TOWSON UNIVERSITY

OFFICE OF GRADUATE STUDIES

Network Address Translation, 6to4 Tunneling, and IVI Translation on a Bare PC

By

Anthony Kofi Tsetse

A Dissertation

Presented to the Faculty of

Science

in partial fulfillment of the requirements for the degree

Doctor of Science

in

Information Technology

Department of Computer and Information Sciences

Fall 2012

TOWSON UNIVERSITY

OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by **ANTHONY KOFI TSETSE** entitled **Network Address Translation, 6to4 Tunneling, and IVI Translation** on a Bare PC has been approved by the dissertation committee as satisfactorily completing the dissertation requirements for the degree Doctor of Science.

Dr. Alexander Wijesinha Chair, Dissertation Committee		8/14/12 Date
Dr. Ramesh Karne Committee Member		8/14/12 Date
Dr. Marius Zimand Committee Member		8-14-2012 Date
Dr. Wei Yu Committee Member		8-14-2012 Date
Dr. Janet V. DeLany Dean of Graduate Studies		8-17-2012 Date

ACKNOWLEDGEMENT

To God be the glory for great things he continues to do in my life. I would like to thank my advisor, Professor Alexander Wijesinha, for supporting me over the years, and for giving me so much freedom to explore and discover new areas of knowledge. He has shared his great creative and analytical skills with patience, endless encouragement, and generosity. It has been my very good fortune to have had such a conscientious and inspirational advisor. Professor Ramesh Karne has been exceptionally helpful for giving me more insight into Bare Machine Computing and spending endless times helping with debugging most of my source code. Without his support, this dissertation would not have been a reality. To my other committee members; Professors Marius Zimand and Wei Yu, I am highly indebted to you for agreeing to serve on my committee and also offering me the necessary guidance.

To my colleagues in the Bare Machine Computing Lab, Dr Patrick Appiah-Kubi, Alae Loukili, Dr. Bharat Rawalkshatriya, and Uzo Okafor, I say thank you for your various inputs into this dissertation.

I will like to thank my family for the support and encouragement all this time. You have always inspired me to greater heights and for this, I am highly appreciative.

Finally, to my dear wife Evelyn, you have always stood by me in times of difficulty challenging me to pursue my dreams and letting me realize my potential even when all seem to have been lost. The days and nights spent without you and the kids have finally paid off.

DEDICATION

This dissertation is dedicated to:

My parents

Mr. Bernard Tsetse and Mrs. Mary Tsetse

My wife, Evelyn Tsetse

and kids,

Malvin and Mirabel

ABSTRACT

Network Address Translation, 6to4 Tunneling, and IVI Translation on a Bare PC

By

Anthony Kofi Tsetse

The next-generation Internet is expected to have a long transition period during which Internet Protocol versions 4 and 6 (IPv4 and IPv6) will co-exist. This implies continued use of gateways supporting private/public address translation in IPv4 networks, and an increased use of gateways that are capable of IPv4/IPv6 tunneling or address translation. This dissertation deals with the design, implementation, and performance of bare PC gateways that do the functions of Network Address Translation (NAT), 6to4 tunneling, and IVI (IPv4/IPv6) translation. We describe the architecture of bare PC gateways, and compare their performance with conventional Linux gateway implementations using the same hardware in a test LAN environment.

Bare PC systems, including bare PC gateways, enable software to run directly on ordinary PC hardware without using any operating system or kernel. They are of interest to builders of minimalist platforms that serve as an alternative to feature-rich systems. In addition to their performance advantages, bare PC systems provide fewer opportunities for attack.

We first consider a bare PC NAT device and its performance. NAT (Network Address Translation) is a critical function in IPv4 networks that occurs at the boundary of all private and public networks including ISP boundaries in homes and businesses. Our results show that the bare PC NAT has better performance than the Linux NAT with respect to inbound and outbound packet processing time, and throughput, regardless of packet size and payload application type. We show in particular that there is a 34% improvement in the maximum number of packets per second (pps) over Linux under heavy traffic. Internal timings on the bare PC NAT box indicate further that there is plenty of capacity left for implementing supplementary functions such as deep packet inspection and routing if needed.

We next consider a bare PC 6to4 gateway and study performance of 6to4 tunneling with and without NAT co-location. 6to4 tunneling is a transition mechanism for enabling IPv6 devices and networks to connect to today's Internet, which is primarily IPv4-based. Our results show that performance using 6to4 with a co-located NAT is better than with a decoupled NAT regardless of whether a Linux or a bare 6to4 gateway is used. In general, performance improvements with a co-located versus a decoupled NAT range from 34%-57% for the bare PC gateway and 7%-45% for the Linux gateway.

Furthermore, performance improvements for a bare PC versus a Linux gateway range from 23%-86% with co-located and decoupled NATs.

Finally, we consider IVI translation. The stateless IPv4/IPv6 translation technique known as IVI is a relatively new approach that provides connectivity between IPv4 and IPv6 hosts in the Internet. Evaluating IVI overhead, we find that address mapping is the most expensive function in the translation algorithm. Moreover, translating IPv4 packets to IPv6 packets has more overheard than translating in the reverse direction.

This research shows that bare PC gateways can be used as low-cost gateways in the next-generation Internet. It also characterizes the performance trade-off between using Linux gateways with more features and bare gateways with only minimal features that are necessary for its purpose.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Currently, there is considerable interest in using “minimalist” platforms in the next-generation Internet for a variety of reasons [1]. While such “barebone” systems would likely contain some form of an Operating System (OS), it is nevertheless useful to consider the extreme case of running software directly on the bare hardware with no intermediary or “layer” in between them. Running applications on a bare machine or bare PC without an OS gives the end-user flexibility to use the application anytime and anywhere without having to worry about OS patches, compatibility issues etc. It also improves performance and security. Bare machine systems are more secure than conventional systems since there are fewer avenues open for attackers to exploit.

Network address translation (NAT) was introduced primarily to address the issue of Internet Protocol version 4 (IPv4) 32-bit address exhaustion by using private addresses within internal networks. However, continual growth of the Internet has resulted in the rapid depletion of public IPv4 addresses assigned by the Internet Assigned Numbers Authority (IANA) and regional Internet registries (RIR).

The next-generation IP or Internet Protocol version 6 (IPv6) provides 128-bit addresses that resolve the problem of insufficient address space. IPv6 deployment has been quite slow due to a variety of reasons, including the incompatible nature of IPv4 and IPv6 protocols. Since IPv4 networks still form a major part of the Internet, it becomes necessary for mechanisms to be adopted that enable communication between IPv6 and IPv4 nodes/networks during the transition period, which is believed to be long.

To gain insight into transition-related performance issues, it is useful to study the performance of gateways that implement NAT and proposed transition mechanisms including 6to4 tunneling, and IVI translation.

The focus of our research is the implementation and performance of gateways that run on a bare PC. Such gateways could be deployed in secure networks due to their intrinsic immunity against typical OS-based attacks. Given the importance of NAT in IPv4 networks, it is also necessary to implement NAT functionality in the bare gateways. More details on bare PC systems and their use as gateways are given later in this dissertation. Our studies investigate the extent to which OS overhead impacts the performance of NAT, 6to4 tunneling, and IVI translation by comparing the performance of bare PC and Linux gateways with these functions. The main contributions of our research include:

I. The design and implementation of NAT, 6to4 gateway, and IVI translator applications that run on a bare PC.
II. A performance study of 6to4 tunneling with a co-located NAT showing that performance of a 6to4 gateway is better with a co-located NAT than with a decoupled NAT, and that there is a significant performance improvement due to using a bare PC NAT or 6to4 gateway with no OS overhead than its Linux counterparts.
III. A performance study of the OS overhead in IVI translation showing that translating packets from IPv4 to IPv6 is more than from IPv6 to IPv4, and that address mapping is the most expensive operation.

The rest of this dissertation is organized as follows. In Chapter 2, we provide background information on Bare Machine Computing, NAT, 6to4 tunneling and IVI translation. We also discuss related work in this chapter. In Chapter 3, we discuss the design and implementation of a bare PC NAT application and its performance. In Chapter 4, we describe the implementation of a 6to4 gateway on a bare PC, and study the performance of 6to4 tunneling with a co-located NAT. In Chapter 5, we describe the design and implementation of an IVI translator running on a bare PC, and compare the performance of our implementation with that of a Linux-based IVI translator. In Chapter 6, we summarize our contributions and present the conclusion. The Appendix consists of tables containing the numeric data from our performance studies used to obtain the results discussed in Chapters 3-5.

Chapter 2

RELATED WORK

2.1 Bare Machine Computing

Bare Machine Computing (BMC) [2] provides a platform for application development with no operating system, kernel, or other form of centralized control. Bare machine applications (often called bare PC applications since applications currently run on PC hardware) are implemented within a small self-contained program called an Application Object (AO) [3]. The AO contains the necessary elements to manage itself during execution. Figures 1 and 2 [2] show OS-based and BMC application architectures.

The BMC paradigm was also known as the Dispersed Operating System Computing (DOSC) paradigm [4]. It allows application developers to have full control over hardware resources through APIs, and optimize CPU task scheduling. Bare PC applications have low overhead, small code size, lean intertwined protocols, and immunity against OS-based attacks. More details on AOs and application design using AOs are given in [4]. There are numerous publications dealing with a variety of bare PC applications including Web servers, email servers, SIP servers, VoIP softphones etc. For example, a bare PC Web server is described in [5]. The aspects of BMC relevant to designing and implementing bare PC gateways are discussed in Chapters 3-5.

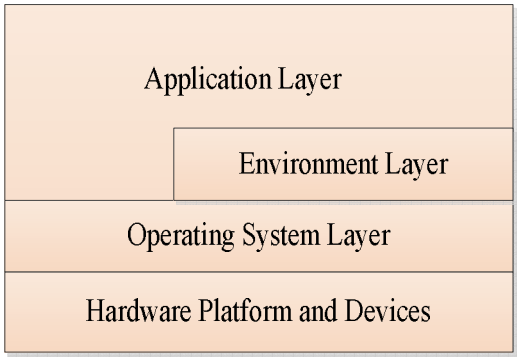

Figure 1. OS application architecture

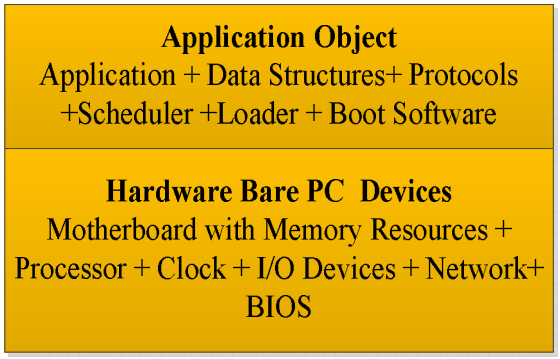

Figure 2. BMC application architecture

2.2 Network Address Translation

In NAT, the private (internal) IP address of the originating host device is translated to a common public (external) IP address representing the NAT device. In its most popular form, used in homes and corporate networks, NAT also involves translating the application TCP or UDP internal port number to an external port number [6]. Address/port number translation is done in all inbound and outbound packets. In most networks, a device handling NAT (also known as a NAT box) is usually co-located with a router and a firewall deployed at the entry/exit points of the network. NAT functionality can also be provided by security gateways used as tunnel endpoints in IPsec VPNs.

Well-known issues that arise due to using NAT are discussed in [7]. For example, to work in the presence of NAT, FTP requires special handling. NAT in general breaks the end-to-end TCP connection paradigm in IP networks [8]. NAT also makes it difficult to initiate direct communication between peers. STUN [9] attempts to address this problem for UDP. In [10], a host-initiated approach to NAT is proposed wherein hosts would perform some of the NAT functions thereby reducing the workload on the NAT device. In [11], a NAT router relays the MAC addresses of internal hosts on a LAN to external hosts enabling them to identify LAN hosts using their MAC addresses.

The design, implementation and performance of a programmable network address translator are the focus of [12]. The implementation and performance of a NAT gateway designed for signaling, and NAT traversal for peer-to-peer networks are discussed in [13] and [14] respectively. In contrast to previous work on NAT, this dissertation describes a novel network address translator that runs as an application on a bare PC with no OS. Such a NAT box is an alternative to a conventional NAT that would run on the hardware (typically supported by some form of an OS-most frequently Linux, or an adaptation of it).

2.3 6to4 Tunneling

Two types of tunneling techniques have been proposed for connecting IPv6 networks using the current global IPv4 infrastructure: configured and automatic tunneling. IPv6-over-IPv4 encapsulation was initially used by the 6bone project [15] to manually configure tunnels for IPv6 connectivity, but this method is not used in practice due to complexity involved in managing manually configured tunnels.

6to4 tunneling [16] is an automatic tunneling mechanism that requires no explicit tunnel setup. 6to4 gateways are dual stack devices that have both IPv4 and IPv6 functionality. Since these gateways connect to the global IPv4 infrastructure, they must each have a public IPv4 address. The IPv4 tunnel endpoints of 6to4 sites are advertised dynamically, thus eliminating the need for keeping state information. 6to4 sites trying to reach each other will discover the 6to4 tunnel endpoint from a Domain Name System (DNS) name through address lookup, and use a dynamically built tunnel from site to site for communication. A native IPv6 site trying to communicate with a 6to4 site will have to use a 6to4 relay.

Teredo [17, 18] (or its Linux version Miredo) is another automatic tunneling method to help nodes located behind NATs gain IPv6 connectivity. However, Teredo/Miredo requires relays and/or servers, and is intended to be used as a last resort due to its encapsulation overhead. Tunnel brokers [19] can use Tunnel Setup Protocol (TSP) to set up tunnel parameters and encapsulate IPv6 in IPv4, and vice-versa. Dual stack devices have implementations of both IPv4 and IPv6 protocols running independently; most operating systems support dual stacks. The dual stack approach only allows communication between like network nodes (i.e., IPv6-IPv6 and IPv4-IPv4). Translation mechanisms (discussed later) are used to translate the IPv6 headers into IPv4 headers and vice-versa.

Most studies dealing with the IPv4-IPv6 transition propose new transition mechanisms, or compare the performance of existing transition mechanisms. In [20], a hierarchical routing architecture to integrate both IPv4 and IPv6 networks is proposed. A performance study on configured tunnels and 6to4 mechanisms running on Linux is described in [21], while 6to4 and tunnel broker performance is compared in [22]. In [23], VoIP performance over IPv4 and IPv6 is evaluated using a bare PC softphone as a control to analyze the effect of OS overhead on VoIP call quality. VoIP performance with IPv6 and IPsec is studied in [24], and it is concluded that NAT and 6to4 processing (using a decoupled NAT) have little impact on VoIP quality in IPv6 networks.

Although there have been several studies on the performance of various IPv4/IPv6 transition mechanisms, our study differs in that it focuses on the performance of 6to4 gateways with a co-located NAT.

Our work also differs from previous work in that we describe the design and implementation of a 6to4 gateway application with NAT on a bare PC that runs without an OS. The study in [23] uses a bare PC softphone to study VoIP performance, while our study uses a bare PC as a network gateway device. Moreover, we use IPv6 background traffic for our measurements as opposed to [24], which uses IPv4 background traffic and primarily focuses on VoIP performance with 6to4 and IPsec (a bare PC is not used in that study). We also consider network performance with both VoIP and HTTP traffic passing through the 6to4/NAT gateways.

2.4 IVI Translation

Several techniques have been proposed for translating IPv4 packets to IPv6 packets and vice versa. One such technique, Network Address Translation–Protocol Translation (NAT-PT and NAPT-PT) [25], defined in RFC 2766 and subsequently recommended for historical status in RFC 4966 because of several issues, allowed a set of IPv6 hosts to share a single IPv4 address for IPv6 packets destined for IPv4 hosts. NAPT-PT also allows for mapping of transport identifiers of the IPv6 hosts. In RFC 3142 [26], the operation of IPv6-to-IPv4 transport relay translators (TRT) is discussed. TRT enables IPv6-only hosts to exchange two-way TCP or UDP traffic.

Current work on IPv4-IPv6 translation is discussed in a group of several related RFCs. A technique for network address and protocol translation from IPv6 clients to IPv4 servers (stateful NAT64) is described in RFC 6146 [27], wherein translation of IP/ICMP headers are done based on the recommendations stated in the Stateless IP/ICMP Translation Algorithm (SIIT) of RFC 6145 [28].

NAT64 itself uses an address mapping algorithm discussed in RFC 6052 [29] for IPv4-IPv6 address mapping. RFC 6147 [30] specifies the DNS64 mechanism to be used together with IPv6/IPv4 translators to enable IPv6 clients communicate with IPv4 servers using the fully qualified domain name of the server. These RFCs form the basis for the IVI translator and translation approach given in [31]. While the future evolutionary path of IPv6 deployment using the various means for co-existence with the current IPv4 infrastructure is unknown, it is expected that IVI translation will continue to be used as it is currently deployed in several Internet domains.

Chapter 3

NETWORK ADDRESS TRANSLATION ON A BARE PC

Network address translation (NAT) (as discussed earlier) is widely used in the Internet today for both IP address conservation [32] and security. In this chapter, we describe the design and implementation of a self-supporting NAT application that runs on a bare PC with no operating system (OS), and conduct experiments in a LAN environment to compare its performance with a NAT device running on Linux. While it is expected that the bare PC NAT application will have better performance than a Linux NAT due to the elimination of OS overhead, it is important to determine the extent of improvement when designing "barebones" network security devices that incorporate NAT functionality. The remainder of this chapter is organized as follows. In Section 3.1, we describe the design and implementation of the bare PC NAT application, and in Section 3.2, we present the experimental results.

3.1 Design And Implementation

The bare PC NAT box runs an application that contains an Ethernet driver, does IP packet processing, and implements NAT functionality. Traffic enters from either of its two interfaces and is forwarded or dropped according to the criteria described below. NAT performance depends on the ability to process packets at high speed and look-up/update the NAT table efficiently when there is heavy traffic and a large number of entries in the table.

Furthermore, the NAT application translates the address fields in the IP header and port number fields in the TCP or UDP header of received packets prior to forwarding them. It must also be capable of translating addresses and query IDs in ICMP packets (unless all ICMP traffic is dropped for security reasons).

The software architecture of the bare PC NAT application is shown in Fig. 3. Only two tasks main and receive (RCV) are implemented.

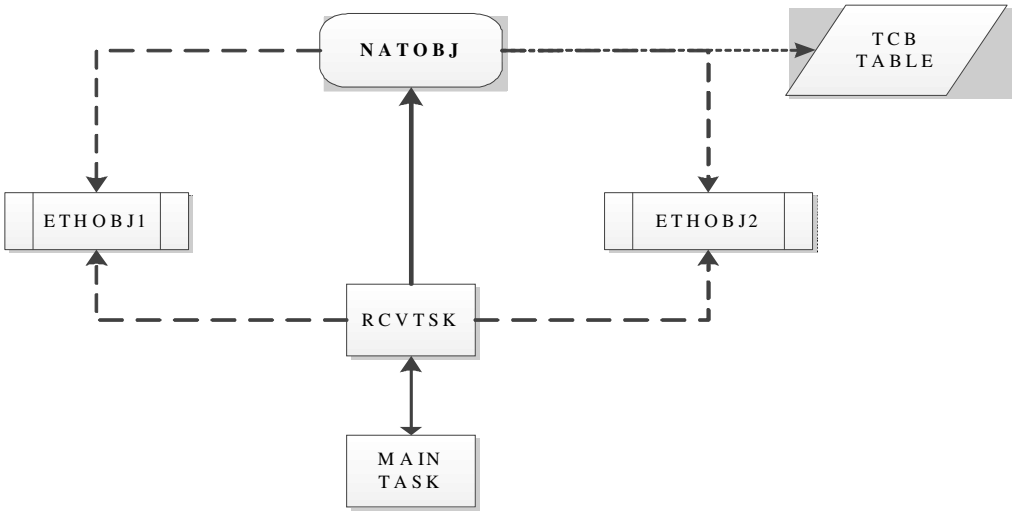

Figure 3. Software architecture of a bare PC NAT

These tasks are required by all bare PC applications (additional tasks are added as needed for a given application). In this case, the RCV task is responsible for handling received packets all the way from the Ethernet level through the IP level and NAT-related processing. The main task runs at start-up and whenever the RCV task terminates i.e., after a packet is forwarded.

As in all bare PC applications, the bare PC NAT application is started by invoking an executable boot program stored on a USB flash drive. The initial sector of the USB loads the menus that an administrator can use to select and configure the NAT application.

Once the NAT application is selected, a self-supporting bare PC application object containing the monolithic executable is loaded. Control is then passed to the main task. The Ethernet objects ETHOBJ1 and ETHOBJ2 represent the two network interfaces. Depending on the interface from which a packet arrives, the RCV task (RCVTSK in the figure) reads data from either ETHOBJ1 or ETHOBJ2, and a flag is set to indicate the relevant interface. Next, the RCV task calls the NAT Object (NATOBJ) which is responsible for NAT-related processing. The main data structure that stores the NAT table and other relevant protocol-related information is the Transition Control Block (TCB table). Depending on the source and destination IP address of the packet, after translating the headers, the NAT Table is updated accordingly, and the packet is either forwarded to the internal or external interface. We have implemented ETHOBJ1 and ETHOBJ2 as two instances of the same Ethernet object. However, this only works if both network interfaces are to Ethernets.

Fig. 4 shows the main packet processing logic in a bare PC NAT application. Whenever a packet is received, the IP header checksum is verified. If the checksum fails, the packet is discarded. Otherwise, the source and destination IP addresses are checked to determine if the packet entered from the external or internal interface. If the packet is from an interface and destined for the same interface, the packet is dropped. If the packet is from the internal “LAN” interface, NATOBJ checks the TCB table for an existing entry.

If an entry exists, its timeout value is refreshed and the translated packet is forwarded to the external interface.

If there is no existing entry in the TCB table, an entry is added prior to forwarding the packet to the external interface.

For a packet entering from the external interface, if a valid entry exists in the TCB table, the packet is forwarded to the internal interface; else the packet is dropped. Prior to forwarding a packet, the appropriate IP address and TCP or UDP port number (or ICMP ID) translations are made to the relevant packet header fields. In case ICMP packets are to be handled, the processing logic is similar (except that there are no port numbers). The AddEntry method was implemented as an overloaded method with different processing for ICMP and TCP/UDP packets. Once this method is called for an outgoing packet that has no prior entry in the table, it invokes the InsertTCB method. This method computes a hash value based on the IP address and port number and returns a pointer to the record number of the next free slot in the table (after doing a search if needed). The record number is the table index and also serves as the translated port (for TCP/UDP) or translated query ID (for ICMP).

Before a record for an outgoing packet is added to the NAT table, the SearchTCB method checks if there is an existing entry in the table that should be used. If so, this method computes a hash value (with a search if needed) and returns a pointer to the corresponding record in the TCB table. For incoming packets (from the external "WAN/Internet" interface), the source IP address and destination port is extracted.

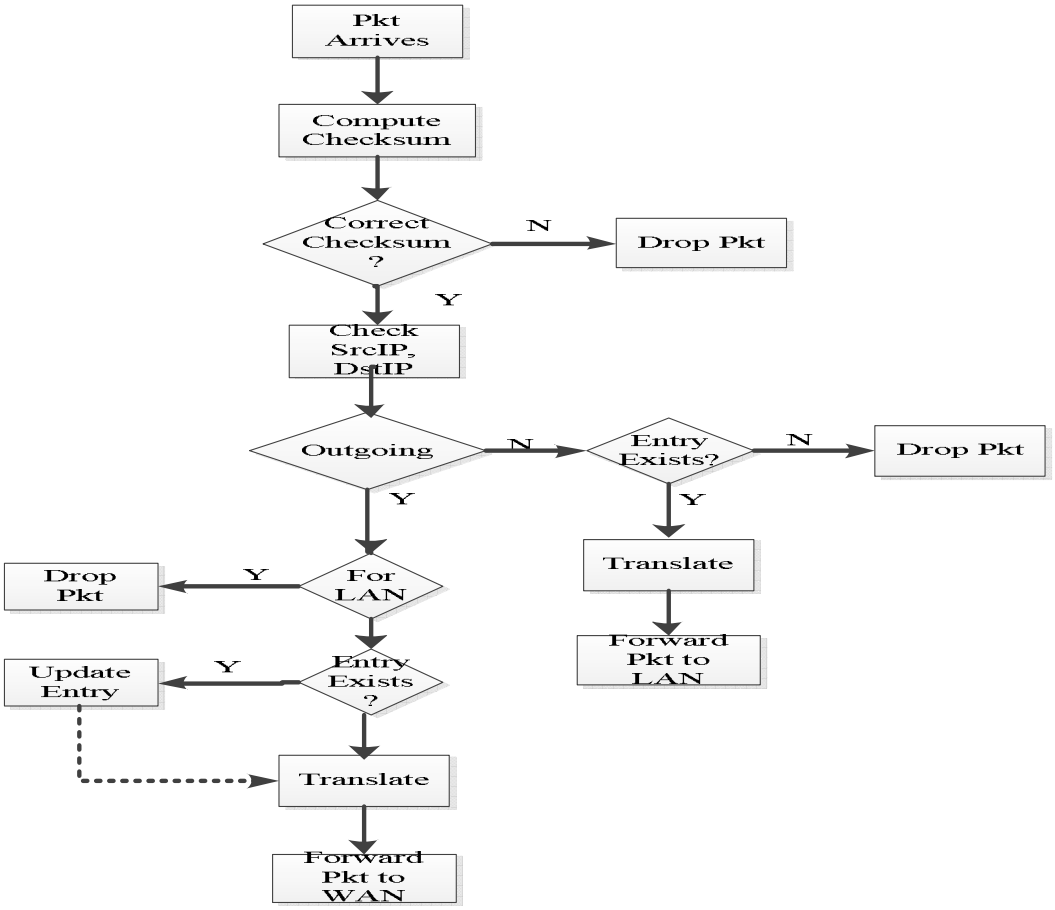

Figure 4. Packet processing logic

Since the destination port serves as an index to the TCB table, a search is not needed to look up the entry in the table.

In the case of incoming ICMP packets from the "WAN/Internet" interface, the query ID serves as an index. Depending on the age field in a NAT entry, it may be deleted from the NAT table. The TCB table, which is the primary data structure storing all the necessary protocol-related information, can hold a maximum of 100,000 TCB records.

Each NAT entry in the TCB table is of type TCB record with the following fields: source and destination IP addresses and ports, protocol, query ID (for ICMP), translated port, translated query ID, age, and availability (which indicates if an active entry is present in a slot). The average size of a NAT entry is 25 bytes. Microsoft Visual Studio was used for developing the bare NAT application. The main code for NATOBJ and other auxiliary protocols were written in C++. Since there is no OS, the application does not use any system calls. The drivers for the hardware interfaces (NIC, keyboard, display, and USB) were written using a combination of assembly and C code.

3.2 Experimental Results

The test LAN used for the performance study is shown in Fig. 5. All links and the two Ethernet switches were 100 Mbps and the NAT boxes to be tested were placed between the switches.

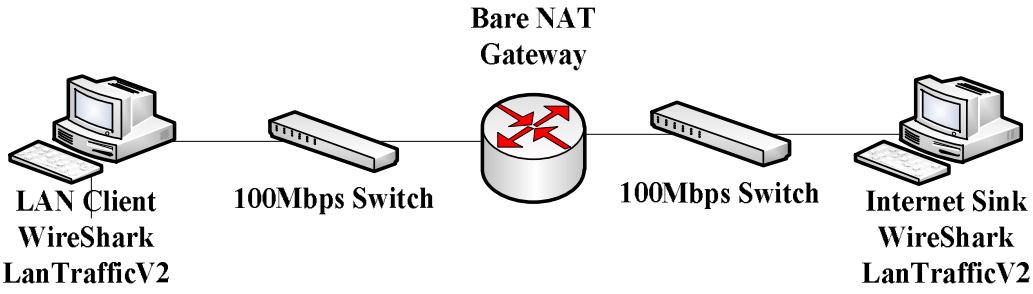

Figure 5. Test LAN

Although the figure only shows a single client, more clients were added as needed. We conducted experiments to study NAT performance under both normal and heavy traffic.

Three different configurations were used: no NAT, bare PC NAT, and Linux (Fedora) NAT. The detailed hardware and software specifications are given in Table I.

Three types of traffic (TCP, ICMP, and HTTP) were used to generate normal (moderate) loads as follows: first, the source clients were configured to establish 16 simultaneous connections via LanTraffic V2 [33] to the sink using different ports and different IP datagram sizes. These sizes were based on the guidelines given in [34]. For each size, the source node then continuously transmitted TCP data to the sink for 10 minutes and the data was collected using a Wireshark packet analyzer [35] and the LanTrafficV2 network tool. For ICMP, the ping command was used to directly transmit 200 packets between source and the sink for each datagram size. For HTTP traffic, http_load [36] was used to request files of varying sizes from an IIS Web server running on Windows 2008/server. To ensure fair comparison between the bare PC and the Linux (Fedora 12) NAT, all non-essential services on the latter were disabled with the exception of IPtables.

TABLE I. Hardware and Software specifications

PC/Switch	**Role**	**Hardware**	**OS/Software (non-bare only)**
Dell Optiplex GX270	bare/ Linux NAT Gateway; bare client	Pentium 4, 2.4 GHz, 512 MB RAM, 3Com 10/100 NIC, Intel PRO/1000	Fedora Core 12 (2.6.18-92.1.22) IP Tables
Dell Optiplex GX755	Webserver Sink, OS clients	Pentium 4, 2.4 GHz, 1GB RAM, Intel PRO/1000 NIC	Windows 2008 Server, Fedora Core 12 (2.6.18-92.1.22), http_load, LanTraffic V2, Windows XP SP3, Mgen5
Netgear GS108	Ethernet Switch		

To compare performance of the NAT boxes under heavy traffic, a) a mix of UDP and TCP traffic (80% TCP and 20% UDP) and b) TCP traffic only was generated.

For this purpose, bare PC TCP clients and Windows UDP clients running Mgen5 [37] were used as sources; and an IIS Web server running on Windows 2008/server (configured to have 10 different IP addresses) was used as the sink. The bare clients established TCP connections to the server using different source/destination IP addresses and port numbers.

3.2.1 Throughput and RTT

Fig. 6 compares the throughput for TCP traffic with and without NAT using LAN_Traffic V2. As expected, throughput without using NAT is higher compared to that using a bare PC/Linux NAT. However, the peak throughput is about 70 Mbps for the bare PC NAT and about 63 Mbps for Linux (i.e., an improvement of about 11%). Fig. 7 compares the round trip time (RTT) for ICMP traffic, which reflects the inbound and outbound NAT processing time. It can be seen that the processing times for the bare PC NAT are considerably less than those for the Linux NAT.

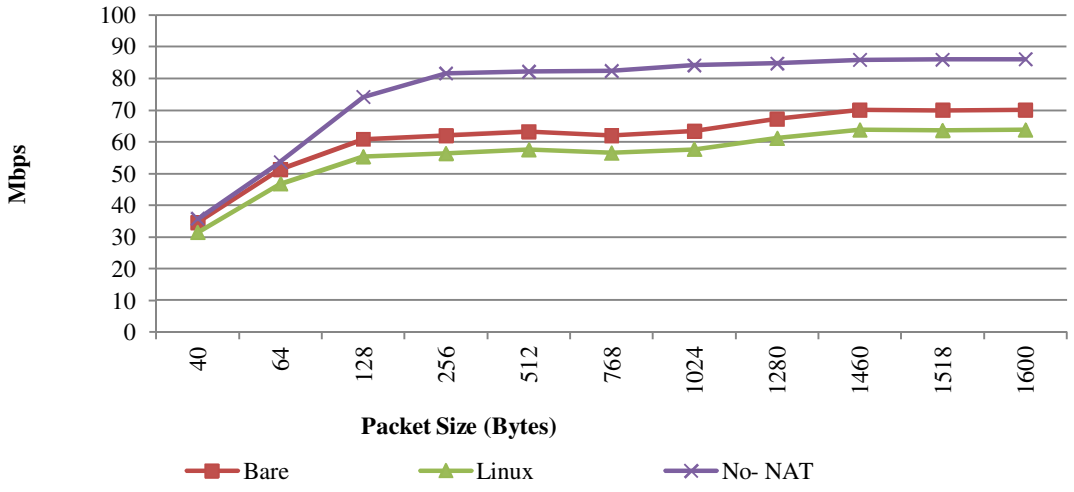

Figure 6. Throughput

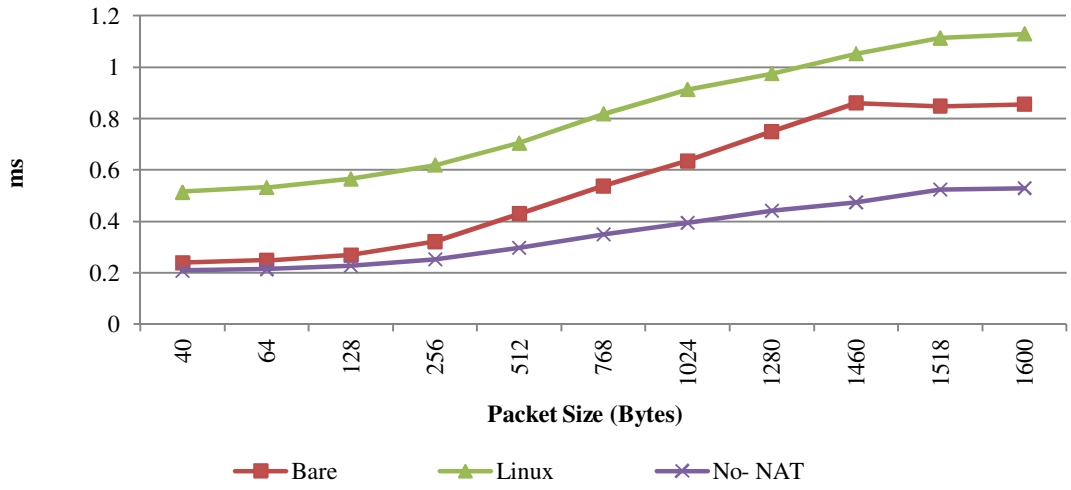

Figure 7. RTT (ICMP)

3.2.1 Connection and Response Times

To measure connection and response time, HTTP_load was used to generate requests. In Fig. 8, the connection time for the Linux NAT increases with increasing file size to a peak value of 0.262ms for a size of 32KB. For the bare PC NAT, a relatively constant connection time of 0.214ms is maintained irrespective of the file size since data copying is minimized.

In Fig. 9, the response times for the Linux and bare PC NAT are 0.522ms and 0.5ms respectively. In both cases, a small performance improvement of about 5% is attained by the bare PC NAT.

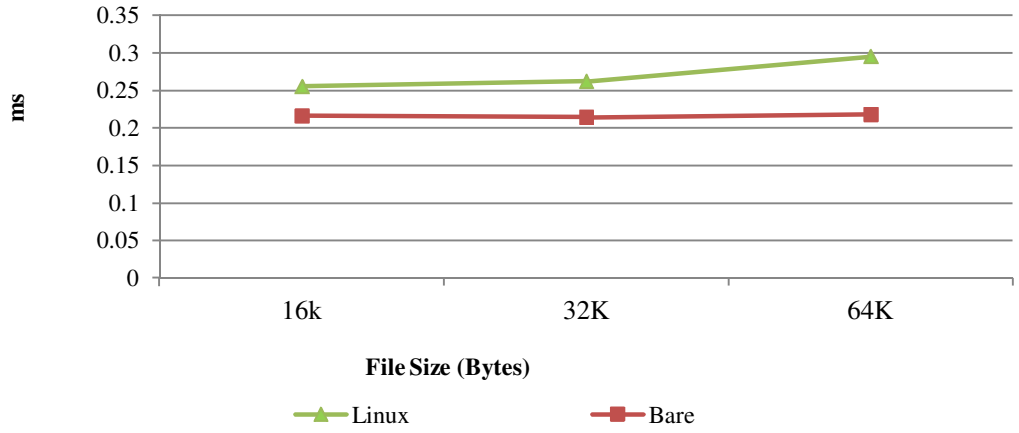

Figure 8. Connection time

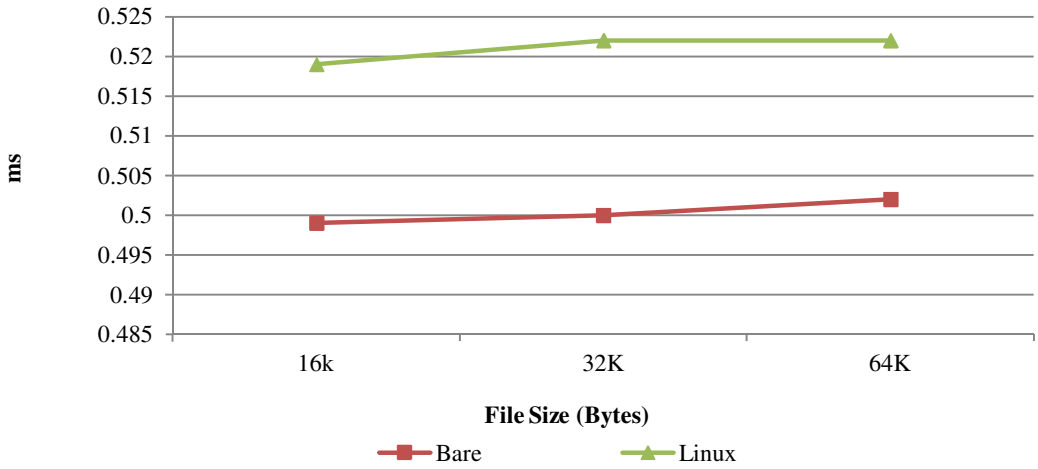

Figure 9. Response time

3.2.2 Packets Processed Per Second (PPs)

To measure pps, bare clients were used to generate TCP traffic. In Fig. 10, it is seen that the maximum number of packets processed per second (pps) by the Linux and bare NAT for TCP traffic only are 9,560 pps and 13,992 pps respectively (i.e., a 46% improvement). In Fig. 11, the corresponding maximum pps values for a traffic mix of UDP and TCP generated using bare clients and Mgen5 are 9,522 and 12,735 for the Linux and bare PC NAT respectively (i.e., a 34% improvement).

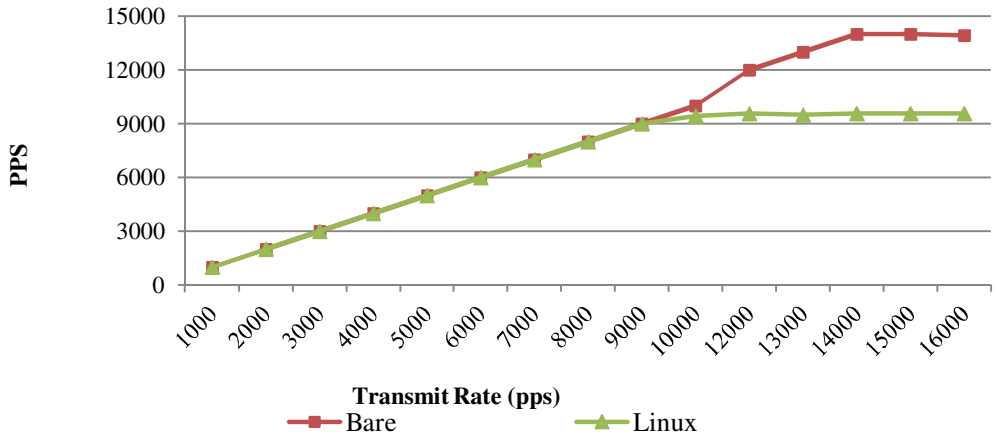

Figure 10. Packets processed per second (TCP)

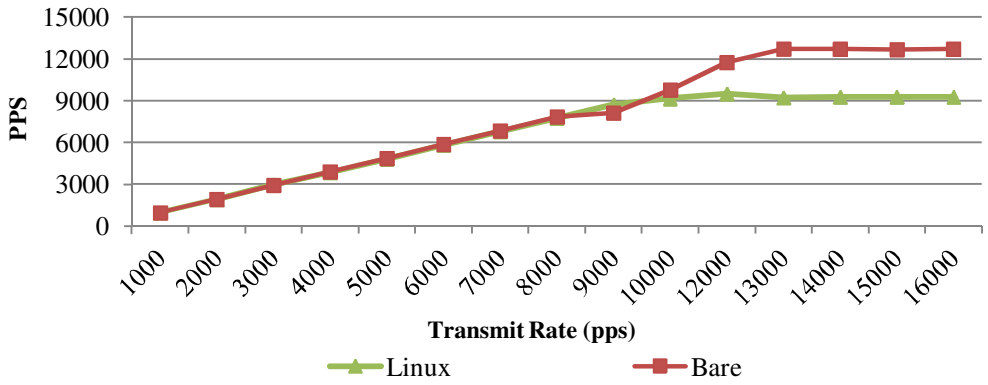

Figure 11. Packets processed per second (traffic mix)

3.2.3 Internal Timings

Internal timings for key operations on the bare PC NAT are shown in Table II. The table also shows that the total CPU utilization measured at a maximum of 14,000 pps is 0.46%. This implies that the unused CPU cycles could be used by other processes (for example, to improve NAT security). Packet forwarding is seen to be the most expensive operation. This is because (unlike the *Lookup* and *AddEntry* functions, which only process headers), the forwarding function has to take into account the payload as well. The *AddEntry* function for outgoing packets is more expensive than *Lookup* because adding an entry to the NAT table requires a prior search and computation of a hash value.

TABLE II. Internal timings for the bare PC NAT

Function	**Metric**	
	Processing Time (µs)	CPU Utilization (%)
Add Entry	1.83	0.144
Look Up	0.165	0.12
Forwarding	1.86	0.196

Chapter 4

6TO4 TUNNELING ON A BARE PC

Deployment of Internet Protocol Version 6 (IPv6) in the Internet has been relatively slow since its introduction over a decade ago [39]. There are a variety of business and practical reasons for the low prevalence of IPv6 networks. However, the difficulty of agreeing on a single technology or standard for use during the IPv4-IPv6 transition has made it harder for IPv6 networks to communicate across the existing IPv4 network infrastructure. Several transition mechanisms were originally proposed [40]. Since then, the automatic tunneling technique known as 6to4 [16] (discussed earlier) has become one of the most widely used transition mechanisms. 6to4 can be deployed on end systems as well as on routers/gateways, and most operating systems support 6to4.

Unfortunately, using 6to4 on the Internet has proved to be challenging due to asymmetry in outbound and return addresses with long routing paths, firewalls, and other causes. This has resulted in a relatively high rate of connection failures being reported on many Web sites, and suggestions have been made to disable 6to4 altogether. A recent informational RFC [42] provides advice to ISPs, content providers and implementers regarding the avoidance of 6to4 failures. It is expected therefore that 6to4 will continue to be used during the transition period.

Since 6to4 gateways are deployed at the edge of the network, it is convenient if they handle both IPv4 and IPv6 packets that may be generated by internal networks. Since internal IPv4 traffic almost always uses private (non-routable) addresses, Network Address Translation (NAT) is then needed.

This implies that a 6to4 gateway must have a co-located NAT [17], or else have NAT performed on another device. We refer to the latter approach as 6to4 with a decoupled NAT. While a co-located NAT is convenient to use, it has more overhead than a gateway that only handles 6to4 traffic. We conduct studies to evaluate the performance of a 6to4 gateway on Linux with a co-located NAT. In order to compare the overhead of only the 6to4 and NAT functions, we also implemented a 6to4 gateway with a co-located NAT as a bare PC application that runs with no operating system (OS) or kernel support.

In this chapter, we describe the design and implementation of the bare PC 6to4 gateway with a co-located NAT, and present the results of experiments conducted in a test LAN environment to compare the performance of the Linux and bare PC gateways with a co-located NAT. Our results show that both the Linux and bare PC 6to4 gateways with a co-located NAT perform better than their respective counterparts with a 6to4 gateway and a decoupled NAT. We also determine the extent of network performance improvement for HTTP and VoIP traffic when a bare PC 6to4 gateway with co-located and decoupled NATs are used instead of the Linux systems. We also compare network performance using 6to4 with both co-located and decoupled NATs.

The remainder of this chapter is organized as follows. In Section 4.1, we describe the design and implementation of the bare PC 6to4 gateway with a co-located NAT, and in Section 4.2, we present the results of the performance study.

4.1 Design and implementation

The self-supporting bare PC 6to4 gateway application has its own lean IPv6 and IPv4 stacks with direct interfaces to the hardware.

There is no OS or kernel in the bare machine, and the application itself processes packets arriving from its two network interfaces. Ethernet Objects ETHOBJ1 and ETHOBJ2 were implemented to interface with the two NICs. Fig. 12 shows the software architecture of the bare PC 6to4 application.

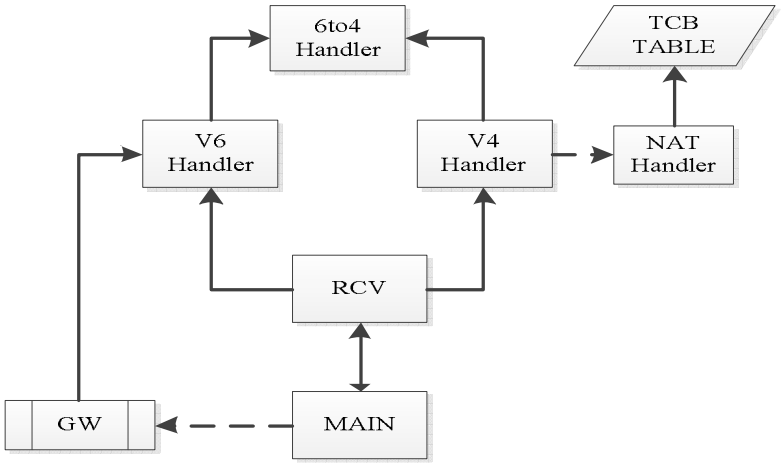

Figure 12. Bare PC 6to4 gateway software architecture

In the bare PC 6to4 gateway, only two tasks are implemented (MAIN and RCV). These two tasks are present in all bare PC applications. Other tasks are added as needed based on the application. The gateway application includes v4 and v6 handlers to process the IPv4 and IPv6 packets respectively. In addition, a 6to4 handler is used for 6to4 processing. The 6to4 gateway also has a NAT handler to process IPv4 traffic entering and exiting the internal network. The NAT mapping table is stored in a data structure called TCBTABLE. The bare gateway application also sends router advertisements to IPv6 clients as done by the radvd daemon [43] used in Linux routers. Router advertisements in the bare gateway are handled independently of the RCV task since they have to be sent even if no packets are received. All bare gateway application code is written in C/C++.

The bare PC 6to4 gateway application is started by invoking an executable boot program stored on a USB flash drive. The initial sector of the USB loads the menus that an administrator can use to select and configure the 6to4 application. Once the 6to4 executable is run, the self-supporting bare PC application object containing the monolithic executable is loaded. Control is then passed to the MAIN task, which starts the IPv6 router advertisements and passes control to the v6 handler. The RCV task is only called when a packet arrives on either interface.

The packet processing logic of the 6to4 gateway is shown in Fig. 13. When a packet arrives, the RCV task checks the packet to determine the packet type (i.e., IPv4 or IPv6). If the packet is an IPv6 packet, it is encapsulated and forwarded. If the packet is an IPv4 packet, the IP header is checked to see if it is a 6to4 packet. If so, the packet is forwarded after decapsulation or dropped, depending respectively on whether the IP checksum is valid or fails. If the packet is not 6to4, the usual IPv4 processing is done prior to forwarding it.

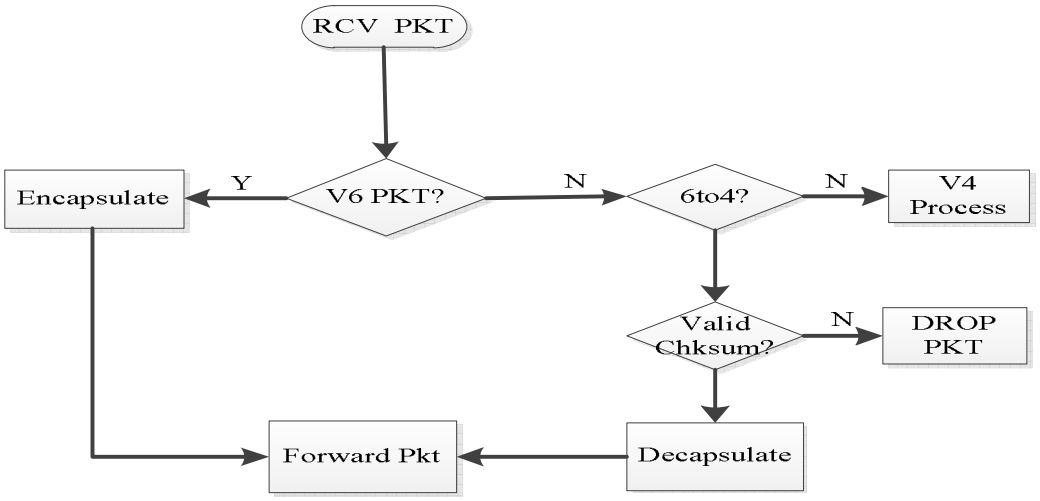

Figure 13. Packet processing logic

4.2 Performance Study

We performed several experiments to measure performance of 6to4/NAT gateways as described below. The generic network setup for the experiments is shown in Fig. 14. Modifications were made as needed to this setup for experiments with co-located or decoupled NATs. The 6to4/NAT gateways run on Linux (Fedora 12) or a bare PC. In the first set of experiments, 6to4 and NAT functionality were configured on the same gateway (co-located NAT). In this case, NAT was configured to process encapsulated IPv6 packets with protocol type 41. In the second set of experiments, NAT and 6to4 functionalities were decoupled to run on separate machines. We conducted four different experiments with decoupled NATs:

- Linux-Linux (both NAT and 6to4 on Linux)
- Linux-bare (NAT on Linux and 6to4 on bare)
- bare-Linux (NAT on bare and 6to4 on Linux)
- bare-bare (both NAT and 6to4 on bare)

The hardware and software used were as follows: the 6to4 and NAT gateways run on Dell OptiPlex GX270 PCs with Intel Pentium IV 2.4 GHz processors, Intel PRO 10/100 and 3Com 10/100 NICs, and 512 MB RAM. The Linux systems used Fedora 12 Linux kernel 2.6.35 with IPtables for NAT and radvd for router advertisements. The client and server systems were Dell Optiplex G520 PCs with 1 GB RAM, Intel PRO/1000 NICs, and 2.4GHz processors.

The switches were 100 Mbps Ethernet Cisco Catalyst 2950 (S1 and S4) and Netgear GS108 (S2 and S3). Data collection was done by capturing packets using Wireshark [35] with mirrored ports on switches S1 and S4.

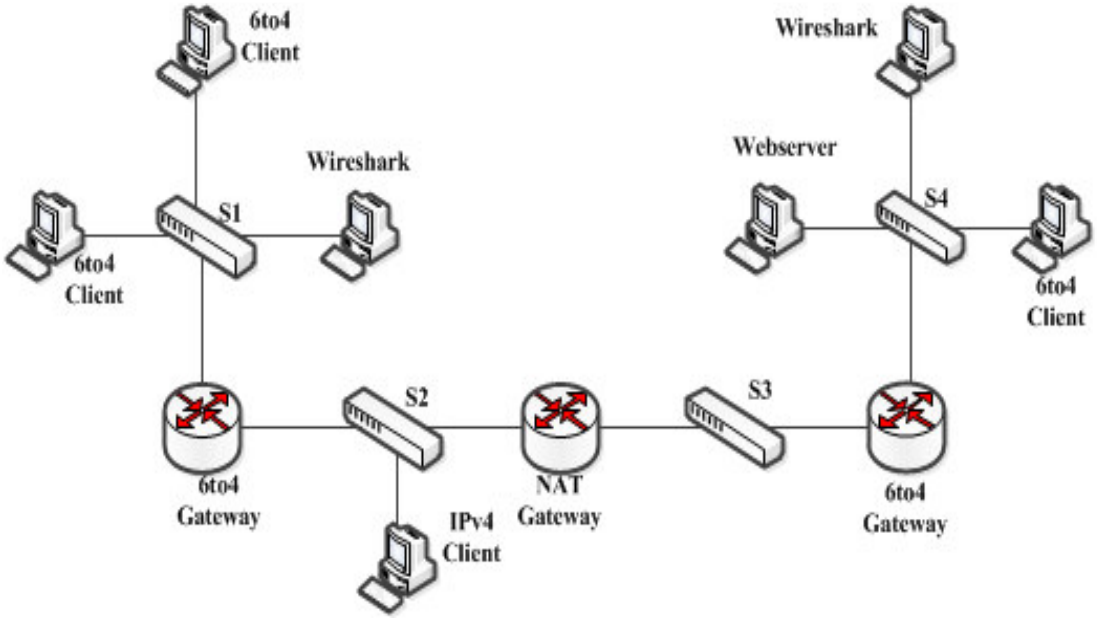

Figure 14. Experimental setup

The experiments were conducted by passing different types of traffic through the gateway in the presence of background traffic generated by using the MGEN traffic generator version 5 [37]. For HTTP traffic, we configured an Apache Web server with IPv6 and IPv4 running on Fedora 12. For VoIP traffic, we used Linphone clients [38]. Each experiment was run 3 times and the measured values were averaged. The values averaged did not differ significantly. The Linux gateway was configured with minimal functionality i.e., all unessential services on it were disabled.

4.2.1 Connection Time

We define the connection time as the time it takes for a TCP connection to be established by measuring the delay between the SYN and ACK from the client.

The connections to the Web server were made in the presence of IPv6 background network traffic at 50 Mbps. Fig. 15 shows that the connection time for a decoupled NAT is higher than that for a co-located NAT.

For Linux, the average connection times are 1.91 ms and 1.72 ms with decoupled and co-located NATs respectively, indicating an improvement of about 11% with a co-located NAT. For the bare PC, the average connections times are 1.38 ms and 0.98 ms with decoupled and co-located NATs respectively. This shows a performance improvement of 40% with a co-located NAT. Comparing the bare PC and Linux gateways, the bare PC performs 76% and 39% better than a Linux gateway with co-located and decoupled NATs respectively. With decoupled NATs, using a bare PC gateway for both devices (bare-bare) is best, and a bare 6to4 gateway and a Linux NAT (Linux-bare) has better performance than the reverse configuration (bare-Linux) as would be expected.

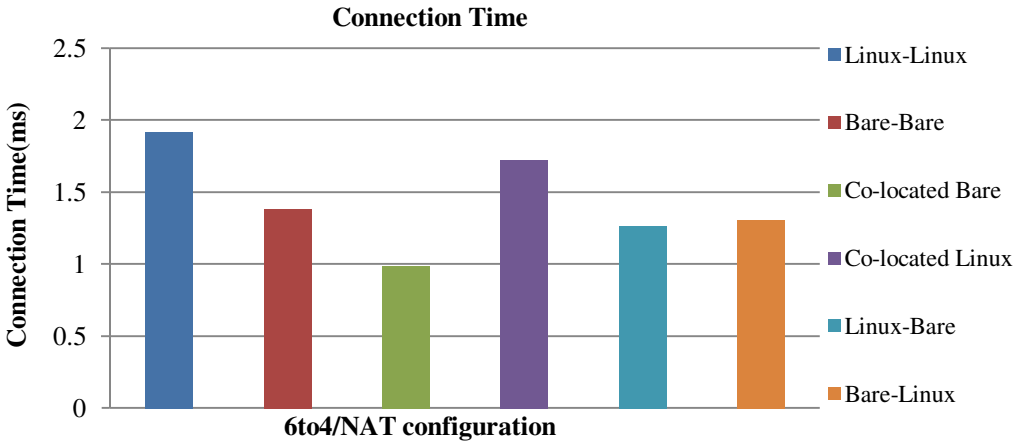

Figure 15. Connection time

4.2.2 Response Time

The response time is the time to download a file for an HTTP GET request i.e., the delay between the GET and OK messages. The HTTP traffic was generated by requesting HTML pages with embedded images of sizes ranging from 4 KB to 150 KB in the presence of 50 Mbps IPv6 background network traffic.

Fig. 16 shows that performance of both the Linux and bare gateways with a co-located NAT is better than with a decoupled NAT. The performance improvements with a co-located NAT versus a decoupled NAT are 7% and 34% for the Linux and bare gateways respectively, and the bare PC performs 51% and 38% better than the Linux gateway with a co-located and decoupled NAT respectively. The figure also shows the increase in IPv6 response time as file size increases; with decoupled NATs, performance for a bare 6to4 gateway and a Linux NAT (Linux-bare) is again better than for the reverse configuration (bare-Linux).

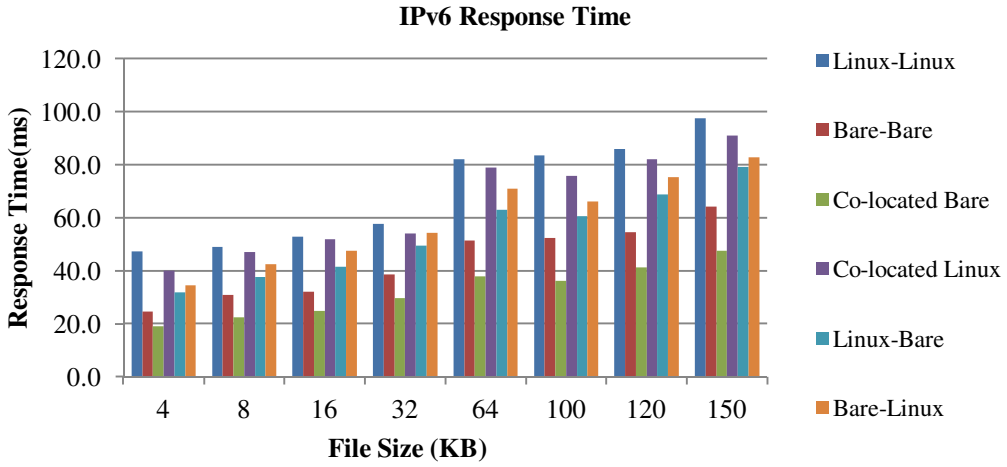

Figure 16. Response time

4.2.3 VoIP Call Setup

VoIP traffic was generated by playing a 3-minute audio file while the background traffic rate was varied from 0-90 Mbps. Fig. 17 shows the call setup times for VoIP at various background traffic rates measured as the delay between the SIP INVITE and the 200 OK messages. Call setup times with a co-located NAT is less than with a decoupled NAT for both the Linux and bare gateways.

The improvements with co-located versus decoupled NATs are 38% and 57% for the Linux and bare gateways respectively, and the bare gateway has an 86% and 65% improvement over the Linux gateway with co-located and decoupled NATs respectively.

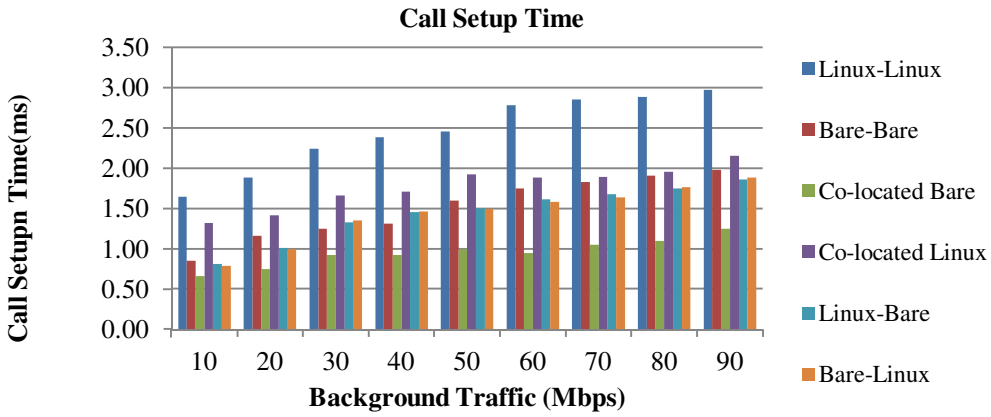

Figure 17. VoIP call setup time

4.2.4 Mean Jitter

Fig. 18 shows the increase in mean jitter for VoIP traffic with increasing background traffic. The performance improvements with a co-located NAT over a decoupled NAT for the Linux and bare gateways are 28% and 33% respectively, and the bare PC performs 82% and 75% better than the Linux gateway with co-located and decoupled NATs respectively.

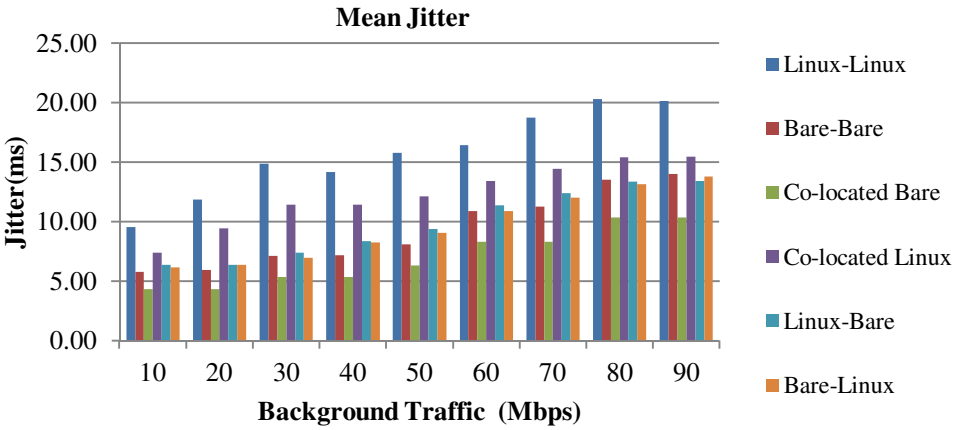

Figure 18. Mean Jitter For VoIP

4.2.5 VoIP Throughput

Fig. 19 shows the decrease in VoIP throughput with increasing background traffic. The performance improvements for the Linux and bare gateways with a co-located versus a decoupled NAT are 12% and 5% respectively, and the bare PC performs 23% and 35% better than the Linux gateway with co-located and decoupled NATs respectively.

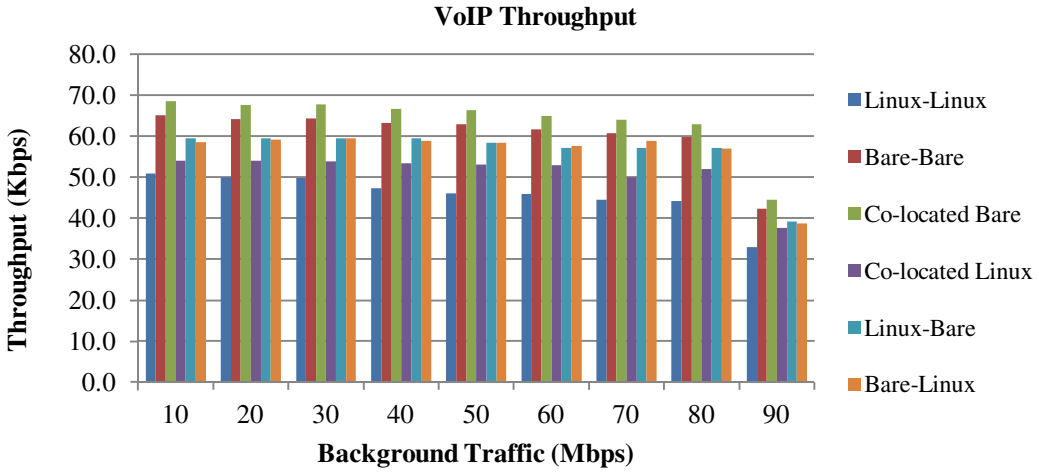

Figure 19. VoIP throughput

4.2.6 Packet Loss

The increase in packet loss as the background traffic rate increases is shown in Fig 20. No packet loss occurs when the background traffic is less than 50 Mbps.

The packet loss ratio (or rate) is less with a co-located than with a decoupled NAT for both the bare and Linux gateways. The decrease in packet loss ratio with a co-located over a decoupled NAT for the Linux and bare gateways is 16% and 13% respectively, and the packet loss ratio decreases by 51% and 52% due to using a bare gateway instead of a Linux gateway with co-located and decoupled NATs respectively.

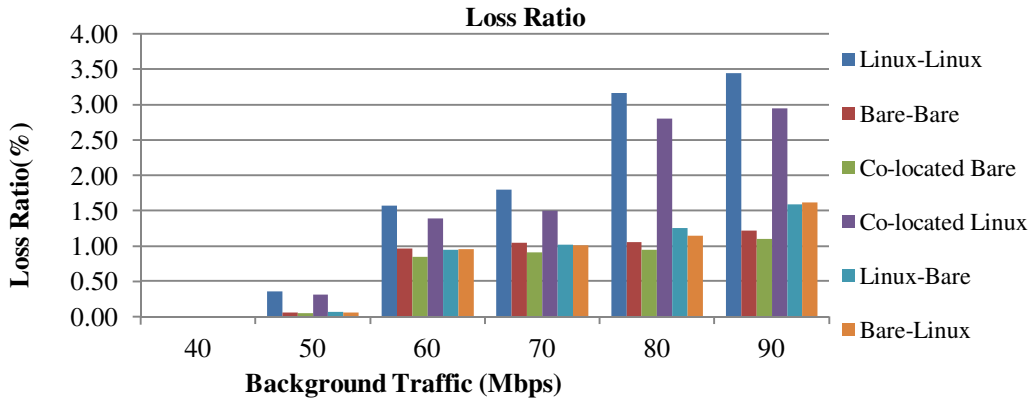

Figure 20. VoIP packet loss rate

4.2.7 Round Trip Time (RTT)

Fig. 21 shows the IPv6 round trip time (RTT) for the network. To measure the RTT, we used the ping6 utility to send 100 packets containing 1400 bytes each with background traffic varying from 10-90 Mbps. The RTT increases with increasing background traffic as expected irrespective of whether co-located or decoupled NATs are used, and a co-located NAT has a smaller RTT than a decoupled NAT for both the Linux and bare gateways. The RTT decreases by 45% and 34% with a co-located over a decoupled NAT for the Linux and bare gateways respectively, and the RTT decreases by 57% and 60% for the bare PC versus the Linux gateway with co-located and decoupled NATs respectively.

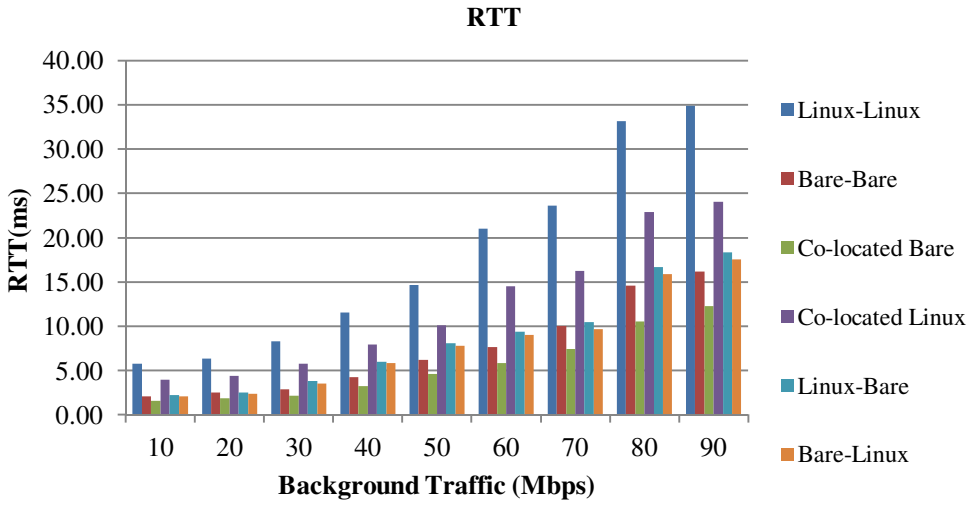

Figure 21. RTT for Ping6

4.2.8 CPU Utilization

To measure the CPU utilization of the co-located gateways, we varied the background traffic rate from 10-200 Mbps. Fig. 22 shows that the peak CPU utilization occurs with 90 Mbps of network traffic and is 1.9% for Linux and 0.5% for the bare gateway. The smaller CPU utilization in the bare gateway is due to using a single thread of execution and streamlined code to process packets.

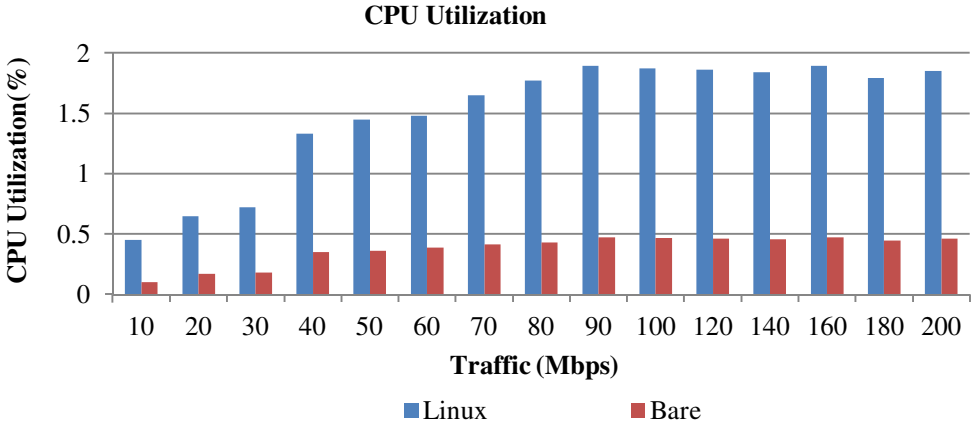

Figure 22. CPU utilization

Chapter 5

BARE PC IVI TRANSLATOR

As we have discussed earlier, IPv6 was introduced to replace IPv4, which is commonly used in the Internet today. IPv6 provides increased address space (16 byte addresses), efficient routing, reduced management requirement, multi-homing, improved security and mobility compared to IPv4 [39]. In spite of these perceived benefits, there is limited use of IPv6 in the Internet at present due to the established and familiar base of IPv4, and interim measures such as NAT and CIDR that have served to conserve IPv4 address space. However, it is expected that IPv6 usage will increase as the number of networks and devices (such as low power sensors) that require IP addresses continues to grow. IPv6 and IPv4 are not compatible due to using different header sizes and formats.

To enable coexistence of IPv4 and IPv6 networks during the transition period, several mechanisms have been proposed [40]. These include dual stack, tunneling, and translation, which we have discussed briefly. Recall that dual stack devices have implementations of both IPv6 and IPv6 protocol stacks running independently, which make it possible for such devices to process both IPv4 and IPv6 packets [41]. We also saw that tunneling is used when communication between isolated IPv6 networks over the current global IPv4 infrastructure is required [42]. Tunneling can also be used to connect IPv4 networks via an IPv6 backbone. Finally, we noted that for IPv6 devices to seamlessly communicate with IPv4 devices during the transition period, translation can be used to convert IPv6 packets to IPv4 packets, and vice versa.

The primary advantage of translation is that it avoids the overhead of the dual-stack approach. The IVI translator [44] is an implementation of a recently proposed IETF initiative for IPv4-IPv6 translation (a brief overview of this initiative was given earlier).

In this chapter, we measure the overhead due to IVI translation by determining internal timings on Fedora Linux using the source code provided for the implementation in [44]. We also implement the IVI translator on a bare PC without an operating system or kernel. This enables us to compare the overhead due to only the IVI translation process with the overhead measured using Linux (Fedora) IVI translator.

The remainder of this chapter is organized as follows. In Section 5.2, we briefly provide an overview of IVI translation. In Sections 5.3 and 5.4, we describe the Linux and bare PC implementations of the IVI translator. In Section 5.5, we present the results of the performance study to determine IVI overhead.

5.1 Overview of IVI translation

The IVI translation employs a prefix-specific and stateless address mapping scheme to enable IPv4 hosts to communicate with IPv6 hosts as described in RFC 6219 [44]. In IVI translation, subsets of an ISP's IPv4 addresses are embedded in the ISP's IPv6 addresses. To enable translation, IPv6 addresses have to be converted to IPv4 addresses and vice versa.

To represent IPv4 addresses in IPv6, the ISP normally inserts its IPv4 address prefix after a unique IPv6 prefix as shown in Fig. 23, which is adapted from [31].

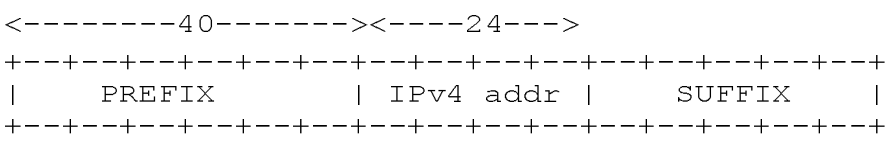

Figure 23. Representing IPv4 ISP addresses in IPv6.

For example, an ISP with an IPv6/32 address block will have a prefix of /40, since bits 32 through 39 are set to all ones, which implies that IPv4 addresses are translated into /64 IPv6 addresses assuming the IPv4 addresses are /24 blocks. The suffixes of these IPv6 addresses are normally set to all zeros. So with this mapping scheme, an ISP with a /32 IPv6 address block 2001:dba:: will translate the IPv4 address 202.38.97.205/24 into the equivalent IPv6 address 2001:dba:ffca:2661:cd00::.

Translation of IPv4 and IPv4 headers is done as prescribed in [28] with the source and destination address translated based on the address mapping scheme as described above. After translating the transport layer (TCP or UDP) and ICMP headers, their checksums are recomputed to reflect the changes in the IP header. The IP header mapping scheme specified in [31] enables an IPv6 header to be constructed from an IPv4 header and vice versa in the following manner (in addition to converting source and destination IPv4 or IPv6 addresses into their IVI mapped equivalent addresses). To translate an IPv4 header into a IPv6 header, the IHL, identification, offset, header checksum, and options are discarded; version (0x4), TOS, protocol, and TTL fields are mapped into version (0x6), traffic class, next header, and hop limit respectively; and the value of total length is decremented by 20 and assigned to payload length.

Conversely, to translate an IPv6 header into an IPv4 header, flow label is discarded; version (0x6), traffic class, next header, and hop limit fields are mapped into version (0x4), TOS, protocol, and TTL respectively; the value of payload length is

incremented by 20 and assigned to total length; IHL is set to 5; and the remaining fields in the IPv4 header are assigned in the usual way (such as generating a value for the identification field and computing the IPv4 checksum). Tables III and IV taken from [31] summarize the key elements of the respective header translations discussed above.

TABLE III. IPv4-to-IPv6 header translation (from [31])

IPv4 Field	**IPv6 Equivalent**
Version (0x4)	Version (0x6)
IHL	Discarded
Type of Service	Traffic Class
Total Length	Payload Length = Total Length – 20
Identification	Discarded
Offset	Discarded
Protocol	Next Header
TTL	Hop Limit
Header Checksum	Discarded
Source Address	IVI mapped address
Destination	IVI mapped address mapping
Options	Discarded

TABLE IV. IPv6-to-IPv4 header translation (from [31])

IPv6 Field	**IPv4 Equivalent**
Version (0x6)	Version (0x4)
Flow Label	discarded
Traffic Class	Type of Service
Payload Length	Total Length = Payload Length + 20
Next Header	Protocol
Hop Limit	TTL
Source Address	IVI mapped address
Destination Address	IVI mapped address mapping
	Header Checksum Recalculated
	IHL=5

5.2 IVI Linux Implementation

The Linux implementation of an IVI translator has two main components: a configuration utility and a translation utility. The configuration utility is used to setup the routes using *mroute* or *mroute6* commands. The translation utility is responsible for translating the packets. To run the IVI translator on Linux, a patch has to be applied to the Linux kernel source. Once the patch is applied, the Linux kernel can be configured for IVI translation after recompiling the kernel. Patching the kernel results in some kernel source files being changed, and 4 other files (mapping.c, proto_trans.c, mroute.c and mroute6.c) being added for configuration and translation.

The functions required for translation referred to in the discussion below are all in the file proto_trans.c. When an IPv4 packet is received, the function *mapping_address4to6* is invoked to map the IPv4 address to an IPv6 address. Based on the value of the protocol field in the received IPv4 header, *icmp4to6_trans*, *tcp4to6_trans* or *udp4to6_trans* is invoked to translate the appropriate protocol.

Each of these methods calls a function that computes the appropriate higher layer protocol (ICMP, TCP, or UDP) checksum as well. Once the protocol translation is done, *iphdr6to4_trans* is called to translate the IP header. For translating IPv6 to IPv4 packets, a similar process is used.

5.3 IVI Bare PC Implementation

The IVI translator application that runs on a bare PC with no OS/kernel support was built by adding IVI translation capability on top of lean bare PC versions of the IPv6 and IPv4 protocols.

The bare PC translator application included the implementation of two Ethernet objects ETH0 and ETH1 that directly communicate with its two network interface cards (NICs) each connecting to one of the two IP networks. Unlike in the Linux system, the IP protocols, address mappings, and protocol mappings in the bare PC are implemented as part of a single object called the XLATEOBJ. The architecture of the bare PC IVI translator is shown in Fig. 24.

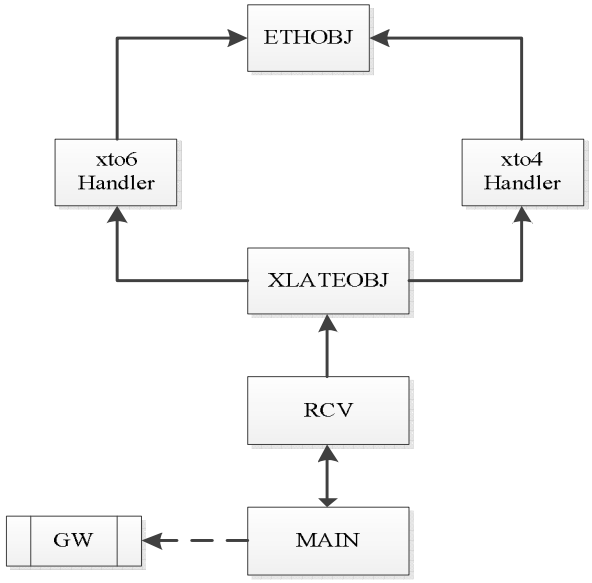

Figure 24. Bare PC IVI architecture

The MAIN and RCV tasks are the only tasks running in the bare PC IVI translator application. The MAIN task runs when the system is started and whenever the RCV task is not running. The RCV task is invoked from the MAIN task whenever a packet (Ethernet frame) arrives on either interface. The capability to send out router advertisements on the IPv6 interface of the bare PC IVI Translator was implemented in the GW object as part of the MAIN task.

As part of XLATEOBJ, *xto4handler* and *xto6handler* methods that translate IPv6 packets to IPv4 packets and IPv4 to IPv6 respectively were implemented.

The *xto4handler* invokes the *addr6to4* method which maps the source and destination IPv6 addresses to the corresponding IPv4 addresses.

The processing logic of the bare PC IVI translator is shown in Fig. 25. The translator initially checks the arriving packet to determine if it is an IPv6 or IPv4 packet. If the packet is an IPv6 packet, the payload protocol type is determined from the next header field, and the appropriate protocol checksum is computed. If the checksum is valid, the IPv6 packet is translated into an IPv4 packet and forwarded. If the packet is an IPv4 packet, the IP checksum is computed. If the checksum is valid, the packet is translated into an IPv6 packet based on the protocol type in the packet and forwarded after validating the higher layer protocol checksum as needed. If any checksum fails, the packet is dropped.

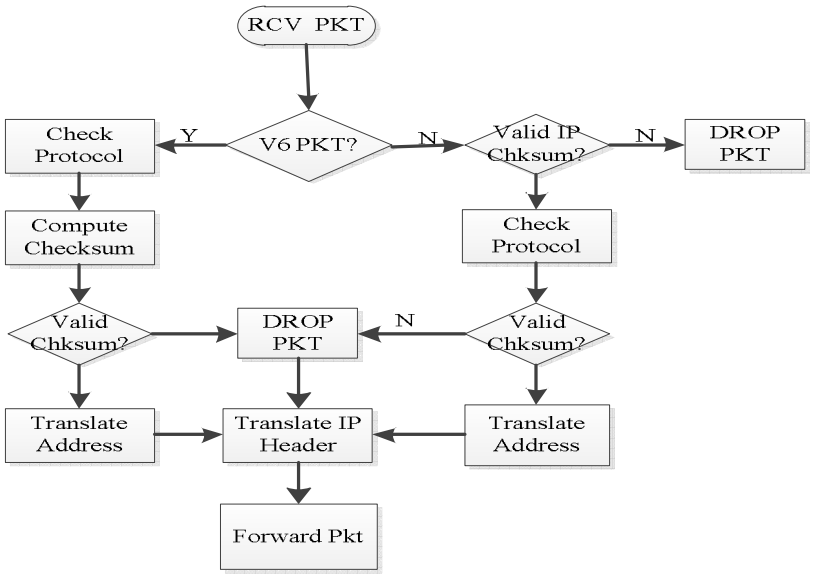

Figure 25. Bare PC IVI packet processing logic

5.4 Internal Timings

5.4.1 Experimental Setup

Fig. 26 shows the test LAN used to generate TCP data. A similar set up was used to generate UDP and ICMP data. In the figure, S1, S2, S3 and S4 are 100 Mbps Ethernet switches, and R1 and R2 are IPv4 and IPv6 routers respectively. Router R1 serves the IPv4 routing domain, which consists of an IPv4 client C4 and an IPv4 server SV4. On the IPv6 side, C6 is an IPv6 client and SV6 is an IPv6 server. C6 and SV6 obtain the necessary routing information from the IPv6 router R2. The IVI translator is represented by XT. The translator is run on a Dell Optiplex GX270 with an Intel Pentium IV 2.4 GHz processor, 512 MB RAM, and Intel PRO 10/100 and 3Com 10/100 NICs. Using the same hardware, the IVI translator was run either on Fedora Linux (Fedora 12, Linux kernel 2.6.31) as the operating system, or as a bare PC application. The client and server systems were run on Dell Optiplex GX520s with a 2.6 GHz processor, 1 GB RAM, and an Intel PRO/1000 NIC, and Fedora Linux as the operating system.

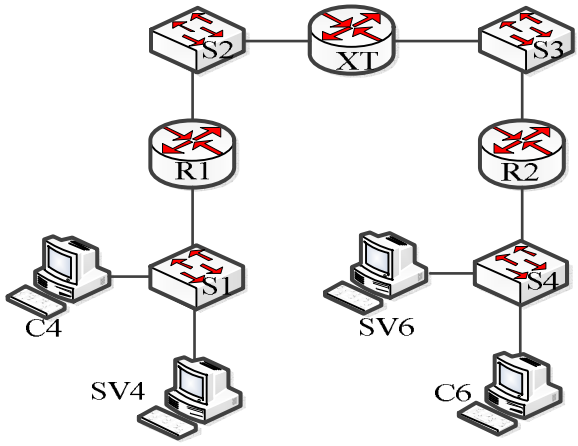

Figure 26. Test LAN for TCP connections

To capture timings during translation with TCP data, connections were made as follows. For IPv4 to IPv6 translation, the IPv4 client (C4) initiated a connection to the IPv6 server (SV6) in the IPv6 network. Router R1 forwarded the IPv4 request to the IVI translator XT, which translated the received packets to IPv6 packets, and sent them to the IPv6 server (SV6) via the IPv6 router R2. For IPv6 to IPv4 translation, the IPv6 client (C6) initiated a connection to the IPv4 server (SV4.) Router R2 forwarded this request to the IVI translator XT. The packet was translated to an IPv6 packet and forwarded to the IPv6 router R1, which delivered the packet to the IPv4 server (SV4).

5.4.2 Results

The internal timing data presented in this section was captured by inserting timing points in the Linux (Fedora) and the bare PC source code. The TCP traffic was generated by sending HTTP requests from clients to servers based on the network setup in Fig. 24. Using a similar setup, Mgen [36] was used to generate UDP traffic and the ping/ping6 utility was used to generate ICMP packets. Each experiment was run three times and the average data is reported (we omit the deviation since the differences in the values for each experiment were small).

Fig. 27 shows the time it takes internally to translate packets using the Linux implementation of the IVI translator. It is seen that the IP header translation is the most expensive operation for both IPv6-IPv4 and IPv4-IPv6 translation, while UDP, TCP and ICMP translation have almost the same translation time. The header translation time is larger since it includes both address and protocol (TCP, UDP or ICMP) translation.

It is also evident that the IP address translation time is larger than the UDP, TCP, and ICMP header translation time since the higher layer translations involve very little processing. Comparing translations between the IP versions in each direction, the IP address translation time is 3.9µs and 0.5µs higher than TCP, UDP, and ICMP translation time for the IPv4 to IPv6 and IPv6 to IPv4 translations respectively. The larger time for the IPv4 to IPv6 address translation compared to the reverse is because the IPv6 address needs to be constructed by extending the IPv4 address, whereas the IPv4 address is simply extracted from the IPv6 address.

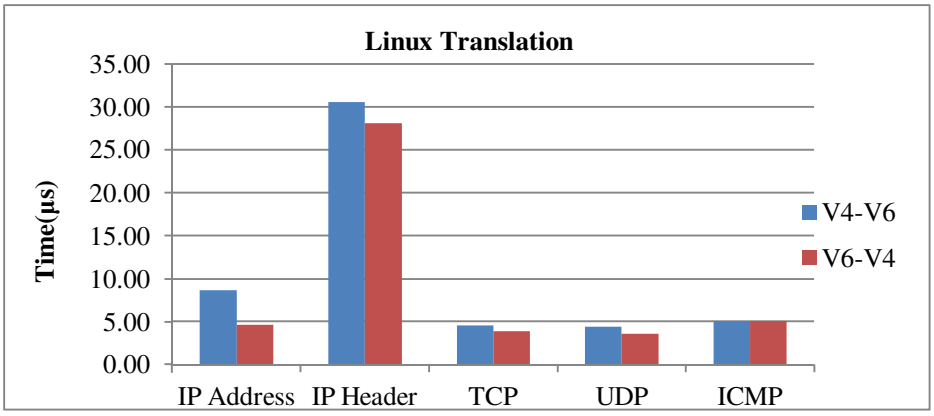

Figure 27. Linux translation overhead

Fig. 28 shows the corresponding IVI translation times on a bare PC. The times for IP header translation from IPv4 to IPv6 is 15.8µs compared to 12µs for translation from IPv6 to IPv4. TCP and ICMP translation take almost the same time: about 2.5µs for IPv4 to IPv6 translation and 2.0µs for IPv6 to IPv4 translation. For IP address mapping, the processing time is 5µs from IPv4 to IPv6, and 2µs from IPv6 to IPv4. For UDP, translating from IPv4 to IPv6 and from IPv6 to IPv4 takes 2µs and 1.5µs respectively.

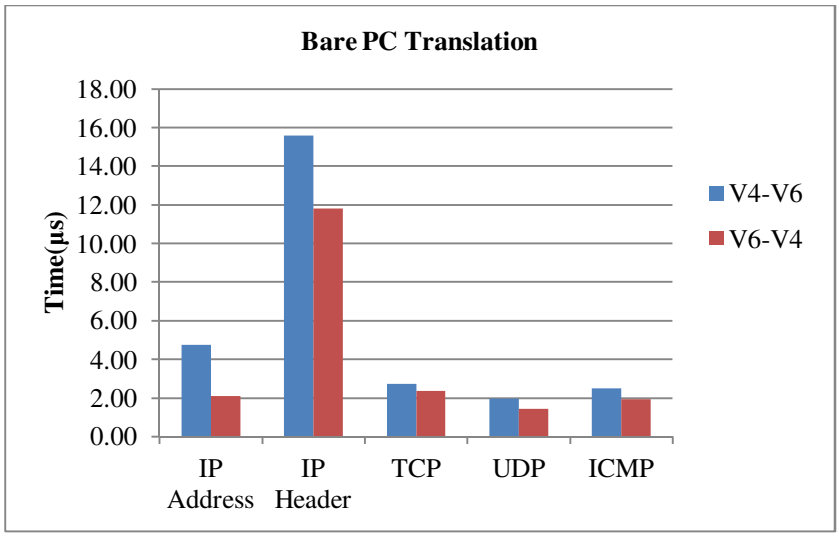

Figure 28. Bare PC translation overhead

Fig. 29 compares the times for IVI translation from IPv4 to IPv6 for the Linux and bare PC implementations. For TCP, UDP and ICMP translations, the time for the bare PC is about 2.3µs less than for the Linux implementation. For IP address translation, the processing time for the bare PC is about 3.9µs less than for the Linux implementation. For IP header translation, the processing time for the bare PC is 15µs less than for the Linux implementation.

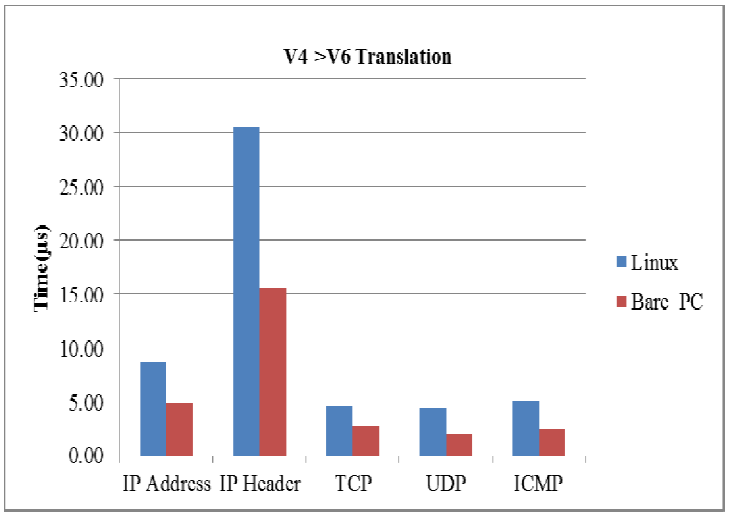

Figure 29. Comparison of IPv4-IPv6 translation overhead

Fig. 30 compares the times for IVI translation from IPv6 to IPv4 translation for the Linux and bare PC implementations. For TCP and UDP traffic, a 2.2µs improvement is seen due to using a bare PC. For ICMP, the improvement in processing time due to using a bare PC is 3.2µs, and for IP address mapping it is 2.6µs for bare PC. These results show that the overhead for IVI translation for both the Linux and bare PC implementations is small. However, for networks in which a very large number of packets need to be translated, the processing gain due to eliminating operating system overhead may be advantageous.

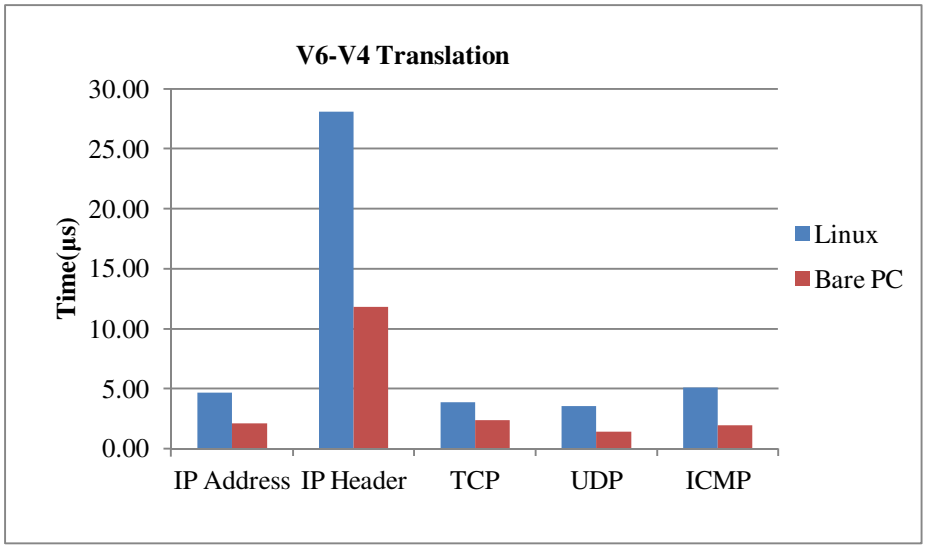

Figure 30. Comparison of IPv6-IPv4 translation overhead

Chapter 6

CONCLUSION

We have designed and implemented NAT, 6to4 tunneling, and IVI translation functionalities on a bare PC platform. We also compared the performance of our implementation with conventional Linux systems to determine the extent to which OS overhead impacts performance.

The bare PC NAT has less overhead than the Linux NAT and performs better with respect to throughput, inbound and outbound processing, and packets processed per second regardless of packet size and payload application type. There is a 34% improvement in the maximum number of packets per second (pps) over Linux under heavy traffic. Internal timings showed that while packet forwarding is the most expensive operation on the bare PC NAT, there is sufficient capacity left for implementing additional functionality such as security-related processing and routing. Bare PC NAT applications could be used in the next-generation Internet to enhance performance as well as security.

In the case of the 6to4 gateways, we evaluated their performance with co-located and decoupled NATs using the Linux and bare PC implementations. In particular, we evaluated response and connection times for HTTP traffic, and call setup time, jitter, and throughput for VoIP traffic in a test LAN. We also compared packet loss, RTT, and CPU utilization. In general, our results show that performance of a 6to4 gateway is better with a co-located NAT than with a decoupled NAT. For a Linux gateway, there is a performance

improvement of between 7%-45%, and for a bare PC gateway, there is a performance improvement of between 13%-57%. Moreover, using a bare PC instead of Linux gateway improves performance by 23%-86% with co-located and decoupled NATs. Reduction of operating system overhead can improve performance on a 6to4 gateway with either a co-located or a decoupled NAT.

Finally, we determined the overhead due to IVI translation by measuring the internal timings for both IPv6 to IPv4 and IPv4 to IPv6 translation using the Linux and bare PC implementations. The results show that in general it is more expensive to translate packets from IPv4 to IPv6 than from IPv6 to IPv4 with a 4.1 µs difference on Linux and a 3.8µs difference on a bare PC. It was also found that address mapping is the most expensive operation in IVI translation regardless of the system on which the translator is implemented. While IVI translation has little overhead for both the Linux and bare PC implementations regardless of the higher layer protocol carried in the payload, eliminating the operating system overhead may enable very efficient translation in very large networks.

Our research has demonstrated the feasibility of building gateways running on bare platforms by using ordinary PCs with no OS or kernel. We showed that the performance of these devices when used to implement NAT, 6to4, or IVI translation is significantly better than that of existing implementations using Linux. While OS-based gateways provide enhanced functionality, they do so at a cost of reduced performance and greater security risks compared to bare PC gateways.

Given the importance of transition mechanisms such as 6to4 tunneling and IVI translation in the next-generation Internet, gateways running on minimalist systems such as bare platforms could provide an attractive low-cost alternative to conventional gateways that trade-off better performance and security for reduced features.

APPENDIX

TABLE V. ICMP(RTT) and throughput values

	ICMP RTT(ms)			Throughput (Mbps)		
Packet Size(Bytes)	**Bare**	**Linux**	**NO-NAT**	**Bare**	**Linux**	**NO-NAT**
40	0.238	0.514	0.208	34.6	31.49	35.8
64	0.248	0.532	0.214	51.4	46.77	53.7
128	0.27	0.566	0.227	60.9	55.42	74.2
256	0.322	0.619	0.252	62	56.42	81.6
512	0.43	0.705	0.297	63.3	57.60	82.2
768	0.538	0.818	0.349	62.1	56.51	82.4
1024	0.636	0.913	0.394	63.4	57.69	84.2
1280	0.75	0.975	0.442	67.2	61.15	84.8
1460	0.861	1.053	0.474	70.1	63.79	85.9
1518	0.848	1.114	0.524	70	63.70	86.0
1600	0.856	1.13	0.53	70.1	63.79	86.1

TABLE VI. Connection and response times data values

	Bare		Linux	
File Size	**Connection Time(ms)**	**Response Time(ms)**	**Connection Time(ms)**	**Response Time(ms)**
8K	0.21	0.496	0.222	0.518
16k	0.216	0.499	0.255	0.519
32K	0.214	0.5	0.262	0.522
64K	0.218	0.502	0.295	0.522

TABLE VII. Packets processed per second

Transmit Rate(pps)	Packets Processed per second(PPS)		Packets Processed per second(PPS)- Traffic Mix	
	Bare	Linux	Bare	Linux
1000	998	997	978	967
2000	1997	1998	1950	1938
3000	2999	2995	2942	2990
4000	3997	3999	3925	3882
5000	4993	4993	4889	4839
6000	5997	5993	5879	5825
7000	6998	6996	6866	6786
8000	7992	7993	7845	7753
9000	8994	8995	8140	8722
10000	9997	9444	9793	9165
12000	11991	9565	11750	9522
13000	12999	9500	12735	9215
14000	13991	9560	12711	9273
15000	13992	9563	12700	9277
16000	13940	9559	12710	9280

TABLE VIII. IPv6 connection time

File Size(KB)	Linux-Linux	Bare-Bare	Co-located Bare	Co-located Linux	Linux-Bare	Bare-Linux
4	1.92	1.46	1.05	1.80	1.24	1.27
8	1.90	1.39	0.99	1.70	1.27	1.29
16	1.87	1.31	0.94	1.60	1.21	1.22
32	1.78	1.23	0.88	1.70	1.14	1.20
64	1.86	1.31	0.94	1.60	1.21	1.29
100	1.91	1.39	0.99	1.60	1.37	1.42
120	2.11	1.54	1.10	1.90	1.40	1.41
150	1.96	1.39	0.99	1.89	1.27	1.30

TABLE IX. IPv6 response time

File Size(KB)	Linux-Linux	Bare-Bare	Co-located Bare	Co-located Linux	Linux-Bare	Bare-Linux
4	47.3	24.5	19.2	40.0	32.0	34.5
8	49.1	31.0	22.6	47.0	37.6	42.5
16	52.9	32.2	25.0	52.0	41.6	47.5
32	57.8	38.6	29.8	54.0	49.6	54.3
64	82.1	51.4	37.9	78.9	63.1	71.1
100	83.5	52.3	36.3	75.7	60.6	66.2
120	85.8	54.5	41.3	82.0	68.8	75.4
150	97.6	64.2	47.5	91.0	79.2	82.9

TABLE X. IPv6 jitter

BG Traffic (Mbps)	Linux-Linux	Bare-Bare	Co-located Bare	Co-located Linux	Linux-Bare	Bare-Linux
10	9.53	5.81	4.33	7.41	6.37	6.15
20	11.87	5.95	4.35	9.44	6.40	6.37
30	14.85	7.15	5.34	11.42	7.38	6.98
40	14.15	7.20	5.34	11.42	8.38	8.25
50	15.77	8.10	6.32	12.13	9.36	9.09
60	16.43	10.89	8.33	13.41	11.37	10.88
70	18.75	11.25	8.34	14.42	12.38	11.99
80	20.26	13.54	10.33	15.41	13.37	13.17
90	20.10	14.00	10.37	15.46	13.41	13.78

TABLE XI. IPv6 call setup time

	CALL SETUP TIME(ms)					
BG Traffic (Mbps)	**Linux-Linux**	**Bare-Bare**	**Co-located Bare**	**Co-located Linux**	**Linux-Bare**	**Bare-Linux**
10	1.65	0.86	0.66	1.32	0.81	0.79
20	1.88	1.16	0.75	1.41	1.01	0.99
30	2.24	1.25	0.92	1.66	1.33	1.35
40	2.38	1.31	0.92	1.71	1.45	1.46
50	2.46	1.60	0.99	1.92	1.50	1.49
60	2.78	1.75	0.95	1.88	1.61	1.58
70	2.85	1.83	1.05	1.89	1.68	1.64
80	2.88	1.90	1.10	1.95	1.75	1.76
90	2.97	1.98	1.25	2.15	1.86	1.88

TABLE XII. IPv6 roundtrip time

	RTT (Ms)					
BG Traffic (Mbps)	**Linux-Linux**	**Bare-Bare**	**Co-located Bare**	**Co-located Linux**	**Linux-Bare**	**Bare-Linux**
10	5.76	2.10	1.60	3.97	2.23	2.12
20	6.33	2.50	1.85	4.37	2.52	2.37
30	8.33	2.91	2.13	5.74	3.81	3.55
40	11.52	4.29	3.25	7.95	5.96	5.83
50	14.65	6.24	4.62	10.11	8.08	7.78
60	20.99	7.67	5.83	14.48	9.36	8.99
70	23.58	10.01	7.42	16.26	10.45	9.68
80	33.15	14.56	10.56	22.86	16.65	15.89
90	34.92	16.15	12.26	24.08	18.31	17.55

TABLE XIII. VoIP throughput

	THROUGHPUT(Kbps)					
BG Traffic (Mbps)	**Linux-Linux**	**Bare-Bare**	**Co-located Bare**	**Co-located Linux**	**Linux-Bare**	**Bare-Linux**
10	51.0	65.2	68.6	54.1	59.5	58.5
20	50.0	64.2	67.6	54.1	59.5	59.1
30	49.8	64.3	67.7	53.9	59.5	59.5
40	47.3	63.3	66.6	53.4	59.5	58.9
50	46.1	63.0	66.3	53.0	58.3	58.4
60	46.0	61.8	65.0	52.9	57.2	57.7
70	44.5	60.8	64.0	50.0	57.2	58.9
80	44.2	59.9	63.0	52.0	57.2	57.0
90	33.0	42.3	44.6	37.7	39.2	38.8

TABLE XIV. VoIP loss ratio

	LOSS RATIO					
BG Traffic (Mbps)	**Linux-Linux**	**Bare-Bare**	**Co-located Bare**	**Co-located Linux**	**Linux-Bare**	**Bare-Linux**
10	0.00	0.00	0	0	0.00	0
20	0.00	0.00	0	0	0.00	0
30	0.00	0.00	0	0	0.00	0
40	0.00	0.00	0	0	0.00	0
50	0.36	0.06	0.055	0.31	0.07	0.06
60	1.57	0.97	0.85	1.39	0.95	0.96
70	1.80	1.05	0.91	1.5	1.02	1.01
80	3.16	1.05	0.95	2.8	1.25	1.15
90	3.44	1.22	1.1	2.95	1.59	1.62

TABLE XV. CPU utilization

	CPU% Utilization	
Load(Mbps)	**Linux**	**Bare**
10	0.45	0.10
20	0.65	0.17
30	0.72	0.18
40	1.33	0.35
50	1.45	0.36
60	1.48	0.39
70	1.65	0.41
80	1.77	0.43
90	1.89	0.4725
100	1.87	0.4675
120	1.86	0.465
140	1.84	0.46
160	1.89	0.4725
180	1.79	0.4475
200	1.85	0.4625

TABLE XVI. IVI translation overhead

	Translation times (µs)			
	Linux		Bare	
Metric	**V4-> V6**	**V6->V4**	**V4-> V6**	**V6->V4**
IP Address	8.65	4.66	4.76	2.10
IP Header	32.55	28.10	15.58	11.80
TCP	4.55	3.89	2.73	2.38
UDP	4.40	3.56	1.98	1.43
ICMP	5.12	5.10	2.51	1.94

BIBLIOGRAPHY

[1] S. Sen, R. Guerin and K. Hosanagar, "Functionality-rich versus minimalist platforms", Computer Communication Review, 41(5), pp. 36-43, 2011.

[2] R.K, Karne, A.L, Wijesinha, and G. Ford, "Opinion-- Stay on course with an evolution or choose a revolution in computing", ACM SIGARCH Computer Architecture News, Volume 36, Number 4, September 2008, pp1-6.

[3] R.K. Karne, R. Gattu, R. Dandu, and Z . Zhang, "Application-oriented Object Architecture: concepts and approach", 26th Annual International Computer Software and Applications Conference, IASTED International Conference, Tsukuba, Japan, NPDPA 2002, October 2002.

[4] R.K. Karne, K. Venkatasamy, T. Ahmed, "Dispersed Operating System Computing (DOSC)", Onward Track OOPSLA 2005, San Diego, CA, October 2005.

[5] L. He, R. K. Karne, and A. L. Wijesinha, "Design and performance of a bare PC Web server", International Journal of Computer Applications, pp. 100-112, Volume 15, No. 2, June 2008.

[6] P. Srisuresh and K. Egevang, "Traditional IP network address translator (Traditional NAT)", RFC 3022, Jan 2001.

[7] T. Hain, "Architectural implications of NAT", RFC 2933, Nov 2000.

[8] S. Guha and P. Francis, "Characterization and measurement of TCP Traversal through NATs and firewalls", Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement (IMC '05).

[9] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session traversal utilities for NAT (STUN)", RFC 5389, October 2008.

[10] L. Zheng, "Host-initiated NAT", IETF Draft, March 2001.

[11] R. Murakami, N. Yamai, and K. Okayama, "A MAC-address relaying NAT router for PC identification from outside of a LAN", 10th Annual International Symposium on Applications and the Internet, pp. 237-240, 2010.

[12] T-C. Huang, S. Zeadally, N. Chilamkurti and C-K. Shieh, "A programmable network address translator: design, implementation, and performance", ACM Transactions on Internet Technology, Vol. 10, No. 1, Article 3, February 2010.

[13] R. Bless and M. Rohricht, "Implementation and evaluation of a NAT-Gateway for the general Internet signaling transport protocol", IEEE 12th International Conference on High Performance Computing and Communications, September 2010.

[14] Z. Xiangming and W. Zheng, "A NAT traversal mechanism for peer-to-peer networks", International Symposium on Intelligent Ubiquitous Computing and Education, 2009.

[15] R. Fink and R. Hinden, "6bone (IPv6 testing address allocation)", IETF RFC 3701, March 2004.

[16] B. Carpenter and K. Moore, "Connection of IPv6 domains via IPv4 clouds", RFC 3056, February 2001.

[17] C. Huitema, "Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs)", RFC 4380, February 2006.

[18] D. Thaler, "Teredo extension", RFC 6081, January 2011.

[19] M. B. Viagenie and F. Parent, "IPv6 tunnel broker with the Tunnel Setup Protocol (TSP)", RFC 5572, February 2010.

[20] H. J. Liu and P. S. Liu, "Hierarchical routing architecture for integrating IPv4 and IPv6 networks", 2008 IEEE Asia-Pacific Services Computing Conference.

[21] S. Narayan and S. Tauch, "IPv4-v6 transition mechanisms network performance evaluation on operating systems", 2nd International Conference on Signal Processing Systems (ICSPS), July 2010.

[22] C. Friaças, M. Baptista, M. Domingues and P. Ferreira, "6to4 versus tunnel brokers", International Multi-Conference on Computing in the Global Information Technology (ICCGI'06).

[23] R. Yasinovskyy, A. L. Wijesinha, R. K. Karne and G. Khaksari, "A comparison of VoIP performance on IPv6 and IPv4 networks", ACS/IEEE International Conference on Computer Systems and Applications", pp. 603-609 , May 2009.

[24] R. Yasinovskyy, A. L. Wijesinha, and R Karne, “Impact of IPsec and 6to4 on VoIP quality over IPv6”, 10th International Conference on Telecommunications (ConTEL), pp 235-242, June 2009.

[25] G. Tsirtsis and P. Srisuresh, “Network Address Translation-Protocol Translation (NAT-PT)”, RFC 2766, February 2000.

[26] J. Hagino and K. Yamamoto, “An IPv6-to-IPv4 transport relay translator”, RFC 3142 , June 2001.

[27] M. Bagnulo, P. Matthews, and I. van Beijnum, “Stateful NAT64: Network address and protocol translation from IPv6 Clients to IPv4 servers”, RFC6146, April 2011.

[28] X. Li, C. Bao, and F. Baker, “IP/ICMP translation algorithm”, IETF RFC 6145, Mar. 2011.

[29] C. Bao, C. Huitema, M. Bagnulo, M. Boucadair, and X. Li, “ IPv6 addressing of IPv4/IPv6 translators”, RFC 6052, Oct. 2010.

[30] M. Bagnulo, A. Sullivan, P. Mathews, and I. van Beijnum, “DNS64: DNS extensions for network address translation from IPv6 clients to IPv4 servers”, RFC 6147, April 2011.

[31] Y. Zhai, C. Bao, and X. Li, “Transition from IPv4 to IPv6: a translation approach”, Sixth IEEE International Conference on Networking, Architecture and Storage, 2011.

[32] M. Boucadair et al., ”Anticipate IPv4 address exhaustion a critical challenge for Internet survival”, 1st International Conference on Evolving Internet , pp. 27-32, 2009.

[33] LanTrafficV2 , http://www.zti-telecom.com, accessed 09/20/2011.

[34] S. Bradner and J. McQuaid, “Benchmarking methodology for network interconnect devices”, RFC 2544, Jan 2001.

[35] Wireshark packet analyzer, http://www.wireshark.org/,accessed 12/28/2011.

[36] http_ load, http://acme.com, accessed 09/20/2011.

[37] Mgen, http://cs.itd.nrl.navy.mil, accessed 05/10/2012.

[38] Linphone, http://www.linphone.org/, accessed 12/28/2011.

[39] S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification”, RFC 2460, December 1998.

[40] R. Gilligan and E. Nordmark , “Transition mechanisms for IPv6 hosts and routers”, RFC 2893, August 2000.

[41] D. Punithavathani and K. Sankaranarayanan, “IPv4/IPv6 transition mechanisms”, European Journal of Scientific Research ISSN 1450-216X Vol.34 No.1 (2009), pp.110-124.

[42] B. Carpenter, “Advisory Guidelines for 6to4 Deployment”, RFC 6343, August 2011.

[43] Linux IPv6 router advertisement daemon (radvd), http://www.litech.org/radvd/ accessed 12/30/2011.

[44] X. Li et al., “The China Education and Research Network (CERNET) IVI translation design and deployment for the IPv4/IPv6 coexistence and transition”, RFC 6219, May 2011.

CURRICULUM VITA

NAME: Anthony Kofi Tsetse

PERMANENT ADDRESS: Box GP 17855 Accra, Ghana

PROGRAM OF STUDY: Information Technology

DEGREE AND DATE TO BE CONFERRED: Doctor of Science, 2012

SECONDARY EDUCATION: Sunyani Secondary School, Ghana

Universities Attended

1. Towson University , Jan 2009- Aug 2012., DSc Information Technology
2. IT University of Copenhagen – Denmark, Aug 2002- Sept 2005, MSc Information Technology
3. University of Applied Sciences, Offenburg-Germany, Aug 2001- Sept 2003, MSc Communication and Media Engineering
4. Kwame Nkrumah University of Science and Technology , Sept 1997 – July 2001, BSc(hons) Computer Science

Professional Publications:

1. **A. K. Tsetse**, A. L. Wijesinha, R. K. Karne and A. Loukili , " A Bare PC NAT Box", International Conference on Communications and Information Technology (ICCIT 2012)

2. **A. K. Tsetse**, A. L. Wijesinha, R. K. Karne and A. Loukili , "A 6to4 Gateway with Co-located NAT", IEEE International Conference on Electro Information Technology (EIT 2012)

3. **A. K. Tsetse,** A. L. Wijesinha, R. K. Karne and A. Loukili , "Measuring the IPv4-IPV6 IVI Translation Overhead", ACM Research in Applied Computation Symposium(RACS 2012)

4. A. Loukili, A. L. Wijesinha, R. K. Karne and **A. K. Tsetse**, "Web Server Performance with Cubic and Compound TCP", IASTED International Conference on Communication, Internet, and Information Technology (CIIT2012)

5. A. Loukili, A. L. Wijesinha, R. K. Karne and **A. K. Tsetse**, "TCP's Retransmission Timer and the Minimum RTO", 6th International Workshop on Performance Modeling and Evaluation of Computer and Telecommunication(PMECT 2012)