset that needs to be added to the *ack_seq* number on packets going from server to client and the off-

¹The HTTP GET request may be carried in multiple IP packets; this situation is handled in our implementation, but the details are beyond the scope of this paper.

set that needs to be subtracted from *seq* on the packets that go from client to server. Starting *seq* and *ack_seq* numbers are provided so that the dispatcher does not perform these translations on older packets. From then on, the dispatcher handles all packets within the fast path through appropriate address and sequence number translations. Maintaining TCP connection information as well as cleaning up this information is performed using the NEPPI function calls as described in the previous example.

The next example illustrates some additional functionality available in NEPPI that allows more complicated gateway programs to be implemented. Specifically, this example illustrates how application-level programs can interact with NEPPI through gateway programs and also shows how NEPPI can be used to inject packets generated by gateway programs into the network.

C. URL-aware switching

URL-aware switching (also known as “content-smart switching” [11]) refers to the capability of a switch located in front of clients or servers to redirect HTTP requests to servers based on the URL specified by the client in its GET request. When a user enters a URL into a browser, the browser constructs an HTTP GET request which contains the URL and other HTTP client header information. With URL-aware redirection, a switch located on the path between the client and the servers will intercept the request and use the information within the request to make a decision about the server to which the request should be directed. All this happens transparently to the client. Switches that switch traffic based on URL and other application-level information are also referred to as layer 7 switches.

Redirecting HTTP traffic based on application-level (layer 7) information is not as simple as layer 4 switching. For any HTTP transaction, application-level information is not available until the TCP connection establishment phase has been completed. This means that connections cannot be redirected at a switch by simply peering into a SYN packet as is possible with basic layer 4 switching. The TCP connection request from the client needs to be accepted at the switch by an application-layer proxy, and the connection must be established between the client and the switch before any application-level information can be received. Once the application-level information is received, this information is parsed to determine which back-end server should receive this request and the request is redirected. One approach for redirection is the TCP gateway approach where another TCP connection is established between the switch and the back-end server, the client request is passed to the server through that connection, the

response is received at the switch from the server on the connection and transferred through the other connection to the client. To speed up the performance of TCP gateways, the TCP splicing mechanism has been proposed [8]. In this approach, once the two TCP connections are established, they are “spliced” or “patched” together so that IP packets are forwarded from one TCP connection to the other at the network layer without having to traverse the TCP layer to the application level. This requires that appropriate address translations and sequence number modifications be performed on the packets. For example, packets arriving on the connection from the server to the switch that are to be forwarded to the client should be translated so that the addresses and sequence numbers on these packets match the ones that would have been found on the corresponding packets if the application-layer proxy had received this data and then put the data back on the TCP connection from the switch to the client.

We have implemented a URL-aware switch that incorporates TCP splicing using NEPPI. There are two components, other than NEPPI, to this implementation. The first component is an application-level proxy (*proxy-s*) that accepts TCP connections from the clients, parses the GET requests, determines the destination server, and establishes a connection to that server thereby enabling URL-aware redirection. The second component is the *splice* gateway program that “splices” the TCP connections together. The gateway program transparently intercepts HTTP request packets from the clients destined to the origin servers, and sends them up the protocol stack to *proxy-s*. It accomplishes this by instructing the dispatcher using the **neppi_start_flow()** function call to forward to it *all* packets destined to port 80. It also instructs the dispatcher using another **neppi_start_flow()** function call to forward to it all packets on the connection from *proxy-s* to the server. It further uses a special functionality of NEPPI which provides for a redirection of packets to a local TCP port regardless of their destination address instead of performing forwarding. In this case, the packets are redirected to the TCP port where *proxy-s* is listening. The gateway program monitors the state of the connection between the client and *proxy-s* and records TCP *seq* and *ack_seq* numbers on the GET packet(s) going from the client to *proxy-s*; these will be used later for splicing connections. It further monitors the connection between *proxy-s* and the chosen server and records *seq* and *ack_seq* numbers on the GET packets from *proxy-s* to the server. Just before *proxy-s* sends the message containing the GET request to the server, it sends a *patch* command to the gateway program. The *patch* command contains the TCP endpoint information for each of the two connections. For the connection between the client

and *proxy-s*, it contains the IP address and TCP port number of the client and the IP address and TCP port number where the client request was received on the switch itself (the local endpoint). For the connection between *proxy-s* and the server, it contains the IP address and port number of the local endpoint from which the connection was established to the server and also the server IP address and port number. As part of the NEPPI infrastructure, a library is provided for communication between user-level programs such as *proxy-s* and gateway programs which are loadable kernel modules.

As the gateway program is monitoring both connections and recording relevant information, the above information contained in the *patch* command is enough to identify the connections to be spliced. When the first data packet arrives from the server, the gateway program uses the `neppi_start_trans()` function call to instruct the dispatcher to patch the two connections together. The `trans_desig_s` structures are set up so that the destination addresses on the packets going from the server to *proxy-s* are changed to that of the client and the *seq* and *ack_seq* numbers are re-mapped to those that would have been found in corresponding packets if these packets had been received by *proxy-s* and put on the TCP connection to the client. Information needed to re-map the *seq* and *ack_seq* numbers is calculated by the gateway program using the *seq* and *ack_seq* numbers on the GET packets that were recorded earlier. Thus, all data packets from the server to *proxy-s* are redirected at the network level to the client. Similarly, the dispatcher is instructed to perform address and sequence number translations on the acknowledgment and other packets from the client before they are sent to the server. The source address is changed so that the packet appears to the server as if it was originated by *proxy-s*. The *seq* and *ack_seq* numbers are re-mapped to the appropriate numbers to appear as if they were sent by *proxy-s* on its established connection to the server. Once the splice is performed, the two TCP endpoints at *proxy-s* are closed. To achieve this, RST packets are locally generated by the gateway program masquerading as the client or the server (depending on which endpoint needs to be closed) and sent to *proxy-s*. The packets are appropriately constructed with the correct source and destination addresses and sequence numbers. The `neppi_send_packet()` function is used for this purpose. This function sends out packets locally generated at a gateway program; it takes the packet and sends it onto the network (or in this case, up the protocol stack on the local machine) while bypassing the NEPPI packet filtering mechanism. Note that the packet filter needs to be bypassed—otherwise it will send these packets back to the gateway program as it has been instructed to send all

packets originating from the server destined to *proxy-s* to the gateway program.

When the data transfer has been completed, either the server or the client can initiate an active close. In either case, the FIN packets and the corresponding ACKs are still spliced so that the endpoints at the client and the server, each representing one endpoint of each connection, are closed cleanly. The TCP connection information is maintained as well as cleaned as described in the earlier examples. More details on the URL-aware switch implementation can be found in [3].

V. RELATED WORK

There is work both in academia and in industry which is related to the work presented in this paper. Most of the related academic work is in the context of *active networks* which are networks containing programmable switches or routers. A distinction is made in [1] between two types of approaches to programming the nodes of active networks: *active packets* (also known as *capsules* [9]), and *active extensions*. The active packets approach is based on sending packets containing both a program and data, while the active extensions approach is based on sending the programs to the nodes separately from the passive data packets on which they operate. NEPPI can implement active extensions, and the applications of NEPPI mentioned in section IV can be viewed as active extensions. NEPPI is tailored for layer 4/7 switching applications, but other types of active extensions are possible as well since gateway programs can obtain all packets within specified flows, and process them in any desired way.

A software architecture called *router plug-ins* for dynamically loading packet processing code modules (plug-ins) into the kernel is described in [6]. A plug-in is somewhat analogous to a NEPPI gateway program. Similarly to gateway programs, different plug-ins may be bound to different flows. The purpose of the plug-ins is to support a modular and extensible network subsystem. Plug-ins implementing IPv6 options, packet scheduling, packet classification and routing, and IP security are mentioned in [6]. NEPPI gateway programs operate at a higher level than router plug-ins since the NEPPI dispatcher (implemented in the kernel or in hardware) provides facilities for performing common packet manipulation tasks such as address translations and TCP sequence number modifications on behalf of gateway programs.

There are relatively few research publications addressing layer 4/7 switching issues. A layer 4 redirection mechanism which involves a special redirection protocol which needs to be supported by both the clients and the servers is introduced in [7]. This is in contrast to the layer 4/7

applications we describe here which operate at the switch transparently to the clients and servers and without requiring special support at the endpoints.

Most of the work on layer 4/7 switching has been done in industry; mostly in start-up companies. Naturally, this work is only partially described in white papers and product literature. Typically, the capabilities of the products are described, but not the mechanisms used for implementing these capabilities. Some references for such product literature are included at the end of the bibliography. All these products include support for layer 4 switching (described in section IV-A) and some support URL-aware switching (described in section IV-C). The mechanisms used internally by the switch to provide this functionality are not published. We are not aware of a product supporting transparent caching with standard proxy caches (described in section IV-B).

VI. CONCLUSIONS

We added support for programmable packet processing to the Linux OS. This support is tailored for implementing efficient layer 4/7 switching functionalities. A loadable kernel module called the dispatcher provides services such as packet interception and injection, address translations, and sequence number translations. Other loadable modules called gateway programs use these services for the purpose of providing new kinds of switching capabilities. The dispatcher provides a “fast path” where packets may be manipulated on behalf of gateway programs. Most packets are processed in this fashion, while some packets are processed by the gateway programs themselves. Packets processed by the gateway programs include packets signifying the establishment or termination of TCP connections, and packets whose payload must be examined or changed by the gateway programs. The dispatcher functionality could potentially be implemented in hardware or in network port processors to achieve a higher level of performance than possible with a PC-based architecture. The paper focused on the implementation of three services using our infrastructure: a layer 4 application (TCP traffic redirection) and two layer 7 applications (transparent switching of HTTP traffic to standard proxy caches, and URL-aware switching).

REFERENCES

- [1] Alexander, D.S.: ALIEN: A Generalized Computing Model of Active Networks. PhD thesis, University of Pennsylvania (1998)
- [2] Cohen, A., Rangarajan, S.: A Programming Interface for Supporting IP Traffic Processing. In: Proceedings of the 1st International Working Conference on Active Networks (1999)
- [3] Cohen, A., Rangarajan, S.: On the Performance of TCP Splicing for URL-aware Redirection. To appear in: Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (1999)
- [4] Cohen, A., Rangarajan, S., Singh, N.: Supporting Transparent Caching with Standard Proxy Caches. In: Proceedings of the 4th International Web Caching Workshop (1999)
- [5] Danzig, P.: NetCache Architecture and Deployment. <http://www.networkappliance.com/technology/level3/3029.html>. Presented at the 3rd International WWW Caching Workshop (1998)
- [6] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B.: Router Plugins: A Software Architecture for Next Generation Routers. In: Proceedings of ACM SIGCOMM (1998) 229–240
- [7] Gupta, S., Reddy, N.: A Client-oriented IP Redirection Mechanism. In: Proceedings of IEEE INFOCOM '99 (1999)
- [8] Maltz, D.A., Bhagwat, P.: TCP Splicing for Application Layer Proxy Performance. IBM Research Report RC 21139 (1998)
- [9] Wetherall, D.J., Guttag, J.V., Tennenhouse, D.L.: ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In: Proceedings of IEEE OPENARCH (1998) 117–129
- [10] “ACEdirector”, <http://www.alteon.com>
- [11] “The Content Smart Internet”, <http://www.arrowpoint.com/solutions/whitepapers/CSI.asp>
- [12] “ServerIron Switch”, <http://www.foundrynet.com/serverironspec.html>
- [13] “Cisco LocalDirector”, <http://www.cisco.com>
- [14] “Cisco DistributedDirector”, <http://www.cisco.com>
- [15] “Deploying Transparent Caching with Inktomi’s Traffic Server”, <http://www.inktomi.com/products/traffic/tech/deploy.html>
- [16] “High-Performance Web Caching White Paper”, http://www.cacheflow.com/technology/wp/hp_cache.html
- [17] “Internet Commerce Appliances”, <http://www.ipivot.com>