Network Processors Applied to IPv4/IPv6 Transition

Eric Grosse and Lakshman Y. N., Bell Laboratories, Lucent Technologies

Abstract

We describe a high-speed IPv6-IPv4 gateway on an experimental board containing a pair of Intel IXP network processor chips, an FPGA, and a pair of TCAMs. The device is capable of supporting several hundreds of thousands of concurrent TCP/UDP sessions and sustaining close to the line rate on a GbE link. It provides an order of magnitude improvement in packet throughput over an implementation of the same functionality on a commodity PC. 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 of such gateways. Network processors have the flexibility custom silicon lacks and the speed generic microprocessors lack, and hence are especially well suited for early implementation of network elements such as this gateway between IPv6 islands and the IPv4 ocean. A major challenge in building a scalable middlebox is redundancy support for stateful failover and load balancing, again putting a premium on programmability.

The next-generation Internet Protocol, IPv6, became an Internet Engineering Task Force (IETF) standard in late 1995, motivated by concern that the IPv4 address space would soon be exhausted. The use of classless interdomain routing (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 limited 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, will be used [1].

In full deployment (as opposed to small trials) the dual stack approach will not be feasible due to unavailability of IPv4 addresses. Network address/port/protocol translation (NAPT-PT) will be required when hosts with only IPv6 addresses need to talk with legacy IPv4-only systems in the rest of the world unless every popular network service is hosted on dual stacked machines. Consequently, the NAPT-PT functionality is likely to move inward 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 heavy traffic, perform complex packet processing, and be failsafe. Designing the hardware and software to keep up with these increased demands on middleboxes at low cost is a challenge.

Application-specific integrated circuits (ASIC)s provide the highest performance for hardware speedups, but programmable network processors are becoming available that offer enough performance for fast path implementations while maintaining more flexibility for adapting to unpredictable markets such as IPv6 deployment. Most of the currently available network processors are 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 from them have been detailed in [2, 3], and some of the difficulties of programming them have been addressed in [4]. To further explore the suitability of network processors for performing middlebox functions, we have built an experimental 1 Gb/s 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 article is organized as follows. We first describe the context for using a middlebox performing NAPT-PT, and the network measurements by which we arrived at our design requirements. We then describe the architecture of our middlebox implementation on a network processor board and measure the performance of the hardware. For clarity in describing the mainstream data flow, we postpone to the last section our simple state-sharing protocol to support active failover.

IPV6-IPV4 Gateway Requirements

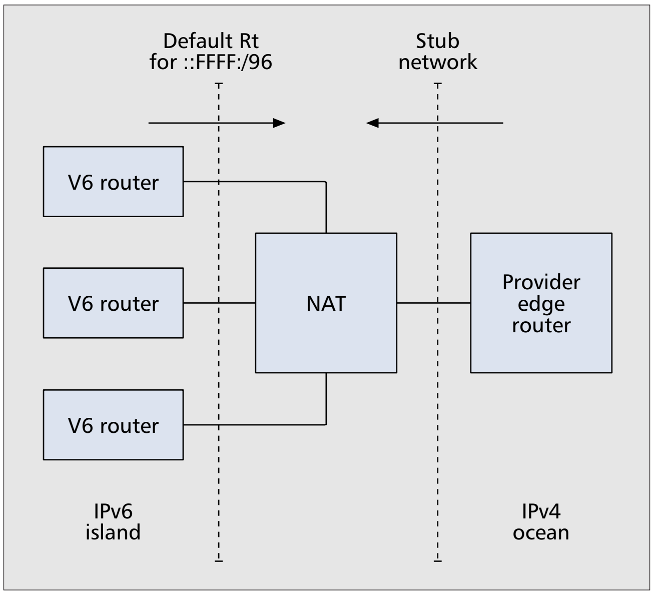
We are concerned here with the requirements of an organization or Internet service provider that needs to provide Internet connectivity to a large number of hosts but has a much smaller number of global IPv4 addresses at its disposal and hence has decided to build an all IPv6 network. It is very likely that at its inception, this IPv6 network is going to be an island in the IPv4 ocean. Hosts in the IPv6 island will need ways to talk to hosts in the IPv4 ocean as well as hosts in other IPv6 islands. Some fraction of the available IPv4 addresses can be assigned to dual stack machines for hosting services reachable by IPv4 as well as IPv6 clients. There will also be tunnel endpoints (6-in-4 automatic/configured/2002-prefix) for hosts making connections to hosts in other IPv6 islands across the IPv4 ocean [5]. For hosts that want to access services available only in the IPv4 ocean, short of proxying every such service, the only choice left is to perform network address translation. Several flavors of network address translation are in use [1, 6].

- 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 possess 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 NAPT-PT — 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 64,000 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. NAPT-PT is a well established, if not yet widely deployed, mechanism (e.g., [7]), 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 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 in this context.

Our design requirements were influenced by observing all packets on a lightly loaded fractional T3 link from Bell Laboratories Research to the outside world for several 12-hour periods. In a typical (6 a.m. to 6 p.m.) case we observed a total of 29 million packets, of which 94 percent were TCP and 5 percent were UDP, with a total of 670,000 sessions, for an average of 43 packets/session. Since the distribution of session lifetimes is longtailed, one cannot estimate from the raw packet counts a crucial parameter: how large the session table needs to be. So we ran a NAPT-PT simulation on the traces, using a TCP timeout of 1 min after the second FIN and a UDP timeout of 5 min. The peak number of sessions was not very sensitive to these parameters. We arbitrarily cut off TCP sessions after 6 h of inactivity; we observed such connections on ports for telnet, ssh, http, and lpr, but rarely; 98.8 percent 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 NAPT-PT, one should allow for hundreds of thousands of peak active sessions. We assume that sessions do not 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 leads to a predicted 320,000 sessions for Gigabit Ethernet (GbE), or 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 IXP, would create a significant bottleneck when 2.5 percent (≈ 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.



■ Figure 1. *The NAPT-PT's location in the network.*

NAPT-PT Architecture

NAPT-PT sits on the boundary between the provider IPv6 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 table entry for the IPv6 destination prefix ::ffff:/96 (IPv4 mapped IPv6 addresses) pointing to the NAPT-PT device (Fig. 1). Admittedly, this setup is not the most general one possible, but it provides a useful simplification — the IPv6 island and IPv4 ocean do not have to exchange routing information (if they did, the routing information would need to undergo IPv6-IPv4 transformation — a complicated and error-prone process, best avoided).

Packets passing through a NAPT-PT 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, etc.), a destination address *d6* (IPv4-mapped IPv6 address), and a destination port *dp*. For each flow, NAPT-PT creates a mapping,

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

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 NAPT-PT device sees an outgoing packet belonging to a session 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. ICMP packets often bear another IP packet as payload, so the header rewriting must be performed on that payload as well. Once a NAPT-PT knows the mapping for the session to which a packet belongs, it rewrites the packet's header (IPv6 to IPv4 for outgoing packets, IPv4 to IPv6 for incoming packets) in the obvious way including IP header checksum for newly generated IPv4 packets and puts out the packet on the appropriate interface. Details of the translation can be found in [8].

We have implemented our NAPT-PT device on a Bell Laboratories designed network processor board [9]. The board has two Intel GC-IXP1200-FCs running at 232 MHz, a high density field programmable gate array (FPGA), and two TCAMs. The FPGA handles communications among daughter cards (GbE, multiple 100 Mb Ethernet cards), the ternary content addressable memories (TCAMs), and the IXPs.

Recall that each Intel IXP chip has a StrongARM core and six microengines each with four hardware contexts. 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 2000 instructions. Each IXP has available off-chip 256 Mbytes of SDRAM and 8 Mbytes of SRAM and a small amount(4 kbytes) 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.

On our experimental boards, the FPGA mediates packets between the IXPs and the various input/output (i/o) ports (to the IXPs, the packets appear to be coming from the IX bus; the FPGA sits between the IX bus and the i/o ports). The designers of the board provide support for packet ingress and egress in the form of a macro library. Each IXP uses one microengine exclusively for receiving and one for transmitting packets from/to external ports. 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 processes the packet and puts it on an egress queue from where it is drained out to the appropriate port by the transmit engine. The microengines process normal TCP/UDP packets. IP header options, fragmented packets, ICMP, and ARP are all passed on to the StrongARM for processing.

We use the two IXP chips on the board to perform packet processing. Each IXP maintains a session table (of all the current sessions between the IPv6 island and the IPv4 ocean) in its SDRAM. Either of the processors can create/delete a session entry. Changes to the 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 entries each with a 144-bit-wide key and 32 bits of associated information.

The session table is organized as a hash table with linear chains for collision buckets. A hash table lookup involves computing a hash value from the packet header, reading a

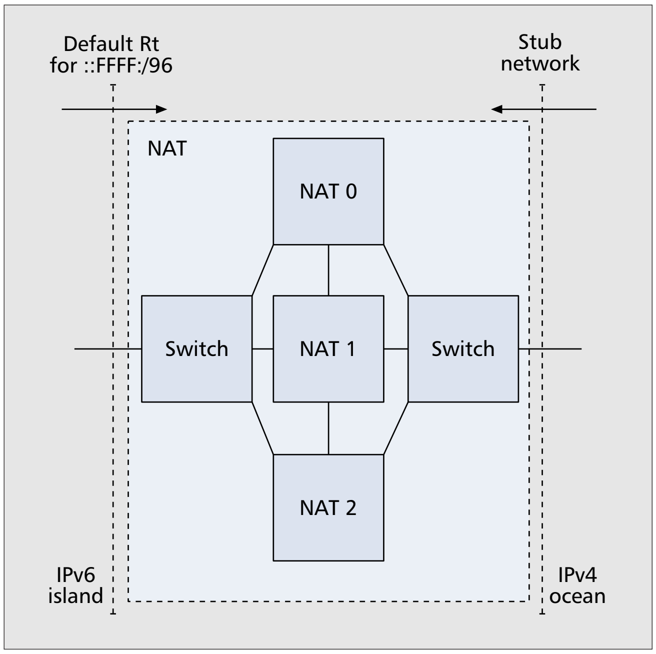


Figure 2. *The configuration with failover capability.*

session table entry using the hash value as an index, and checking against the packet header fields, repeating the last step until a matching entry or end of a collision chain is reached. Each IXP holds the session table in its SDRAM. We use the TCAM as a cache for session table entries. A complete session table entry is large and therefore in the TCAM we store a key, consisting of the 144-bits of v6 source address and source port and an 18-bit session table index hint. The rationale here is that most hosts, while initiating a TCP/UDP connection, do not re-use a (source address, source port) pair.

Outbound packets (arriving from the IPv6 island) are picked up by the FPGA, which then invokes the TCAM lookup and annotates the packet with the 18-bit result, the interface, and the packet size. The FPGA then passes the packet to one of the IXPs in strictly alternating order for load balancing.

In the case of a TCAM match, the receiving IXP uses the 18-bit 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. Thus, a successful TCAM lookup saves the hash value computation (not very significant) and provides the correct session table entry in a single memory access.

If there is no match in the TCAM (or a false hit), the IXP performs a full hash table lookup, which can be expensive if there are hash collisions. The TCAM is much more helpful for firewall rule matching, where an incoming packet header needs to be matched against multiple rules; any general rule matching algorithm implemented on the IXPs will require many more memory accesses than a single hash table lookup.

When an IXP creates a new session entry based on a received packet or finds a missing TCAM entry, it sends a control packet to the FPGA to add that session entry to the TCAM. In the same control message, the IXP 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 IXPs to maintain state information related to LRU. The two IXPs divide the TCAM address space into mutually exclusive halves to ensure that the two IXPs never try to add a TCAM entry at the same TCAM address simultaneously (one IXP

Packet size (type)	Direction	Packets per second	Mb/s
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

■ Table 1. *Throughput numbers.*

uses odd TCAM addresses, the other even). When an IXP 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. The FPGA also reflects the control packets to the remaining IXP, which then updates its session table as necessary. Thus, there is a small window of time (on the order of a few hundred nanoseconds) in which the two IXPs have slightly different session tables. In principle, it can happen that two back-to-back packets on the same flow (a TCP connection/UDP flow) arrive at the FPGA within such a window and get sent to different IXPs, and each IXP creates state (different) for the same flow. It is unlikely, however, considering that the NAPT-PT device sees mostly aggregate traffic. Also, back-to-back TCP SYNs have to be separated by time intervals much larger than the time needed to achieve session table consistency between the two IXPs; hence, this appears to be a reasonable engineering decision.

Packets arriving on a port from the IPv4 ocean are passed on to one of the IXPs in alternating order without a TCAM lookup. The session lookup in this direction is simple indexing based on the IPv4 destination address/port for the packet that is fast (the reverse index points directly to the corresponding entry in the hash table; one need not go through hash chains).

Performance

We have performed stress tests on the NAPT-PT device by connecting its network interfaces to a traffic generator (a Shomiti Explorer) capable of generating and capturing packets at wire speeds on GbE links. The test consisted of configuring the Shomiti to send a small number of fixed types of packets to one of the ports of the NAPT-PT device and capturing 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 (except the first one in a session) 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 (Table 1).

The packet sizes include the FPGA header (8 bytes), an ether header (14 bytes), IP header (20 bytes for IPv4, 40 bytes for IPv6) and a TCP header (20 bytes). For minimal-sized IPv6 packets (82 bytes), one needs to be able to process about 1.5 million packets/s to saturate a GbE link. We have not achieved that. We could not ascertain whether the IXPs were the bottleneck or the FPGA.

The most important table row is the bottom one, showing that for bidirectional small packets (the worst case) our implementation can keep up with a 50 percent utilized GbE link. At slightly larger packets sizes of 120 bytes, it can handle full wire speed.

In comparison, the packet processing code performing the same function as one of the IXPs on a general purpose board with an 866 MHz Pentium III running the Plan 9 operating system took about 20 μs/packet (not measuring the time spent in the IP stack), giving a packet throughput of about 50,000 packets/s. The figure might be significantly improved by carefully tuning our implementation and the drivers used, but it is unlikely that we will be able to approach the throughput numbers achieved on the network processor board.

A single session table entry is a 7-tuple,

[*s6*, *sp*, *proto*, *d6*, *dp*, *s4*, *nsp*],

occupying 43 bytes. The total storage required for the session table can become a concern as the number of active sessions grows into hundreds of thousands. We store *s6* as 2 bytes of a prefix identifier and 8 bytes of host identifier, *d6* as a 4-byte entity, and *s4* as a 2-byte identifier. This brings down the per session storage requirement to 23 bytes, which we round off to 24 bytes for word alignment. Decoding these fields adds a few IXP instructions but no extra memory accesses, so the compression is well justified.

A Protocol for State Sharing

One of the objections to NAPT-PT as practiced today is that it weakens reliability of the network by introducing points of failure in the middle of the network. We have designed a state-sharing protocol to coordinate a cluster of NAPT-PT devices that provide redundancy. An intentional side effect of explicitly defining the redundancy protocol is that software upgrades can be done on a live system by replacing one NAPT-PT device at a time. For easier reading, we replace the term *NAPT-PT device* with *gateway* in this section.

Our protocol is to be implemented among multiple gateways. The protocol is to be executed by the StrongARM core on one of the IXPs on each board. The protocol itself does not add significant traffic of its own, and by enabling active-active redundancy, provides higher total throughput by means of load balancing. We have implemented the protocol on general-purpose PCs since multiple copies of the experimental board were difficult to procure. Further experimentation is required to achieve fine tuning of the protocol parameters.

As shown in Fig. 2, multiple gateways 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 ocean. One would use multiple such switches on either side of NAPT-PT in a Clos network formation to safeguard against switch failures. Switches are stateless as far as NAPT-PT is concerned.

One of the gateways (or a separate host) is designated as a master. All gateways report state changes to the master and request state information from the master. Certain state changes are also brokered by the master. Each gateway has a point-to-point link to the master, by either a direct network link or an authenticated connection over the existing networks. Messages are size limited.

Let O_k denote the pool of globally routable IPv4 addresses available to the IPv6 island. For each address in the pool, we have roughly 64,000 port numbers available (we leave out low numbered ports). At any instant of time, each gateway has ownership of a subrange of the range of possible port numbers for every public address O_k. Ownership over a subrange means the gateway can immediately assign port numbers from that subrange. Ownership of a subrange can change dynamically.

When a gateway 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 gateway 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 gateway to cease being the owner of a port range by sending an RWithdraw message; the receiving gateway 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, the device is considered dead and the controller reroutes all of its traffic to other gateways. Each gateway sends periodic updates (RUpdate) of new sessions created or old sessions deleted to the master. The master in turn relays them to all other gateways.

Each gateway is obligated to send an RUpdate message at regular intervals even if there are no changes to the state at the gateway. Absence of these heartbeats triggers action by the Master to determine if a gateway/some link is dead.

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

Should a gateway die, the master assigns another gateway to respond to the IP addresses on the inside and outside to which the dead NAPT-PT device used to respond. The chosen gateway 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; it 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 gateways by making groups of subnets point to different gateways as the next hop for the destination prefix ::ffff:/96 or by making all the gateways equally weighted next hops for the destination prefix ::ffff:/96 within the IPv6 island.

Conclusions

Network address translation is well known to cause complications with some IP protocols [10], although with its growing adoption in the IPv4-only world, protocol designers are finding it wise to design for NAPT-PT [11]. We believe that during the course of 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 provides the fault tolerance needed for deployment deeper in the network than is common today.

Acknowledgments

We are grateful to the Tadpole team, Michaela Blott, Michael Coss, Bruce Ellis, and Ronald Sharp, for building the platform on which we could implement NAPT-PT and accommodating our numerous requests for TCAM lookup support in the FPGA. We thank Dong Lin for building the packet sampling system from which we were able to derive session size estimates, and David Presotto and the editors and referees for helpful comments on this article.

References

[1] P. Srisuresh and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations," IETF RFC 2663, Aug. 1999.
[2] T. Spalink, S. Karlin, and L. Peterson, "Evaluating Network Processors in IP Forwarding," Princeton Univ. tech. rep. TR-626-00, Nov. 2000.
[3] T. Spalink *et al.*, "Building a Robust Software-based Router using Network Processors," *Proc. 18th ACM Symp. Op. Sys. Principles.*, Oct. 2001, pp. 216–29.
[4] A. T. Campbell *et al.*, "Netbind: A Binding Tool for Constructing Data Paths in Network Processor-Based Routers," *Proc. OPENARCH '02*, June 2002, pp. 91–103.
[5] R. Gilligan and E. Nordmark, "Transition Mechanisms for ipv6 Hosts and Routers," IETF RFC 1933, Apr. 1996.
[6] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)," IETF RFC 3022, Jan. 2001.
[7] M. E. Fiuczynski, V. K. Lam, and B. N. Bershad, "The Design and Implementation of an ipv6/ipv4 Network Address and Protocol Translator," *USENIX Annual Tech. Conf.*, June 1998, pp. 201–12.
[8] G. Tsirtsis and P. Srisuresh, "Network Address Translation — Protocol Translation (NAT-PT)," IETF RFC 2766, Feb. 2000.
[9] R. Sharp *et al.*, "Starburst: Building Next Generation Internet Devices," *Bell Labs Tech. J.*, vol. 6, no. 2, 2002, pp. 6–17.
[10] M. Holdrege and P. Srisuresh, "Protocol Complications with the IP Network Address Translator (NAT)," IETF RFC 3027, Jan. 2001.
[11] D. Senie, "NAT Friendly Application Design Guidelines," IETF RFC 3235, Jan. 2002.

Biographies

ERIC GROSSE (ehg@lucent.com) heads the Secure Networking Research Department at Bell Laboratories, Murray Hill, New Jersey. He has over two decades of experience in computing and communications, with a career that has ranged over algorithms for approximation and visualization, network services, and security. He received his Ph.D. in computer science from Stanford University on tensor splines, founded the netlib repository of mathematical software, and more recently was involved in design and management of the prototype Lucent Managed Firewall, the Viaduct VPN, and redesign of security in the Plan 9 operating system.

LAKSHMAN Y. N. (yagati@lucent.com, ynl@plan9.bell-labs.com) is a member of technical staff in the Secure Networking Research Department at Bell Laboratories, Murray Hill. He received his Ph.D. in computer science from Rensselaer Polytechnic Institute on computer algebra. Prior to joining Bell Laboratories, he was an associate professor of computer science at Drexel University, Philadelphia, Pennsylvania. His current research interests are in high-speed networking, network processors, high-speed wireless data networks, and distributed computing.