Modular Components for Network Address Translation

Eddie Kohler, Robert Morris, and Massimiliano Poletto
MIT Laboratory for Computer Science
{*eddietwo, rtm, maxp*}*@lcs.mit.edu*

Abstract

We present a general-purpose toolkit for network address translation in a modular, component-based networking system. Network address translation is a powerful, general technique for building network applications, such as allowing disparate address realms to communicate, load balancing among servers, and changing ordinary proxies into transparent proxies. The components of our toolkit can be combined in a variety of ways to implement these applications and others. The context of this work, the Click modular networking system, makes the NAT components simpler and more understandable. For example, the NAT components concern themselves solely with address translation; related functions, such as classification, are implemented by separate components. This design is more flexible than most existing NAT implementations. The user can choose where network address translation takes place in relation to other router functions, and can use multiple translators in a single configuration. These components have been in use in a production environment for several months.

We describe our design approach, demonstrate its flexibility by presenting a range of examples of its use, and evaluate its performance.

1 Introduction

Network address translation (NAT), web server load balancing, and redirecting traffic to transparent proxies all share a common feature: they involve a level of indirection in the meaning of IP addresses and port numbers, and can be implemented by rewriting those values in IP headers and payloads. Usually these tasks are carried out in specialized routers tailored to the particular task, with limited configurability beyond that task.

This paper presents a general-purpose toolkit for rewriting packets to provide a level of indirection in IP addresses and port numbers. Individually, each toolkit component is not necessarily more powerful than other existing devices that perform specific address translation or traffic redirection tasks. However, the ability to combine these components into new arrangements makes the component-based system more flexible than any single monolithic system. The particular address translation functions mentioned above fall out as special cases of our general rewriting framework, suggesting that the framework will prove useful for other tasks as well.

Our rewriting toolkit is implemented as a family of *elements* in the Click [8] modular router. The rewriting elements divide into three categories: *translation elements*, such as *IPRewriter*, which actually perform network address translation and modify packet headers; *mapping helpers*, such as *RoundRobinIPMapper*, which help translation elements create new mappings that define how unknown flows should be rewritten; and *application-level gateways*, such as *FTPPortMapper*, which perform packet manipulations required to help specific protocols pass through a NAT. These elements can be combined in many ways to provide almost arbitrary packet rewriting functionality.

The next section gives some background on network address translation. Section 3 provides an overview of the Click system. Section 4 describes the design and implementation of the family of rewriting elements. Section 5 presents a variety of examples of how the rewriter can be used. Section 6 analyzes its performance. Section 7 discusess related work, and Section 8 concludes.

2 Network address translation

Network address translation was originally designed to help ease the demand for IP addresses. For a set of machines on a local network, RFC 1631-style, or "Basic", NAT [5] reserves both a large set of private IP addresses and a small set of public IP addresses. Each machine gets a permanent private IP address; local machines can communicate with one another using these addresses. However, they cannot directly communicate with the Internet at large, where the private addresses are meaningless. Therefore, all packets to and from the Internet are expected to travel through a translator. When a local machine sends a packet to the Internet, the translator assigns that machine a public IP address from the

This research was supported by a National Science Foundation (NSF) Young Investigator Award and the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory under agreement number F30602-97-2-0288. In addition, Eddie Kohler was supported by a National Science Foundation Graduate Research Fellowship.

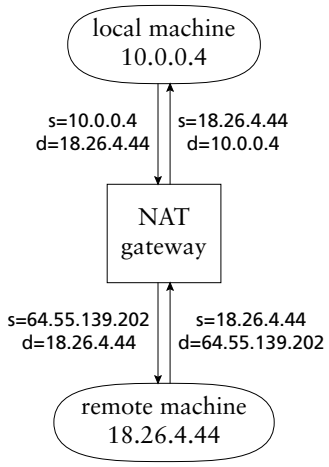

FIGURE 1—Basic NAT, as described in RFC 1631 [5]. The local machine has a private IP address. When it sends packets to an Internet host, the NAT gateway assigns it an address from a public address pool and rewrites the packets accordingly. Replies are rewritten to use the original private IP address.

pool. It rewrites the IP packet to use this public address, and saves the private–public mapping in a table. When a packet from the Internet arrives that is destined for one of the public addresses, the translator looks in its table for a mapping for that address, rewrites the packet to use the corresponding private address, and sends the packet on its way. Thus, local machines transparently gain a public address on a temporary basis. Mappings are garbage-collected after some period of inactivity, making public addresses available for reallocation. If there are n public addresses, then at most n local machines can access the Internet simultaneously. Figure 1 illustrates this structure.

Basic NAT is one instance of a more general idea. General network address translators sit in the flow of packets and change packets' *flow identifiers*. A flow identifier consists of a packet's protocol, source address, destination address, and optionally protocol-specific information such as source and destination ports, or even URLs for HTTP. A network address translator expects to receive bidirectional flow; that is, packets from the local network to the Internet, and packets sent from the Internet in response. It generally maintains a mapping table describing how to rewrite packet headers so that rewritten packets maintain correct session semantics. Finally, some application protocols, such as FTP, contain IP addresses in their data streams. These protocols are not transparent to translation, and require special handling by the translator, often in the form of an application-level gateway [5, 14].

Thus defined, network address translation is a powerful abstraction for manipulating traffic. Translators can allow communication between disparate address realms [13]; Basic NAT lets a network with privately allocated addresses communicate with the Internet at large, while other forms let IPv6-only and IPv4-only hosts interoperate [9, 15]. Translators can also protect a local network from intrusion, similar to a firewall, or load-balance requests between a set of servers [12]. Other uses have been proposed, such as creating network redundancy [7].

NAT has disadvantages too, such as adding points of failure (due to the state translators must generally maintain), breaking the uniqueness of IP addresses, and violating the Internet's end-to-end philosophy [6]. IPv6 addresses these problems, and some recommend that NAT be abandoned for IPv6. However, NAT is undeniably useful on today's Internet; it provides an interesting abstraction for building network applications; and it will even be used to ease the transition to IPv6. A truly flexible and modular NAT implementation may ease some of NAT's difficulties by making its use and deployment more configurable and understandable. Our goal has been to build such an implementation in the context of a modular router toolkit, Click.

3 Click

Click routers are built from components called *elements*. Elements are modules that process packets; they control every aspect of router packet processing. Router configurations are directed graphs with elements as the vertices. The edges, called *connections*, represent possible paths that packets may travel. Each element belongs to an *element class*, which determines its behavior by setting the code executed when the element processes a packet. Each element also has an optional *configuration string*, which element classes can use to select behavior more precisely. For example, the *Tee* element class duplicates packets; a *Tee* element's configuration string, an integer, says how many copies to make. Inside a running router, elements are represented as C++ objects and connections are pointers to elements. A packet transfer from one element to the next is implemented with a single virtual function call.

Elements also have *input* and *output ports*, which serve as the endpoints for packet transfers. Every connection leads from an output port on one element to an input port on another. An element can have zero or more of each kind of port. Different ports can have different semantics; for example, the second output port is often reserved for erroneous packets.

Every queue in a Click configuration is explicit. Thus, a configuration designer can control where queueing takes place by deciding where to place *Queue* elements. This enables valuable configurations like a single queue feeding multiple interfaces. It also simplifies and speeds up

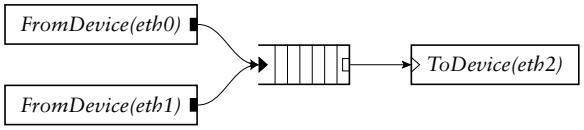

FIGURE 2—A sample Click configuration.

packet transfer between elements, since there is no queueing cost by default.

The Click system includes a simple, declarative language for describing router configurations. The language specifies how elements should be connected together. To configure a router, the user creates a Click-language file and passes it to the system. The system parses the file, creates the corresponding router, tries to initialize it, and, if initialization is successful, installs it and starts routing packets with it.

Figure 2 shows a simple Click configuration. Its effect is to read packets from the network interfaces named *eth0* and *eth1*, append them to a queue, and transmit them out interface *eth2*.

Figure 3 shows a basic 2-interface IP router configuration, the starting point for the various configurations described in the remainder of this paper. This configuration implements all required IP forwarding functionality [1]; see [8] for a complete description of how it works.

Click router configurations run in a downloadable Linux kernel module.

4 Architecture

A flexible NAT implementation should give the user full control over (1) the choice of packets to be translated; (2) the type of translations to be performed—for example, the choice of flow identifier; (3) the addresses and port numbers to use for translated packets; (4) convenient interfaces for use by application-level gateways (which support protocols like FTP); and (5) the placement of translation in relation to other packet processing tasks. Existing NAT implementations generally address a subset of these requirements. Cisco routers, for example, address (1), (2), and (3) to greater or lesser degrees, but do not provide programming interfaces for additional application-level gateways or allow the user to determine when translation should be performed [2]. On the other hand, Cohen and Rangarajan's NEPPI system [4] is designed to support flexible application-level gateway processing—requirement (4)—but supports the other requirements less thoroughly, and requirement (5) not at all. In general, existing NAT implementations only sometimes satisfy requirement (4), and very rarely satisfy requirement (5). That is, interfaces for application-level gateways are often difficult to use or unpublished, and users cannot control when and where translation is per-

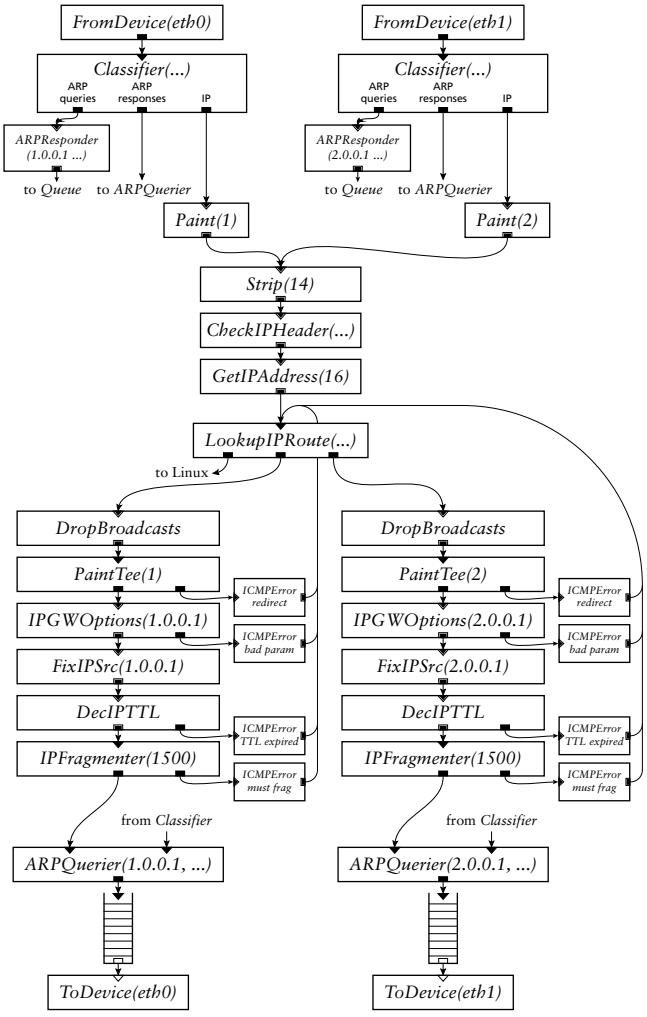

FIGURE 3—An IP router configuration.

formed. A NAT system that does not satisfy these requirements may be useful for a single task, but it will not be easy, or even possible, to use it for other tasks, and its operation in a complex network context may be difficult to analyze or understand.

We have addressed all these requirements by implementing NAT with flexible IP rewriting elements in Click. This section describes those elements and demonstrates their flexibility in theory. The next section demonstrates their flexibility in practice by showing them in context of real network applications.

The IP rewriting elements address the five requirements in the following ways:

1. Choice of packets to be translated: Click IP rewriters are elements—packet processing components; they translate only the packets that pass through them in the router configuration graph. Thus, users can determine which packets to translate by choosing which packets to send through a rewriter. This choice is expressed by other elements in the configuration. It can be based on conventional criteria, such as arrival interface or source address, or arbitrary other criteria, such as type-of-service value or packet contents.

2. Type of translations to be performed: Different Click IP rewriters perform different kinds of translations; the user chooses a translation type by choosing an element. Multiple rewriting elements, and therefore multiple kinds of translation, can coexist in one configuration.

3. Addresses and port numbers to use for translated packets: These are determined by IP rewriter elements' input ports and configuration strings. Each input port corresponds to one configuration argument; packets arriving on that input port are translated as that argument directs. Arguments can delegate the translation decision to user-supplied elements, making translation extensible.

4. Convenient interfaces for use by application-level gateways: The IP rewriters share a common interface for accessing their internal tables, which is available both to other elements, and to user-level programs through `ioctl` commands.

5. Placement of translation relative to other packet processing tasks: Users may place IP rewriter elements wherever they would like in Click router configurations. Again, multiple rewriters with distinct tables can naturally coexist.

Finally, we mentioned how the addresses and port numbers used for a translated flow depend on the input port on which that flow was first encountered. In addition, IP rewriting elements demultiplex packets onto different *output* ports, depending on the mappings specified by the user. This makes it easier to use the rewriting elements in real routers. Users can specify that different kinds of packets—for example, rewritten packets destined for the Internet, as opposed to rewritten packets destined for the local network—should be emitted on different output ports.

4.1 Structure

Click IP rewriting elements divide into three categories. First, *translation elements*, such as *IPAddrRewriter*, *IPRewriter*, *TCPRewriter*, and *ICMPPingRewriter*, actually perform network address translation. They maintain mapping tables for active flows. As packets arrive, the elements use their flow identifiers to find corresponding mappings in the tables. When mappings are found, the elements translate the packets' headers accordingly. When no mappings are found, the elements create new mappings as specified by the *rewrite patterns* in their configuration strings.

Second, *mapping helpers*, such as *RoundRobinIPMapper*, help translation elements create mappings for unknown flows. Users can write arbitrary policies for new mappings by creating mapping helper elements. For example, *RoundRobinIPMapper* takes a set of addresses, and chooses from them in round-robin order as it receives requests for new mappings. Other elements could implement weighted round-robin selection or even selection based on packet characteristics.

Third, *application-level gateways*, such as *FTPPortMapper* and *ICMPRewriter*, perform the packet manipulations required to help particular protocols pass through a NAT. They use interfaces on translation elements to analyze or change existing translations.

4.2 Mapping tables and flow identifiers

Translation elements change packet headers based on the packets' flow identifiers. The definition of "flow identifier" depends on the particular translation element in question. For example, for *IPAddrRewriter*, which implements Basic NAT, flow identifiers consist of protocol, source address, and destination address; *IPRewriter* adds TCP/UDP source and destination ports.

The translation element's *mapping table* relates flow identifiers to structures called *mappings*. A mapping specifies exactly how a packet's header should be changed. It may include a new flow identifier, deltas for incrementally updating IP and TCP/UDP checksums, deltas for TCP sequence numbers, the output port to use for corresponding packets, and so forth. The exact

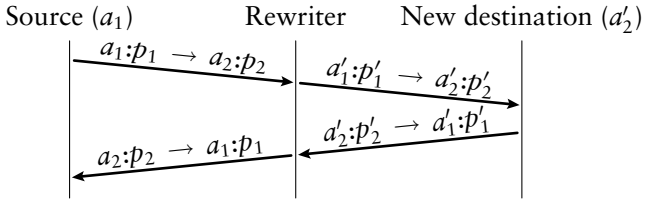

FIGURE 4—Action of a generic NAT with port translation. Each arrow represents a packet with the given flow ID.

contents of a mapping are determined by the translation element of which it is a part.

When a packet arrives at a translation element, its flow identifier is extracted and the mapping table is searched for a corresponding mapping. If there is a mapping, the element will change the packet as that mapping requires and send the modified packet to some output port. However, some packets, which we call *fresh*, do not correspond to a mapping. These packets represent new sessions. The action taken for a fresh packet depends on the input port on which it arrived, as described in the next subsection.

A machine *s* sends packets to a machine *d* generally in expectation of receiving a reply. Network address translators must properly handle these reply packets. Mappings are therefore installed in pairs: one mapping for the flow identifier, another mapping for replies' flow identifier. For example, Figure 4 illustrates a generic NAT with port translation. The central rewriter will have mappings for $a_1{:}p_1 \rightarrow a_2{:}p_2$, the original flow ID, and for $a'_2{:}p'_2 \rightarrow a'_1{:}p'_1$, for replies to the rewritten flow ID.

Translation elements must also have a policy for cleaning old mappings from their tables. These policies might include examining packet data, such as TCP FIN flags, to detect session close, as well as arbitrary timeouts. Click's translation elements currently implement variable timeouts; we will soon add TCP-specific session close detection.

4.3 Rewrite patterns

A central abstraction for Click rewriting elements is the *rewrite pattern*. These are text strings, included in elements' configuration strings, that specify how packets should be translated. Several forms encapsulate common translation directives—for example, that all new flows should be dropped, or that their source addresses and ports should be rewritten. A pattern can also delegate its responsibility to a user-specified mapping helper element. Thus, rewrite patterns are arbitrarily extensible.

Each input port on a translation element corresponds to a single rewrite pattern. That pattern determines how fresh packets arriving on that input port are translated. Patterns are irrelevant for non-fresh packets.

Click's rewrite patterns include:

- '*drop*'. Fresh packets are dropped.
- '*nochange*'. Fresh packets are passed through the translator element without installing any new mappings.
- '*keep*'. The rewriter installs a pair of mappings—one for the fresh packet's flow ID, one for its corresponding reply flow ID—that leave the packet headers unchanged. Thus, future packets with either the input flow ID or its reply flow ID will be passed through the rewriter even if they arrive on an input port with, for example, a '*drop*' rule.
- '*pattern* A_1 P_1 A_2 P_2' (*IPRewriter* and *TCPRewriter* only). Creates a pair of mappings that change the fresh packet's flow ID according to the pattern. The pattern has four parts: a new source address A_1, new source port P_1, new destination address A_2, and new destination port P_2. Any of these parts can be a dash '–', which means "leave unchanged". Thus, the pattern '*1.0.0.1 – 1.0.0.2 –*' will set the packet's source and destination addresses but leave the ports as they are.

  The source port specification can also be a range of ports 'P_L–P_H', in which case the rewriter will choose a port between P_L and P_H. It will also ensure that any two active mappings created by this pattern have different source ports. The pattern '*1.0.0.1 1024–65535 1.0.0.2 80*', for example, sets every fresh packet's source address to 1.0.0.1, destination address to 1.0.0.2, and destination port to 80. The new source port, however, will differ for any two active sessions. Therefore, the new source port uniquely identifies an session, and every reply packet can be mapped back to a unique flow ID.

  Source ports are only unique within the context of a single rewrite pattern. That is, different patterns may allocate the same source port for two simultaneously active sessions, even if both patterns are part of the same translation element.
- '*pattern* A_1 A_2'. This limited version of *pattern* changes packets' source and destination addresses only. It is used for *IPAddrRewriter*. The A_1 argument may be a range of IP addresses, similar to P_1 above.
- '*pattern* name'. Named patterns are stored in a special *IPRewriterPatterns* element. Through references like '*pattern* name', these patterns can be shared by multiple rewriter elements, or by multiple input ports on a single rewriter element. This is particularly useful to keep source ports unique among input ports or translation elements: since the single pattern is shared, source ports are kept unique.

– ‘elementname’. A rewrite rule may consist of a single element name, which names a mapping helper element. When fresh packets are encountered, the translator element will call a method on the specified mapping helper. The mapping helper may install new mappings using whatever criteria it likes, thus specifying how the packet should be translated.

Note that rewrite patterns need only specify new flow identifiers. The rest of the configuration determines which old flow identifiers will reach a particular translation element input port, and therefore a particular rewrite pattern.

4.4 *IPAddrRewriter*

We now turn to particular translation elements. The first, *IPAddrRewriter*, implements Basic NAT as specified in RFC 1631 [5]. It can implement static or dynamic variants depending on its configuration string.

Again, in Basic NAT, local machines are assigned IP addresses from a public pool as they access the Internet. *IPAddrRewriter* stores these address correspondences in a mapping table. A flow identifier is either a source address or a destination address. When a packet arrives, its source address and destination address are each used to look up a mapping. If neither corresponds to a mapping, the packet is fresh, and the element uses the corresponding rewrite pattern. *IPAddrRewriter* mappings only change packets' source address or destination address, and their IP checksums.

This example *IPAddrRewriter* element implements Basic NAT. Packets arriving on the first input, which correspond to the first rewrite pattern, are expected to have originated on the private network. Packets arriving on the second input are expected to have originated on the public network. Thus, fresh packets arriving on the first input should introduce new mappings—add new private–public address correspondences—while fresh packets arriving on the second input should be dropped. This is easily achieved:

```
IPAddrRewriter(pattern 18.26.49.1-18.26.49.14 -,
               drop);
```

(In this example, and all succeeding examples, we have left out additional arguments that specify the output ports to be used for a given rewriter pattern.) Again, *IPAddrRewriter*'s configuration string specifies the public address pool—the 14 addresses from 18.26.49.1 to 18.26.49.14—but not the private address pool.

4.5 *IPRewriter* and *TCPRewriter*

The *IPRewriter* and *TCPRewriter* translation elements implement network address port translation, or NAPT [13]. In this form of NAT, several IP addresses may be simultaneously translated to a single IP address, because the TCP/UDP port number space is further used to distinguish between sessions; see Figure 5.

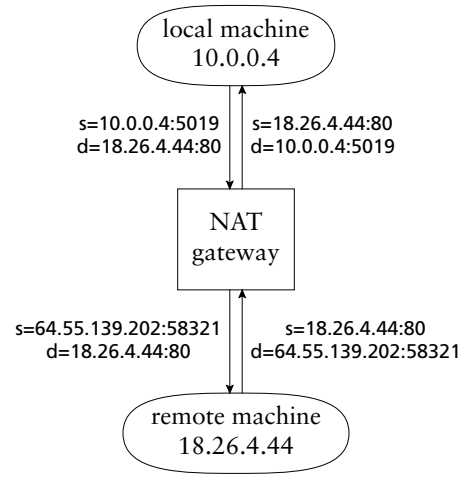

FIGURE 5—Network Address Port Translation.

NAPT is limited to the TCP and UDP protocols. A NAPT flow identifier consists of the tuple of protocol, source address, destination address, source port, and destination port. An *IPRewriter* mapping changes packets' source and destination addresses, source and destination ports, IP checksums, and TCP/UDP checksums. *TCPRewriter* is specialized for TCP only; a *TCPRewriter* mapping can additionally change packets' sequence number and acknowledgement number fields.

This example *IPRewriter* element implements simple NAPT, also known as IP masquerading. This NAT application lets multiple machines with different private IP addresses share a single public IP address for communication with the Internet at large. Packets arriving on the first input, which correspond to the first rewrite pattern, are expected to have originated on the private network. Packets arriving on the second input are expected to have originated on the Internet. Thus, again, fresh packets arriving on the first input should introduce new mappings, while fresh packets arriving on the second input should be dropped.

```
IPRewriter(pattern 18.26.49.7 1024-65535 - -,
           drop);
```

In this example, all outgoing flows are rewritten to use unreserved source ports. However, say we would like outgoing flows with reserved source ports to also have reserved source ports when translated. This is easy to achieve. We just add another input port and corresponding pattern and, elsewhere in the configuration, make sure to send packets to the correct input for their source port.

```
IPRewriter(pattern 18.26.49.7 1-1023 - -,
           pattern 18.26.49.7 1024-65535 - -,
           drop);
```

4.6 *ICMPPingRewriter*

The *ICMPPingRewriter* element demonstrates how the translation element design applies to specific protocols. It provides address translation, similar to *IPRewriter*-style NAPT, for ping packets; since ICMP echoes and echo replies do not have ports, they are not supported by *IPRewriter* itself. *ICMPPingRewriter*'s flow identifiers consist of packets' source address, destination address, and ICMP identifier field. Every input echo request packet is changed to have a single source address. *ICMPPingRewriter* changes the ICMP identifier field to a unique number; this resembles *IPRewriter*'s use of source port ranges. Echo replies are rewritten accordingly.

For example, this simple *ICMPPingRewriter* lets pings travel through the NAPT described in the last subsection.

```
ICMPPingRewriter(18.26.49.7, -);
```

4.7 *RoundRobinIPMapper*

The *RoundRobinIPMapper* element provides one example of a mapping helper. Its configuration argument consists of a list of rewriter patterns. Like any mapping helper, it receives requests from translation elements to create mappings for fresh packets. It delegates each mapping request to one of its component rewriter patterns; it cycles through those patterns in round-robin order as it receives requests.

Mapping helpers need implement exactly one C++ function:

```
Mapping *get_map(IPRw *rewriter, int ip_protocol,
                 const IPFlowID &fresh_flow_id,
                 Packet *fresh_packet);
```

The `rewriter` argument specifies the relevant translation element. The other arguments, including `fresh_packet`, describe the fresh packet that needs to be rewritten. The mapping helper is expected to choose a new mapping, install that mapping into `rewriter`, and return it.

RoundRobinIPMapper can, for example, implement virtual servers through load sharing network address translation, or LS-NAT [12]. In LS-NAT, a single, well-known IP address—often the address of the network address translator—actually refer to a pool of servers. As requests arrive at the well-known server, a translator rewrites them and sends them off to one server from the pool. Responses from the pool servers are re-rewritten to appear as if they originated at the well-known server address.

The following element returns mappings that send input flows to four servers in round-robin order:

```
rr_mapper :: RoundRobinIPMapper
    (pattern - - 18.26.49.2 -,
     pattern - - 18.26.49.3 -,
     pattern - - 18.26.49.4 -,
     pattern - - 18.26.49.5 -);
```

RoundRobinIPMapper is meaningless by itself; for instance, packets do not pass through it. It must be used by a translation element, such as this *IPRewriter* element:

```
IPRewriter(rr_mapper,
           drop);
```

The rest of the configuration might send Web requests to this element's first input port, and replies from one of the 18.26.49.* servers to its second input port.

Clearly, the mapping helper interface is flexible enough to support arbitrarily complex load balancing algorithms, among other things. Mapping helpers are also easy to write: *RoundRobinIPMapper* takes less than 100 lines of C++ code, and most of that is concerned with parsing the configuration arguments.

4.8 *ICMPRewriter*

The *ICMPRewriter* application-level gateway element allows ICMP error packets—redirects, host unreachables, and so forth—to pass through a NAT gateway. These packets contain a portion of the offending packet's header, which will include its flow ID. The *ICMPRewriter* element extracts this flow ID, queries a specified *IPRewriter* element to find the corresponding mapping, and, if that mapping exists, rewrites the ICMP packet and the embedded IP header accordingly.

ICMPRewriter calls a simple function on the relevant translation element to find a mapping, specifically:

```
Mapping *get_mapping(int ip_protocol,
                     const IPFlowID &flow_id);
```

ICMPRewriter must be provided with the names of the translation elements it should check for mappings. For example:

```
ICMPRewriter(rw);
```

Like mapping helpers, application-level gateway elements can be simple to write; *ICMPRewriter* takes about 150 lines of C++ code.

4.9 *FTPPortMapper*

The *FTPPortMapper* element is a slightly more complex application-level gateway element, one that rewrites FTP

control packets to support NAT. The FTP protocol [10] can contain embedded IP addresses in ASCII to support the PORT and PASV commands. A PORT command specifies that an FTP client has opened a port for listening; it expects the FTP server to connect to that port and send it data. This causes problems for NAT or NAPT, both because the client's address may need to be modified, and because the server is not normally allowed to initiate connections into a local network. *FTPPortMapper* takes FTP control packets, which it monitors for PORT commands. When it sees a PORT command, it creates a new mapping on some translation element corresponding to that port—thus allowing the FTP server to access that port—and then rewrites the packet's data to refer to the newly mapped port. Because the IP address and port are in ASCII, this data rewriting may cause the packet to grow or shrink, which will affect TCP sequence numbers. Therefore, *FTPPortMapper* also manipulates sequence number and acknowledgement number offsets on a *TCPRewriter* element, through which the FTP control packets are expected to pass.

Like *ICMPRewriter*, *FTPPortMapper* must be provided with the names of relevant translation elements: one for the FTP control connection, one for the data connection. It also takes a single pattern, which is used for creating mappings for PORT commands. For example:

```
FTPPortMapper(rw, tcp_rw,
     pattern 18.26.49.7 1024-65535 - -);
```

FTPPortMapper takes about 250 lines of C++ code.

4.10 Discussion

Once a given rewrite pattern's source port range is exhausted, that pattern will drop new packets rather than reuse active source ports. As mappings are removed, of course, the corresponding source ports become available again. A similar statement holds for source IP address ranges used with *IPAddrRewriter*.

Some care is required to ensure that different fresh packets are never mapped to the same new flow ID. For example, consider these patterns:

```
pattern 1.0.0.1 1024-4096 - -,
pattern 1.0.0.1 2048-8192 - -
```

If packets with flow IDs $A{:}B \rightarrow$ 18.26.4.44:80 and $C{:}D \rightarrow$ 18.26.4.44:80 arrive on the two input ports, the rewriter might choose 1.0.0.1:2048 $\rightarrow$ 18.26.4.44:80 as the new flow ID for both. This would make the reply mapping ambiguous. However, the rest of the configuration might ensure that packets arriving on input 0 had destination port 80, while packets arriving on input 1 had destination port 22. In this case, there would be no conflict between the patterns, because two new flow IDs created by the two patterns would never share the same destination port. As described above, shared named patterns also prevent this problem.

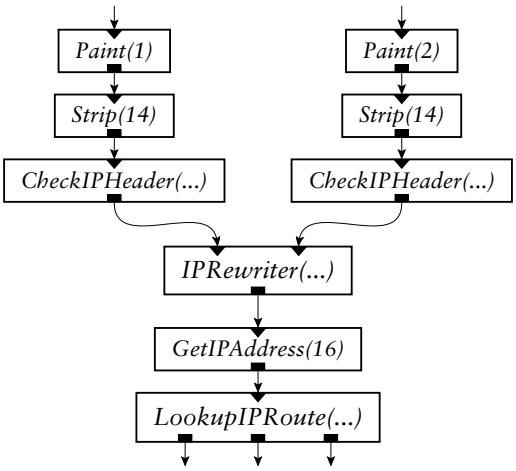

FIGURE 6—An extension to Figure 3 supporting NAPT.

Click supports *hot swapping*, where a newly installed configuration can atomically take the state of an old configuration to avoid routing hiccups. The *IPRewriter* elements support hot swapping, so their mapping tables need not be lost when configurations change.

5 Examples

The best way to demonstrate the flexibility and utility of the IP rewriting elements is through examples. The previous section described example configurations in isolation; this section shows how to use the elements in real router configurations. We describe an IP router configuration extended to use NAPT, a transparent traffic diverter, and a complex NAT configuration that has been in real use for six months.

5.1 Port routing

This section illustrates a simple NAPT, or masquerading, router; that is, an IP router configuration extended to support network address/port translation. Again, this allows a set of machines with private IP addresses to share a single public IP address through TCP/UDP port rewriting. This configuration is built on the Click IP router of Figure 3.

One natural place to insert NAPT into an IP router is just before the routing table. Thus, packets will be routed based on the rewritten flow IDs. Figure 6 demonstrates this extension. The path in Figure 3 from *Strip* to *CheckIPHeader* has been split in two: packets arriving from the two interfaces are kept distinct until *IPRewriter*, so

that *IPRewriter* can distinguish between incoming and outgoing packets. The *IPRewriter* element has two input ports, and therefore two rewrite patterns. Assuming that the externally visible address is 24.1.2.3, its particular patterns might be as follows: (Assume that the left path—and the first input port—is for packets from the Internet, while the right path—and the second input port—corresponds to packets from the private network.)

```
IPRewriter(drop,
           pattern 24.1.2.3 1024-65535 - - 0 0);
```

Notice the extra `0 0` arguments to the `pattern` specification. These numbers indicate the output ports to be used for new flows and their reply flows, respectively; they determine how packets should be demultiplexed between *IPRewriter*'s output ports. In this example, however, the designer has decided that *LookupIPRoute* will do any necessary demultiplexing, so *IPRewriter* has only one output port.

This configuration drops all fresh packets from the outside world. This is common, but often too restrictive. A site with many internal hosts but only one externally visible IP address may want to run servers that accept TCP connections from the outside world. One option is to run all of these servers on the same machine that runs the site's NAT, since that is the machine that owns the one IP address. This may not be acceptable, however, and the site may need a way to route connections to the IP address to internal hosts. If there is more than one kind of service, there may be more than one internal host involved. What's needed is a way to route connections to port 21 to the internal FTP server, connections to port 25 to the internal mail server, and so forth.

We can use extra *IPRewriter* input ports to implement these additional rewriting rules. *IPRewriter* itself will not classify packets by destination port, so a Click *IPClassifier* element is needed to separate packets from the outside world destined for different services. These separate streams of packets should be sent to different inputs of the same *IPRewriter*, rather than to separate *IPRewriter*s; this will let that single *IPRewriter* take care of all reply flows from the internal servers.

As a concrete example, suppose that the externally visible address is 24.1.2.3, and that there are two internal servers: host 10.0.0.10 takes care of port 25, and host 10.0.0.11 takes care of port 80. In addition, ordinary NAT is to be applied to outgoing connections. Figure 7 shows the arrangement of the *IPClassifier* and *IPRewriter* elements for this example. Packets from the outside world arrive on the left-hand input; internal packets arrive on the right-hand input. The *IPClassifier*'s three outputs correspond to packets for port 25, packets for port 80, and other packets, respectively. The relevant element configurations are as follows:

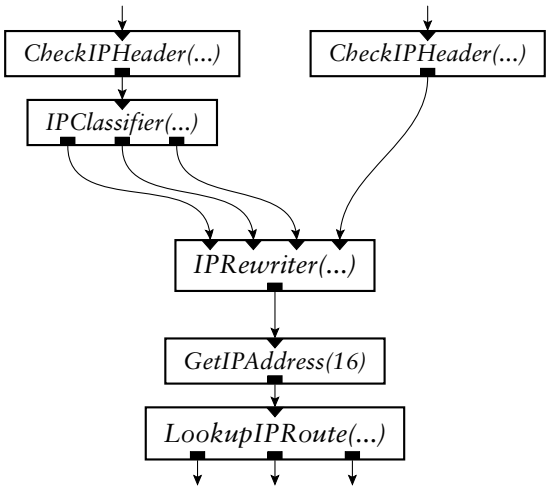

FIGURE 7—A router configuration for port routing.

```
IPClassifier(tcp port 25, tcp port 80, -);
IPRewriter(pattern - - 10.0.0.10 - 0 0,
           pattern - - 10.0.0.11 - 0 0,
           drop,
           pattern 24.1.2.3 1024-9999 - - 0 0);
```

Assume, on the other hand, that there were three Web servers, 10.0.0.11, 10.0.0.12, and 10.0.0.13, among which Web requests should be load-balanced. This requires a four-line addition to the configuration:

```
IPRewriter(pattern - - 10.0.0.10 - 0 0,
           weblb,
           drop,
           pattern 24.1.2.3 1024-9999 - - 0 0);
weblb :: RoundRobinIPMapper
          (pattern - - 10.0.0.11 - 0 0,
           pattern - - 10.0.0.12 - 0 0,
           pattern - - 10.0.0.13 - 0 0);
```

Again, in these configurations, *IPRewriter* and friends just take care of NAT functionality. Classification functionality, for example, is handled by other elements in the configuration. This makes NAT-related classification functionality more flexible, and makes the *IPRewriter* element itself easier to understand.

5.2 Transparent traffic diverter

This section describes how to build a transparent traffic diverter suitable, for example, for implementing transparent proxies, and shows how such a diverter fits into a larger system. The diverter is worth examining in detail because it makes it easy to turn ordinary proxies and servers into transparent proxies.

The diverter is meant to intercept all connections of a certain type, regardless of intended destination, and

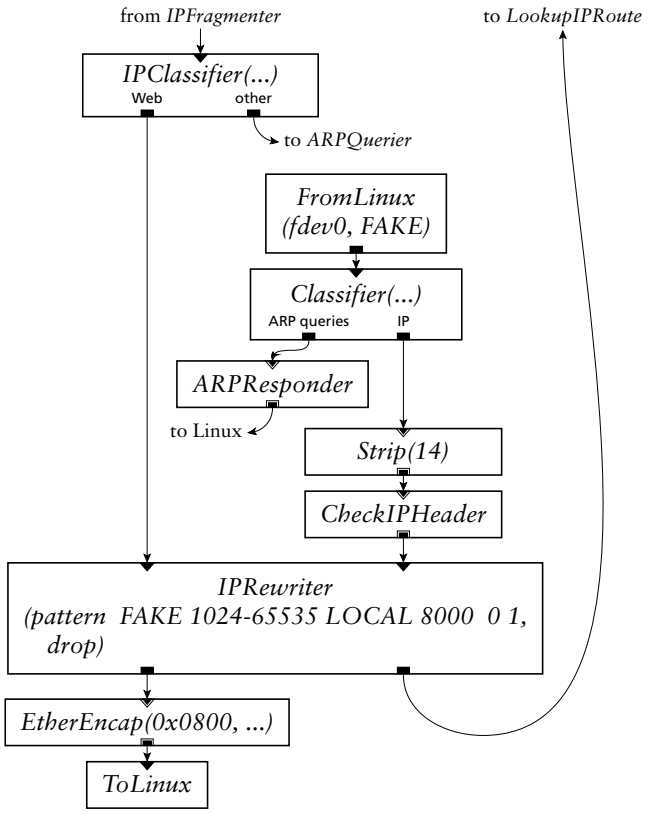

FIGURE 8—A transparent Web traffic diverter extension for the IP router configuration.

send them to a particular host and port. The connections arrive at that host looking as if they were originally intended to connect there. The program listening to the relevant port can accept the connections as if they were ordinary connections. When the program sends data on such connections, the diverter rewrites them to look as if they came from the host the connection was originally meant to connect to.

One use of the diverter is to help build transparent Web proxies [3]. In this case the program would be an ordinary non-transparent Web proxy, slightly modified to deduce the real server from the HTTP `Host:` header.

Figure 8 shows a Click configuration fragment that fits the diverter's *IPRewriter* into the larger IP router configuration of Figure 3. Figure 8 catches outgoing traffic just before it reaches the outgoing interface's *ARPQuerier* and separates Web traffic from other traffic using an *IPClassifier*. Web traffic passes through an *IPRewriter* element, which rewrites packets' flow IDs as described in Section 4. The rewrite rule for outgoing packets is "*pattern FAKE 1024–65535 LOCAL 8000 0 1*", so packets are rewritten and sent to port 8000 on the local machine. The proxy should be listening on that port. The packets' source addresses are also changed, to *FAKE*. (*FAKE* and *LOCAL* stand for IP addresses.)

The *FromLinux* element captures replies from the proxy. *FromLinux* installs a fake device into Linux's device table; here, that device is named *fdev0* and its native IP address is *FAKE*. Linux's routing table is manipulated so that packets destined for the address *FAKE* are sent to the fake device. Since packets directed to the proxy have their source addresses changed to *FAKE*, replies from the proxy will have destination address *FAKE*, and will be sent to the fake device. As packets arrive on that device, they are emitted into the Click router configuration by *FromLinux*. The configuration must respond to ARP requests sent to the fake device, which explains the *Classifier* and *ARPResponder* elements. IP packets from the fake device—namely, the proxy's replies—are sent through the *IPRewriter*, where they are rewritten to look like replies from the intended server and properly forwarded via the configuration's routing table.

This diversion technique works for more than just HTTP traffic. For example, we used exactly this configuration fragment to build a transparent DNS cache that diverts DNS UDP packets to a host running an name server. The name server need not be modified at all as long as recursion is enabled.

5.3 Combining firewalling and NAT

This section describes how different *IPRewriter*-like elements can be composed to create a firewall and network address translator with several interesting features:

- All but a minimal set of services and internal machines are inaccessible from outside.
- Internal hosts have unlimited access to the outside.
- The firewall presents several IP addresses to the outside world. Selected traffic to these addresses is translated and forwarded to internal servers.
- Internal hosts reachable from the outside world can be accessed from internal machines via either their external or internal IP addresses.
- ICMP traffic (for example, "port unreachable" messages) related to traffic going through the firewall is also translated and allowed through the firewall.

The Click configuration that implements these features is compact and easy to understand. Most importantly, since it is constructed from a small set of flexible and general-purpose building blocks, it can be extended easily to support new features.

The rewriter patterns and the *IPRewriter* below form the core of the configuration.

```
IPRewriterPatterns(
  pat_to_out FILESERVER_OUTSIDE 20000-65535 - - ,
  pat_to_file - - FILESERVER_INSIDE - ,
  pat_to_mail - - MAILSERVER_INSIDE - ,
  pat_file_lb FIREWALL_INSIDE 20000-65535
              FILESERVER_INSIDE - ,
  pat_mail_lb FIREWALL_INSIDE 20000-65535
              MAILSERVER_INSIDE - );

rw :: IPRewriter(
  pattern pat_to_out  0 1,
  pattern pat_to_file 1 0,
  pattern pat_to_mail 1 0,
  nochange 2,
  pattern pat_file_lb 1 1,
  pattern pat_mail_lb 1 1 );
```

(Again, words in CAPITALS represent IP addresses.)

Let's look at these patterns one by one. All internal TCP or UDP packets bound for outside are sent to input 0 of rw and pushed out on output 0.

Appropriate packets (for example, SSH, SMTP) from outside sent to the external file server or mail server addresses appear on input 1 or 2, respectively. Their destination addresses are changed to the internal IP address of the appropriate server and they are then sent to the internal network via rw output 1.

All other TCP and UDP packets from outside are sent to input 3 of rw. If a mapping exists—because it was created by some of the outgoing traffic on input 0—they are rewritten, otherwise they are fed through unchanged. In both cases, they are pushed to rw's output 2. Rewritten packets (now with internal destination addresses) are forwarded to the inside network, whereas others are sent to the firewall's Linux IP stack.

Inside traffic may sometimes be directed at the outside addresses of the file or mail server; for example, a laptop user may join the internal network but use an external DNS server to find the address of the mail server. Rather than going out through the firewall and being routed back in by the external router, this traffic is rewritten by rules 4 and 5 of rw to point back to the internal addresses, and never leaves the internal network.

Outgoing FTP control traffic is passed to an *FTPPortMapper*: this element creates a new mapping according to the pat_to_out pattern defined above, rewrites PORT commands in the payload to reflect the mapping, and installs the mapping in two other rewriters, tcp_rw and rw. tcp_rw then takes care of rewriting the headers for inbound and outbound control traffic, whereas rw handles FTP data traffic like any other TCP traffic for which it has a mapping.

```
tcp_rw :: TCPRewriter(pattern pat_to_out 0 1,
  drop);

ftp_pm :: FTPPortMapper(tcp_rw, rw,
  pat_to_out 0 1);
```

ICMP pings are also handled specially. Echo requests from the inside world are sent to ping_rw, which rewrites them to have source FILESERVER_OUTSIDE and sends them outside. Incoming echo replies are also sent to ping_rw, but are rewritten only if they match outgoing requests. Incoming echo requests, on the other hand, pass through ping_rw unchanged and are then passed to the firewall's Linux stack.

```
ping_rw :: ICMPPingRewriter(FILESERVER_OUTSIDE, -);
```

Finally, any other outside ICMP traffic is sent to an *ICMPRewriter*. This element rewrites an ICMP packet (both header and payload) and forwards it on only if an appropriate mapping exists in one of the given rewriters (rw or tcp_rw)—that is, only if the ICMP packet is related to other traffic passing through the firewall. Otherwise, the packet is discarded.

```
icmp_rw :: ICMPRewriter(rw tcp_rw);
```

6 Performance

IP rewriting in Click is sufficiently fast to make its performance impact negligible in the context of a larger router, firewall, or other packet processing configuration.

We evaluated the latency of an individual *IPRewriter* by using micro-benchmarks. Measurements were taken on a 700MHz Pentium III with 256MB of memory and a 256KB L2 cache. The test harness consists of a packet source element feeding fake UDP packets through an *IPRewriter* and into a packet sink.

We ran two tests. In the first test, the packet source generates a uniform stream of 100 identical packets. These identical packets create only one mapping in the *IPRewriter*, so the test measures the forwarding cost of the *IPRewriter*, excluding the overhead of generating new mappings. The median forwarding latency of *IPRewriter* in this scenario is 393 cycles, or 561 ns, per packet, as measured with Pentium performance counters.

In the second test, the packet source generates a stream of 100 unique packets by successively incrementing the UDP source and destination port numbers. This measurement combines the overhead of packet forwarding with the overhead of generating new mappings. Generating a new mapping involves adding two entries—one for the forward mapping and one for the corresponding reverse mapping—to a hash table. Each entry consists of an integer that represents a unique flow identifier and a structure that contains information about the mapping. The median latency for forwarding and generating a new mapping *IPRewriter* is 2338 cycles,

or 3.34 μs, per packet. This is of course a worst-case value, since for normal traffic not every packet will result in a new mapping being created. Furthermore, we expect to reduce this overhead considerably by improving the implementation—the existing hash table is generic and unoptimized. Lastly, this overhead is comparable to that of common operations that must happen for *every* packet: for example, on our hardware, device interaction cost almost 1000 cycles per packet.

7 Related Work

The first documented IP network address translator [5] performed address-based NAT only. Network Address Port Translation, or NAPT, and the use of NAT for load balancing appeared later [12, 13].

RFC 2663 lays out consistent terminology for NAT variants [13]. Using its terminology, Click NAT elements can perform Basic NAT with *IPAddrRewriter*, Network Address Port Translation with *IPRewriter*, Load Sharing NAT with *IPRewriter* plus an *IPMapper* element, and limited Two-Way NAT with *IPRewriter*. Full Click configurations can implement full Two-Way NAT (with help from a DNS proxy), Twice-NAT (two *IPRewriter*s in different realms), and Multihomed NAT.

Hasenstein describes a wide variety of network address translation configurations in the context of a system for NAT in Linux [7]. All of his configurations may be easily implemented in Click.

Cisco IOS features a flexible NAT implementation [2] supporting static and dynamic address-based NAT, NAPT, round-robin load sharing NAT, and combinations thereof. Interfaces are divided into two classes, "inside" and "outside". The translations applied to a particular packet depend on the class of interface on which it was received, and, optionally, on its source address, destination address, protocol, or port number. Other arrangements, such as more than two classes of interface or other load sharing arragements, appear difficult or impossible to achieve.

Similarly, while the NAT implementations shipped with desktop operating systems—Linux's ipchains and ipnat [7], BSD's IP Filter [11], and Windows 2000's NAT—are flexible to different degrees, none of them appear to support multiple NAT components in a single configuration, or allow fully flexible control over NAT placement relative to other forwarding tasks.

Cohen et al. [4, 3] present a configurable tool for re-mapping packet addresses and port numbers. It consists of a kernel module that implements re-mapping with a fixed table, and applications that add new mappings when they observe packets from as-yet unmapped flows. While the system can perform a wide range of NAT functions, it is embedded inside a fixed router configuration; unlike Click's NAT tools, the way it interacts with other forwarding functions cannot be changed.

8 Conclusion

We have presented a flexible, usable set of components for network address translation in a modular networking system, Click. These components implement only the core functionality required of any network address translator—namely, changing IP packet headers and finding mappings corresponding to input packets' flow identifiers. They leave other functions, such as determining which packets should be subject to translation, to other parts of a router configuration. This makes configurations involving address translation more flexible and understandable. NAT elements can be placed in a configuration exactly where they are required; packets meant for translation can be selected in arbitrary ways; the mechanism for choosing a translation for a new packet is completely extensible; and multiple NAT elements can coexist in a single configuration. The IP rewriting components are made more useful and general by the modular networking system of which they are a part. We demonstrated the practical usefulness of this system with real configurations, including an IP router with port translation (IP masquerading), a generic transparent traffic diverter, and a real, and complex, combined firewall and NAT configuration, and showed that the IP rewriting elements have acceptable performance cost.

The firewalling NAT described in Section 5.3 has been in real use routing traffic for a small startup for six months. It, and the components described in this paper, are freely available on line at `http://www.pdos.lcs.mit.edu/click/`.

Acknowledgements

We thank Paul Hsiao, Sulu Mamdani, and Mazu Networks, Inc. for support of this project, Benjie Chen for his initial work on the configuration described in Section 5.3, and Frans Kaashoek and the members of MIT LCS's Parallel and Distributed Operating Systems group for supporting Click.

References

[1] F. Baker. Requirements for IP Version 4 routers. RFC 1812, Internet Engineering Task Force, June 1995. `ftp://ftp.ietf.org/rfc/rfc1812.txt`.

[2] Cisco Systems. Cisco IOS Network Address Translation (NAT). Technical report, September 1998. `http://www.cisco.com/warp/public/701/60.html`, as of December 2000.

[3] A. Cohen, S. Rangarajan, and N. Singh. Supporting transparent caching with standard proxy caches. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[4] Ariel Cohen and Sampath Rangarajan. A programming interface for supporting IP traffic processing. In *Proc. of IWAN '99: Active Networks, First International Working Conference*, number 1653 in Lecture Notes in Computer Science, pages 132–143, June 1999.

[5] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, Internet Engineering Task Force, May 1994. `ftp://ftp.ietf.org/rfc/rfc1631.txt`.

[6] T. Hain. Architectural implications of NAT. RFC 2993, Internet Engineering Task Force, November 2000. `ftp://ftp.ietf.org/rfc/rfc2993.txt`.

[7] Michael Hasenstein. IP network address translation. Diplomarbeit, Technische Universität Chemnitz, Chemnitz, Germany, 1997. Available on line at `http://www.suse.de/~mha/linux-ip-nat/diplom/nat.html` as of December 1, 2000.

[8] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4), November 2000.

[9] E. Nordmark. Stateless IP/ICMP Translation algorithm (SIIT). RFC 2765, Internet Engineering Task Force, February 2000. `ftp://ftp.ietf.org/rfc/rfc2765.txt`.

[10] J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC 959, Internet Engineering Task Force, October 1985. `ftp://ftp.ietf.org/rfc/rfc0959.txt`.

[11] Darren Reed. IP Filter TCP/IP packet filtering package. Technical report, 2000. `http://coombs.anu.edu.au/~avalon/`, as of December 2000.

[12] P. Srisuresh and D. Gan. Load sharing using IP Network Address Translation (LSNAT). RFC 2391, Internet Engineering Task Force, August 1998. `ftp://ftp.ietf.org/rfc/rfc2391.txt`.

[13] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) terminology and considerations. RFC 2663, Internet Engineering Task Force, August 1999. `ftp://ftp.ietf.org/rfc/rfc2663.txt`.

[14] P. Srisuresh, G. Tsirtsis, P. Akkiraju, and A. Heffernan. DNS extensions to Network Address Translators (DNS_ALG). RFC 2694, Internet Engineering Task Force, September 1999. `ftp://ftp.ietf.org/rfc/rfc2694.txt`.

[15] G. Tsirtsis and P. Srisuresh. Network Address Translation—Protocol Translation (NAT-PT). RFC 2766, Internet Engineering Task Force, February 2000. `ftp://ftp.ietf.org/rfc/rfc2766.txt`.