Network processors applied to IPv4/IPv6 transition

Eric Grosse Lakshman Y. N.
Bell Labs, Lucent Technologies
600 Mountain Ave
Murray Hill NJ 07974
{ehg,ynl}@research.bell-labs.com

ABSTRACT

IPv6 is beginning to be adopted by organizations and countries that expect to run critically short of IPv4 addresses. Small scale trials can rely on dual-stack transition mechanisms, in which both an IPv4 and an IPv6 address are assigned to new hosts, which can therefore talk directly to old and new networks. But full deployment will need to use network address/port/protocol translation (NAPT-PT), in which new hosts are given only IPv6 addresses and must talk through a gateway in order to speak to old networks. The natural location for these NAPT-PT gateways will gradually shift from very local subnets to the edge of a provider network as IPv6 becomes more widely deployed, increasing the demands on the capacity and availability.

Programmable network processors have the flexibility that custom silicon tends to lack and hence are especially well-suited for this crucial gateway between IPv6 clouds and the IPv4 world. To build a scalable middlebox requires redundancy support for stateful failover and load balancing, again putting a premium on programmability.

1. INTRODUCTION

The next generation IP protocol, IPv6, became an IETF standard in late 1995, motivated by concern that the IPv4 address space would soon be exhausted. The use of CIDR, less generous allocation of address blocks and the use of network address translation at the edges has postponed the day of reckoning and so far, there has been no large scale deployment of IPv6. However, large populations in countries such as Japan, China and India are gradually coming online and depletion of the IPv4 address space is inevitable. It is generally agreed that IPv4 and IPv6 will coexist for a long time and that three main classes of IPv4/IPv6 transition mechanisms, dual stacking, tunneling and address translation NAPT-PT will be used [11].

In full deployment (as opposed to small trials) the dual-stack approach will not be feasible due to extreme pressure on IPv4 addresses. Network address/port/protocol translation (NAPT-PT) will be required when hosts with only IPv6 address need to talk with legacy IPv4-only systems in the rest of the world. Consequently, the NAPT-PT functionality will move inwards from the very edge of the network (as in home and small business boxes now) to the edge of a provider network. As a single middlebox providing NAPT-PT becomes a critical element in the paths of a large number of users, it will have to be able to handle large volumes of traffic, perform complex packet processing and be fail-safe. Designing the hardware and software to keep up with these increased demands on such middle boxes at low cost is a challenge.

ASICs provide the highest performance for hardware speed-ups, but programmable network processors are becoming available that offer enough performance for middleboxes while maintaining more flexibility for adapting to unpredictable markets such as IPv6 deployment. Network processors employ pipelined parallelism for achieving high packet processing throughput in addition to a host of other techniques such as instruction sets and hardware support for data structures specially geared towards packet processing. Most of the currently available network processors are of "first-generation" – somewhat hard to program and constrained in what one can do with them. Many of the strengths and weaknesses of network processors and strategies for deriving high performance out of them have been detailed in [8, 9] and some of the difficulties of programming them have been addressed in [1].

Middleboxes that need to perform deep packet inspection and rewriting stress current network processor architectures significantly more than packet lookup and forwarding do because the session state puts a load on the memory bus, which is not much larger on the network processor than it is on an ordinary microprocessor. To explore these issues, we have built an experimental gigabit/second stateful IPv6-IPv4 translator capable of handling hundreds of thousands of concurrent conversations between a large IPv6 island and the global IPv4 ocean on a network processor board. The device has automatic failover and load sharing capability that preserves ongoing conversations.

The following discussion is focused on NAPT-PT, motivated by the IPv6 transition issue described above. However, this is a specific instance of a more general problem and many of our observations apply to firewalls just as well; some even

apply to DHCP/DNS servers. The rest of the paper is organized as follows. We first describe the context for using a middlebox performing NAPT-PT and its place in the network. We follow up with some details of our middlebox implementation on a network processor board. We then describe a simple state-sharing protocol to support active failover. We conclude with a discussion of the difficulties encountered, remedies and future directions.

2. WHAT KIND OF ADDRESS TRANSLATOR AND WHY?

We are most interested in the case of a service provider or organization that needs to provide internet connectivity to a large number of hosts but has only a much smaller number of global IPv4 addresses at its disposal and hence, has decided to build an all IPv6 network. It is very likely that at its inception, this IPv6 network is going to be an island in the IPv4 ocean. Hosts in our provider's island will need ways to talk to hosts in the IPv4 ocean as well as in other IPv6 islands.

Of the transition mechanisms mentioned earlier, dual stacking with global IPv4 addresses for each host would be the preferred solution, but is not a possibility here since the provider does not have enough global IPv4 addresses. The provider may choose to use up some fraction of the IPv4 addresses at its disposal on dual-stack machines for hosting services reachable by IPv4 as well as IPv6 clients. Our provider will also have to provide some kind of tunnel endpoints (6-in-4 automatic/configured/2002-prefix) for hosts in the provider's network that want to talk to hosts in other IPv6 islands across the IPv4 ocean.

For hosts that want to access services available only in the IPv4 ocean, short of proxying every such service, the only choice left to the provider is to perform network address translation. Several flavors of network address translation are in use[11, 10]. Static mapping maps each IPv6 address in the provider's network to a global IPv4 address. Clearly this is not feasible since our provider does not have enough IPv4 addresses. Dynamic address mapping maps an IPv6 address in the provider's network to a global IPv4 address for some length of time; the mapping can change over time, allowing different hosts in the provider's network to masquerade as a particular IPv4 host at different times; however, at any one time, the number of hosts in the provider's network that can talk to hosts in the IPv4 ocean is limited to the number of global IPv4 addresses available to the provider for this purpose. This may be a very severe restriction. The next possibility is network address-port translation – this allows multiple hosts in the IPv6 network to share the same IPv4 address concurrently. It can support up to about the number of global IPv4 addresses times 64K simultaneous TCP and UDP sessions between hosts in the providers' network and the IPv4 ocean and provides the most flexibility.

Our implementation provides this last option. The device, which we call NAT, sits between the provider's IPv6 network and the IPv4 ocean and performs NAPT-PT (network address-port and protocol translation). The device is stateful (state being the mappings between the IPv6 and IPv4 worlds for each live session) and would be a single point of failure. We have implemented a simple state-sharing protocol that allows us to use multiple NATs for redundancy. Failure of one/more NATs can possibly result in reduced performance under heavy load but not in session loss. As an additional benefit of the state-sharing protocol, we get some coarse grain load balancing among multiple NAT devices, resulting in reduced latency/packet loss under heavy loads.

Our design requirements were influenced by observing all packets on a lightly loaded fractional T3 link from Bell Labs Research to the outside world for several 12 hour periods. In a typical (6am to 6pm) case we observed a total of 29 million packets, of which 94% were TCP and 5% were UDP, with a total of 670,000 sessions, for an average of 43 packets per session. Since the distribution of session lifetimes is long-tailed, one can not estimate from the raw packet counts a crucial parameter: how large the session table needs to be. So we ran a NAT simulation on the traces, using a TCP timeout of 1 minute after the second FIN and a UDP timeout of 5 minutes. The peak number of sessions was not very sensitive to these parameters. We arbitrarily cut off TCP sessions after 6 hours of inactivity; we observed such connections on ports for telnet, ssh, http, and lpr, but rarely: 98.8% of TCP sessions ended within our sampling window with normal FIN exchange. At the peak, we measured 7211 active sessions. Our main design conclusions are that:

- for a gigabit NAT, one should allow for hundreds of thousands of peak active sessions. We assume that sessions don't change size when lots of users are aggregated to get a gigabit Internet connection, so the peak connection count should approximately scale with the bandwidth. Our count of 7200 sessions at fractional T3 scales to 320,000 sessions for GbE, even higher since our link was lightly loaded.
- on a network processor, session setup needs to be done in microcode. Session creation is a costly operation. Doing this in a slowpath, such as in the StrongARM on an IXP1200 would create a significant bottleneck when 2.5% ($\approx 1/43$) of the packets are creating a new session table entry. Other packet samplers in the past have estimated average session sizes at 20 or even 10 packets.

NAPT-PT is a well-established, if not yet widely deployed, mechanism; see for example [3], whose code is now distributed by Microsoft. Independent implementations have been constructed by the Electronics and Telecommunications Research Institute in Korea on Linux, by British Telecom on FreeBSD, and no doubt by many others. As the NAPT-PT functionality moves into the edge of a provider network, the ability to handle large volumes of traffic and failover capability become crucial. To the best of our knowledge, these issues have not been addressed before.

3. NAT DESIGN OVERVIEW

NAT sits on the boundary between the provider network and the IPv4 ocean. It appears as a router to the provider's network and as a set of hosts or a stub network to the uplink provider's edge router in the IPv4 ocean. The hosts and routers in the provider's IPv6 island need to have a route

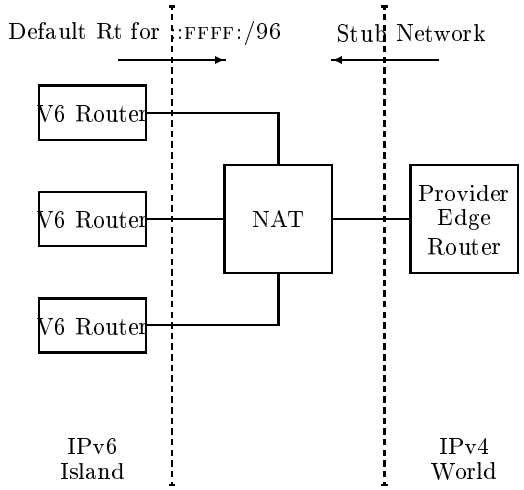


Figure 1: NAT's location in the network

table entry for the IPv6 destination prefix ::FFFF: /96 (IPv6 mapped IPv4 addresses) pointing to the NAT device (Figure 1). Admittedly, this set-up is not the most general one possible but it provides a useful simplification – the IPv6 island and the IPv4 ocean do not have to exchange routing information (if they did, the routing information would need to undergo IPv6 to IPv4 and IPv4 to IPv6 transformation somewhere).

Packets passing through a NAT device are classified into sessions on the IPv6 side where a session is uniquely identified by the 5-tuple consisting of a source address $s6$ (global unicast IPv6), a source port sp, protocol $proto$ (TCP, UDP, ICMP, ...), a destination address $d6$ (IPv6 compatible IPv4 address) and a destination port dp. For each flow, NAT creates a mapping

$$[s6, sp, proto, d6, dp] \Longrightarrow [s4, nsp, proto, d4, dp]$$

where $s4$ is one of the global IPv4 addresses available to the IPv6 island for this purpose, nsp is an unused port on $s4$ and $d4$ is the IPv4 destination address hidden inside $d6$. When a NAT device sees an outgoing packet belonging to a session that it does not know about, it creates a mapping for a new session and adds an entry for the new session in a session table. Care is taken to ensure that a particular IPv6 source address $s6$ is mapped to the same IPv4 source address $s4$ across multiple sessions as this may be important to some layer 7 protocols. Outgoing packets are looked up in the session table using the 5-tuple to identify the session. (For UDP, some applications send a packet to one server and accept a reply from a different server, so we ignore the outside address and port when matching UDP flows.) Incoming packets are looked up using a reverse index into the same session table using the destination host and port fields of the packet as the key. Once a NAT knows the mapping for the session that a packet belongs to, it rewrites the packet's header (IPv6 to IPv4 for outgoing packets, IPv4 to IPv6 for incoming packets) in the obvious way and puts out the packet on the appropriate interface. ICMP packets might be bearing another IP packet as payload, so the header rewriting must be performed on that payload as well. Details of the translation can be found in [13]. The amount of storage required per session (the 7-tuple $[s6, sp, proto, d6, dp, s4, nsp]$ needs 45 bytes) can become a concern as the number of active sessions grows into hundreds of thousands. A few obvious optimizations such as storing $s6$ as 2 bytes of a prefix identifier and 8 bytes of host identifier, using 4 bytes for $d6$, a 2 byte identifier for $s4$ bring down the per session storage requirement to 24 bytes.

4. NAT ON A NETWORK PROCESSOR

We have implemented our NAT device on a Bell labs designed network processor board [7]. The board has two Intel GCIXP1200-FCs at 232MHz, a high density FPGA and and two TCAMs. The FPGA handles communications among daughter cards (GbE, multiple 100Mb ethernet cards), the TCAMs and the IXPs. Each Intel IXP processor has a StrongARM core and six microengines each with four hardware contexts. Each IXP has available off-chip 256MB of SDRAM and 8MB of SRAM and a small amount(4KB) of on-chip scratch. An SDRAM access costs about 33-40 cycles, an SRAM access 16-20 cycles and scratch access about 12-14 cycles when there is no contention. If multiple engines attempt accesses to the same memory simultaneously, the latency can be much higher. Clearly, memory accesses are expensive. Each microengine has 64 general purpose registers and 32 each of SRAM and SDRAM transfer registers. Transfer registers are used to read from/write to memory. Each microengine has a control store able to store 2K instructions.

We use both the IXP1200 chips on the board to perform packet processing. Each IXP1200 maintains a session table (of all the current sessions between the IPv6 island and the IPv4 world) in its SDRAM. Either one of the processors can create/delete a session entry. Changes to session table made by one processor are communicated to the other processor and the two maintain consistent tables. The TCAM is capable of storing up to 128K 144-bit wide entries. We use it as a cache for session table entries.

Packets arriving on a port from the IPv6 island are picked up by the FPGA which then performs a session lookup in the TCAM for the packet. A complete session table entry is a 7-tuple (v6 source address, v6 source port, v4 compatible v6 destination address, v6 destination port, re-written v4 source address, re-written v4 source port), amounting to 336 bits (this can be compressed a bit), way more than 144. Therefore, a TCAM entry uses the v6 source address and the v6 source port as the key. A TCAM lookup match returns an 18-bit quantity which is used as an offset into the full session table by either IXP1200. The FPGA adds an 8-byte header containing information about port numbers, packet size and the TCAM query result and passes the packet to one of the IXP1200s in strictly alternating order (so that the load is evenly split between the two IXP1200's). In case of a successful TCAM lookup, the receiving IXP1200 uses the 18-bit TCAM lookup result as an offset into its session table to make a full 5-tuple comparison with the IPv6 header to ensure a session match before proceeding with header translation. A failed match triggers a full hash table lookup in the IXP1200, which can be expensive if there are hash collisions.

When an IXP1200 creates a new session entry based on a

received packet or finds a session entry for a failed TCAM lookup, it sends a control packet to the FPGA to add that session entry to the TCAM. In the same control message, the IXP1200 also provides an address/index in the TCAM where the new entry needs to be inserted. Entries are replaced in the TCAM using a "least recently inserted" policy, which approximates the usual "least recently used (LRU)" policy without requiring the FPGA or the IXP1200's to maintain state information related to LRU. When an IXP1200 deletes a session entry, it sends another control packet to the FPGA to delete that session entry from the TCAM. The FPGA looks up the entry in the TCAM (thereby obtaining its TCAM index) and then marks that entry as deleted. Control packets get bounced to the remaining IXP1200 by the FPGA which then updates its session table as necessary.

Packets arriving on a port from the IPv4 world are passed on to one of the IXP1200's in alternating order without a TCAM lookup. The session look-up in this direction is through a reverse index on the re-written address/port (IPv4 destination address/port for the packet) which is fast.

Packets transmitted by the IXP1200s onto the IX bus reach appropriate output ports through the FPGA. Each IXP1200 uses one microengine exclusively for receiving and one for transmitting packets from/to external ports via the IX bus. The other four microengines are used for actual packet processing. These engines wait for a signal from the receiver engine for packet arrival. Upon a packet arrival, one of these engines picks up the packet – it inspects the packet up to and including the transport layer header (62 bytes in case of IPv4 packets and 82 bytes in case of IPv6 packets; this includes 14 bytes of ether header and the 8 bytes added by the FPGA). IP header options, fragmented packets, ICMP, and ARP are passed to the StrongARM for processing. We have performed stress tests on the NAT device by connecting its network interfaces to a traffic generator (a Shomiti Explorer) capable of generating and capturing small packets at wire speeds on GbE links. The test consisted of getting the Shomiti to send a small number of fixed types of packets to one of ports of the NAT device and capturing the the packets on the other port and measuring sustainable throughput. The number of session entries was quite small in comparison to the size of the TCAM cache, so every packet resulted in a successful TCAM lookup. Therefore, we have not measured the efficacy of our cache on realistic traffic. We measured the following throughput numbers:

Packet Size (Type)	Direction	Packets per Second	Mb/sec
82 (V6)	6 to 4	1,041,400	683.15
62 (V4)	4 to 6	1,115,000	553.04
82 (V6)	(two way)	809,400	530.70
62 (V4)		1,089,000	540.14

The packet sizes include the FPGA header (8B), an ether header (14B), IP header (20B for IPv4, 40B for IPv6) and a TCP header (20B). In comparison, the packet processing code performing the same function as one of the IXP1200s on a general purpose board with an 866MHz Pentium III running the Plan 9 operating system took about 20μS per packet (not measuring the time spent in the IP stack).

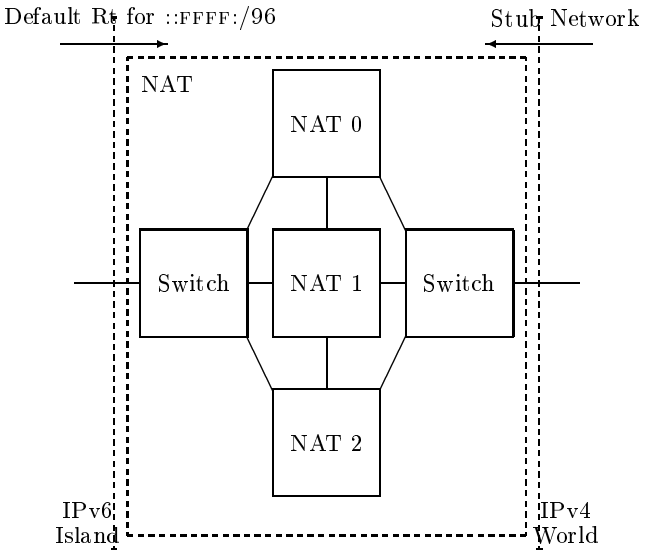


Figure 2: Configuration with failover capability

5. A PROTOCOL FOR STATE-SHARING

One of the objections to NAT as practiced today is that it weakens scalability and availability of a network. Our state-sharing protocol coordinates a set of NAT devices that provide redundancy and high throughput. An intentional side effect of explicitly defining the redundancy protocol is that software upgrades can be done on a live system by replacing one NAT at a time.

As shown in Figure 2, multiple NAT devices can sit between a pair of fast switches on the edge of the IPv6 island. The switches can fail too, cutting off the IPv6 island from the IPv4 world. One would use multiple such switches on either side of NAT in a Klos network formation to safeguard against switch failures. Switches are stateless as far as NAPT-PT is concerned.

One of the NAT devices (or a separate host) is designated as a Master. All NAT devices report state changes to the Master and request state information from the Master. Certain state changes are also brokered by the Master.

Let O_k denote the pool of globally routeable IPv4 addresses available to the IPv6 island. For each address in the pool, we have roughly 64K port numbers available (we leave out low numbered ports). At any instant of time, each NAT device has "ownership" of a subrange of the range of possible port numbers for every public address O_k. Ownership over a subrange means that the NAT device can immediately assign port numbers from that subrange. Ownership of a subrange can change dynamically.

Each NAT device has a point-to-point link to the Master, either by direct network link or by an authenticated connection over the existing networks. Messages are size limited.

When a NAT device starts up, it contacts the Master and asks for the session table using an RTABLE request. The response is a sequence of TTABLE messages giving the current session table in some max size chunks. A NAT device can

request ownership of a port range by sending an RRANGE request to the Master which responds using a TRANGE message. The Master can ask a NAT device to cease being the owner of a port range by sending an RWITHDRAW message; the receiving NAT device confirms by sending a TWITHDRAW message. The recipient of an RWITHDRAW message does not have a choice (of retaining the range) – if it does not acknowledge the message within a specified amount of time to repeated messages, that device is considered dead and the controller reroutes all of its traffic to other NAT devices.

Each NAT device sends periodic updates (RUPDATE) of new sessions created or old sessions deleted to the Master. Master in turn relays them to all the other NAT devices. Each NAT device is obligated to send an RUPDATE message at regular intervals even if there are no changes to the state at the NAT device. Absence of these heartbeats trigger action by the Master to determine if a NAT device/some link is dead.

A NAT device cannot delete session entries it created without checking with others, because session packets may have arrived at any of the other NAT devices and we need to confirm that there has been no traffic for an extended time before expiring the session. The owner NAT does this by sending an RAGE request to the Master which checks with other NAT devices before sending a TAGE response. The Master is responsible for the clean-up of sessions created by a dead NAT device. It follows the same rules for session deletion as any other NAT device.

Should a NAT device die, the Master assigns another NAT device to respond to the IP addresses on the inside and outside that the dead NAT device used to respond to. The chosen NAT device uses gratuitous ARP to announce the change and will handle the dead device's traffic in addition to what it is handling already. This technique is the same as the one in Virtual Router Redundancy Protocol (VRRP, IETF RFC 2338) and Cisco's Hot Standby Router Protocol (HSRP).

This model clearly does not eliminate the single point of failure but simply pushes it to the level of the Master. The Master can have a passive double who shares the state with the Master and takes over the Master's role if it detects that the Master is dead.

The IPv6 island can obtain coarse grain load balancing among multiple NAT devices by making groups of subnets point to different NAT devices as the next hop for the destination prefix ::FFFF: /96 or by making all the NAT devices equally weighted next-hops for the destination prefix ::FFFF: /96 within the IPv6 island.

6. DISCUSSION

NAT is well known to cause complications with some IP protocols [4], although with its growing adoption in the IPv4-only world, protocol designers are finding it wise to design for NAT [5]. We will mention here some additional difficulties that arise in the IPv4/IPv6 context.

NAT sits between two networks with possibly very different properties including the link mtu on each side. If the hosts on the two sides of NAT agree on a maximum segment size at the beginning of a session, the session is free of mtu related problems. This is indeed the case with TCP sessions. If UDP packets arriving on the outside interface are too large (remember that the packet grows by 20 bytes when it enters the IPv6 side of NAT), NAT fragments the packet. This may not be a huge strain but it is better to avoid if possible. A much worse situation is when a NAT device receives fragments from the inside (or the outside). The fragments beyond the first one do not have the transport layer header that a NAT needs. The only way a NAT can cope with this is to reassemble the fragments on one side and refragment it for the other side. This requires queuing up all the fragments belonging to a packet, is a big drain on NAT's memory and is best avoided. However, we are considering performing fragmentation and reassembly in microcode since Shannon et al [6] report that 68% of UDP packets are fragmented. Note that in IPv4/IPv6 conversion, fragment reassembly is required to compute the UDP checksum, which is required in IPv6 but optional in IPv4; that same report shows that fragmented UDP packets are frequently missing the checksum. Similar problems are encountered in VPN or other tunnel end points [2].

Handling of DNS packets by NAT is slightly more interesting. A common practice is to use a DNS ALG (Application Level Gateway) to translate DNS information. If in response to a DNS query, an IPv6-only host gets back only an A record (an IPv4 address), what should the host do? The default behavior of the host resolver in the Plan 9 operating system is to ask for AAAA and A records and if the response has only an A record, to embed the returned IPv4 address in an IPv6 compatible IPv4 address and dial that destination. Not all resolvers behave this way – some tend to give up at this point. However, this is a simple fix and we recommend it. If all hosts in the IPv6 island exhibit this behavior, then we do not need a DNS ALG [12] for the IPv6 island. Hosts inside the island making DNS queries to servers in the IPv4 ocean get the usual responses which they know how to handle; hosts in the IPv4 ocean can not contact a host inside the IPv6 island since they don't have fixed IPv4 addresses and are hidden behind the NAT device. We have assumed that any services offered by the IPv6 island to the IPv4 ocean will be hosted on dual stacked machines that have global IPv4 addresses; for such services, DNS can also be served from a dual stacked host.

There are a variety of ways people have proposed dealing with IPsec through IPv4-only NAT, for example using UDP encapsulation. But in our context, the task is fundamentally impossible; TCP checksums on the IPv6 machine include a pseudoheader with IPv6 addresses and the IPv4 end will surely reject these.

7. CONCLUSIONS

We believe that during the course of the gradual deployment of IPv6, NAPT-PT protocol/address/port translation will be required and this functionality will move inwards from the very edge of the network to the edge of a provider network. Our preliminary results suggest that network processors are capable of significant speedup of NAPT-PT and firewall processing compared to general purpose processors. A

reasonably simple redundancy protocol allows further scaling through load balancing and the fault tolerance needed for deployment deeper in the network than is common today.

8. ACKNOWLEDGMENTS

We are grateful to the Tadpole team, Michaela Blott, Mike Coss, Bruce Ellis, and Ron Sharp for building the platform on which we could construct NAT, to Dong Lin for building the packet sampling system from which we were able to derive session size estimates and to Dave Presotto for numerous discussions and helpful comments on this paper.

9. REFERENCES

[1] A. T. Campbell, S. Chou, M. E. Kounavis, V. D. Stachtos, and J. Vicente. Netbind: A binding tool for constructing data paths in network processor-based routers. *Proc. Fifth Interational Conference on Open Architectures and Network Programming*, OPENARCH '02, June 2002.

[2] J. S. Denker, S. M. Bellovin, H. Daniel, N. L. Mintz, T. Killian, and M. A. Plotnik. Moat: A virtual private network appliance and service platform. *USENIX 13th Systems Administration Conference*, LISA '99, November 1999.

[3] M. E. Fiuczynski, V. K. Lam, and B. N. Bershad. The design and implementation of an ipv6/ipv4 network address and protocol translator. In *USENIX Annual Technical Conference*, June 1998.

[4] M. Holdrege and P. Srisuresh. Protocol complications with the IP network address translator (NAT). *IETF RFC*, 3027, January 2001.

[5] D. Senie. NAT friendly application design guidelines. *IETF RFC*, 3235, January 2002.

[6] C. Shannon, D. Moore, and K. Claffy. Characteristics of fragmented IP traffic on internet links. In *ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[7] R. Sharp, M. Blott, M. J. Coss, B. K. Ellis, D. Majette, and V. Purohit. Starburst: Building next generation internet devices. *Bell Labs Technical Journal*, To Appear, 2002.

[8] T. Spalink, S. Karlin, and L. Peterson. Evaluating network processors in IP forwarding. *Princeton University Technical Report*, TR-626-00, November 2000.

[9] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using netowrk processors. *Proc. 18th ACM Symposium on Operating System Principles*, SOSP '01, October 2001.

[10] P. Srisuresh and K. Egevang. Traditional IP network address translator (traditional NAT). *IETF RFC*, 3022, January 2001.

[11] P. Srisuresh and M. Holdrege. IP network address translator (NAT) terminology and considerations. *IETF RFC*, 2663, August 1999.

[12] P. Srisuresh, G. Tsirtsis, P. Akkiraju, and A. Hefferman. DNS extensions to network address translators (DNS_ALG). *IETF RFC*, 2694, September 1999.

[13] G. Tsirtsis and P. Srisuresh. Network address translation - protocol translation (NAT-PT). *IETF RFC*, 2766, February 2000.