

Network Protocol Performance Evaluation of IPv6 for Windows NT

A Thesis

Presented to the Faculty of
California Polytechnic State University
San Luis Obispo

In partial Fulfillment
Of the Requirements for the Degree
Master of Science in Computer Science

By

Peter Ping Xie

June 1999

AUTHORIZATION FOR REPRODUCTION
OF MASTER'S THESIS

I grant permission for the reproduction of this project report in its entirety or any of its parts,
without further authorization from me.

Signature (Peter Ping Xie)

Date

APPROVAL PAGE

Title: Network Protocol Performance Evaluation of IPv6 for
Windows NT

Author: Peter Ping Xie

Date submitted: June 1999

Dr. Mei-Ling Liu

Advisor

Signature

Dr. James G. Harris

Committee Member

Signature

Dr. Patrick O. Wheatley

Committee Member

Signature

Abstract

Today more and more applications are developed based on network communications. There are a number of factors which affect the performance of networking applications:

- (a) characteristics of the processor such as speed of CPU, size of memory and disk cache,
- (b) bandwidth of physical network connection, (c) processing overhead of the application, and
- (d) processing overhead of the network protocol stack. The last item listed above is the focus of this study.

Microsoft Research implemented a prototype of IPv6 protocol stack, as known as MSR IPv6, for the Windows NT platform. This research evaluates the performance of MSR IPv6 protocol stack by measuring and analyzing its network latency and throughput, as well as identifying its processing overheads and potential bottlenecks. It provides a source code-based instrumentation model on Windows NT platform for protocol performance evaluation. It also presents the measurement results which are obtained from a testbed consists of two Pentium machines with 100Mbps fast Ethernet. Some of the instrument techniques, such as timestamping the programming execution with the Pentium CPU tick counter, and using the NT kernel debugging mechanism for outputting timestamps from a kernel-mode protocol driver, are also discussed in this paper.

Keywords

Network protocol, Performance evaluation, IPv6, Ipng, Windows NT, TCP/IP, Protocol stack, Latency, Throughput, Instrumentation.

Acknowledgements

It is hoped that this paper will provide useful and readable information on the performance evaluation area with regards to Windows NT and the TCP/IP protocol stack.

The author would like to thank Dr. Mei-Ling Liu for her direction and advisorship in this research project. The time she put forth and her patience were very much appreciated. Special thanks goes to the thesis committee members Dr. James Harris and Dr. Patrick Wheatley for their time and input.

Thanks goes to Dr. Chris Scheiman, who provided valuable input on the measurement methodology, Jim Fischer, who is always willing to discuss and share, and Siu-Ming Lo, who spent countless nights with the author working together in the lab for the same goal.

Thanks goes to the whole 3COM research team of Cal Poly for the good teamwork. Thanks extend to all the faculty and staff members of Computer Science department for providing excellent services to the graduate study program. Many thanks also go to 3COM Incorporation for supporting the research project.

The author is especially grateful to his parents, Yun Xie and Lirong Zhang, for their never-ending support and love, without which he could never have achieved his study and career goals.

Last but not least, the author wants to thank his wife, Kelly Youlan Huang, who has sacrificed so much to support his study, and his 3-year-old son, Jack Junjie Xie, who has proven that life is most enjoyable when looking at it through simplistic, innocent, and trusting eyes.

Table of Contents

	Page
List of Tables.....	ix
List of Figures.....	x
Chapter 1 Introduction.....	1
1.1 Background of the 3Com-Calpoly joint research project.....	1
1.2 Overview of this study.....	2
1.3 Outline of this thesis.....	3
Chapter 2 Basics of network performance evaluation.....	4
2.1 What is protocol latency?.....	4
2.2 What is protocol throughput?.....	6
2.3 What is CPU utilization?.....	7
2.4 Categorizing processing overheads of TCP/IP.....	7
Chapter 3 MSR IPv6, the IPv6 prototype for Windows NT.....	8
3.1 TCP/IP general.....	8
3.2 Introduction to IPv6 protocol, IPv6 vs IPv4.....	9
3.3 Our test subject: MSR IPv6 implementation.....	11
3.3.1 Supported functionalities.....	11
3.3.2 Data structure.....	12
3.4 Current works on MSR IPv6.....	15
Chapter 4 The instrumentation model on Windows NT platform.....	17
4.1 The test bed.....	17
4.1.1 The hardware of test bed: host & target.....	17

4.1.2	Setting up host-target communication with null modem cable.....	18
4.2	Software Environment.....	20
4.2.1	General implementation of a protocol stack in Windows NT.....	20
4.2.2	The MSR IPv6 source code, implementation hierarchy.....	21
4.2.3	SDK WinDbg utility.....	23
4.2.4	Enable target machine's kernel debugging in NT.....	24
4.2.5	Compiling the IPv6 stack with NT DDK.....	25
4.2.6	Checked build versus free build.....	26
4.2.7	Creating an installation kit for a new build of IPv6 stack.....	27
4.2.8	Install or upgrade the new build of IPv6 stack in NT.....	28
4.3	Instrument technology	29
4.3.1	Instrumentation code, general consideration.....	30
4.3.2	Obtaining timestamps.....	30
4.3.3	Buffering timestamps.....	32
4.3.4	Outputting timestamps.....	33
4.3.5	Measurement code overhead.....	34
4.3.6	Identifying functions called by different kernel threads.....	36
4.3.7	Test utilities and applications.....	37
4.3.8	Identifying the common path through the stack	38
4.3.9	Time-stamping locations.....	41
	Chapter Summary.....	42
 Chapter 5 Instrumentation results analysis.....		43
5.1	Latency for the whole stack IPv6.....	43
5.1.1	Latency for TCP/IPv6 of varying packet size	43
5.1.2	Latency for UDP/IPv6 of varying packet size.....	46
5.1.3	Comparison between TCP and UDP over IPv6.....	47
5.2	Latency for the interested parts	50

5.2.1 Check-summing for TCP/IPv6 data.....	50
5.2.2 Check-summing for UDP/IPv6 data.....	53
5.2.3 Buffer allocation and de-allocation in TCP/IPv6.....	54
5.2.4 Buffer allocation and de-allocation in UDP/IPv6.....	56
5.2.5 Data moving operation in UDP/IPv6.....	57
5.3 Payload throughput for the whole stack IPv6.....	59
5.3.1 Payload throughput for TCP/IPv6 versus packet size.....	59
5.3.2 Payload throughput for UDP/IPv6 versus packet size.....	61
5.4 CPU utilization.....	63
5.4.1 CPU utilization for TCP over IPv6.....	64
5.4.2 CPU utilization for UDP over IPv6.....	66
5.4.3 Summary of CPU utilization for TCP and UDP.....	58
Chapter Summary.....	71
Chapter 6 Conclusion and future study.....	72
6.1 Conclusion.....	72
6.2 Future works.....	73
References.....	75
Appendices.....	

List of Tables

Page

Table 3a	Supported functionality and included test applications in MSR IPv6 1.1 and 1.2.....	12
Table 3b	Summary of abbreviations and annotations in the MSR IPv6 data structure diagram [2].....	14
Table 4a	Summary of instrumentation location numbers.....	41
Table 5a:	Summary of trend line formulas of TCP and UDP start and stop latency through the whole stack.....	48
Table-5b	Formulas for buffer management latency trend lines in TCP/IPv6.....	55
Table 5c	Functions with CPU utilization greater than 1% in TCP/IPv6.....	68
Table 5d	Functions with CPU utilization greater than 1% in UDP/IPv6.....	69
Table 5e	Chapter 5 summary.....	71

List of Figures	Page
Figure 2.1 Comparing protocol latency model with water pipe model	5
Figure 2.2 Start latency and stop latency illustration.....	6
Figure 3.1 TCP/IP Hierarchy and OSI Reference Model.....	8
Figure 3.2 MSR IPv6 data structure[2].....	13
Figure 4.1 Host-target instrumentation model on Windows NT.....	18
Figure 4.2 Windows NT Networking Protocol Architecture.....	20
Figure 4.3 MSR IPv6 release 1.1 Source Code Components.....	21
Figure 4.4 MSR IPv6 protocol components' interfaces in the NT networking architecture.....	23
Figure 4.5 Subtracting measurement overhead from measured latency.....	36
Figure 4.6 Common call path through MSR TCP/IPv6 stack on the sender's side (downward).....	39
Figure 4.7 Common call path through MSR UDP/IPv6 stack on the sender's side (downward).....	40
Figure 5.1 TCP/IPv6 start latency and stop latency through the whole stack, payload buffer size from 500 bytes to 10,000 bytes in step of 500 bytes.....	44
Figure 5.2 TCP/IPv6 start latency and stop latency through the whole stack; payload buffer size from 100 bytes to 20,000 bytes in step of 100 bytes.....	45
Figure 5.3 UDP/IPv6 start latency and stop latency through the whole stack, payload buffer size from 500 bytes to 10,000 bytes in step of 500 bytes.....	46
Figure 5.4 UDP/IPv6 start latency and stop latency through the whole stack; payload buffer size from 100 bytes up to 20,000 bytes in step of 100 bytes.....	47
Figure 5.5 Comparison of TCP/UDP for start/stop latency through the whole stack.....	48
Figure 5.6 Message diagrams for both TCP and UDP of sending payload size of 20,000 bytes.....	49
Figure 5.7 MSR TCP/IPv6 Checksuming latency versus payload size from 1k to 20k step 1k bytes.....	51
Figure 5.8 The payload/total checksum latency and the stop latency in TCP/IPv6.....	52
Figure 5.9 Percentage of the payload/total checksum latency to the stop latency in TCP/IPv6.....	52
Figure 5.10 Checksum latency of MSR UDP/IPv6 versus payload size from 0.1K to 20K step 1K bytes.....	53
Figure 5.11 Checksum time to stop latency in UDP/IPv6.....	53
Figure 5.12 UDP: Percent of checksum latency to stop latency.....	53

Figure 5.13 MSR TCP/IPv6 Buffer Management overheads versus payload size.....	54
Figure 5.14 TCP: Buffer management time to stop latency.....	55
Figure 5.15 TCP: Percent of buffer management time to stop latency.....	55
Figure 5.16 MSR UDP/IPv6 buffer management latency versus payload size from 1k to 20k step 1k.....	56
Figure 5.17 UDP: Buffer management to stop latency.....	56
Figure 5.18 UDP: Percent of buffer management to stop latency.....	56
Figure 5.19 UDP/IPv6 data move latency versus payload size.....	57
Figure 5.20 UDP data move latency to stop latency.....	58
Figure 5.21 UDP/IPv6:percent of data move latency to stop latency.....	58
Figure 5.22 TCP/IPv6 payload throughput versus payload buffer size, from 500 to 10K step 500 bytes.....	59
Figure 5.23 TCP/IPv6 payload throughput versus payload buffer size, from 100 to 20K step 100 bytes.....	60
Figure 5.24 UDP/IPv6 payload throughput versus packet size from 500 to 10,000 bytes step 500 bytes.....	61
Figure 5.25 MSR UDP/IPv6 payload throughput versus payload size from 100 to 20k step 100 bytes.....	62
Figure 5.26 Illustrate the CPU utilization measurement.....	63
Figure 5.27 TCP/IPv6 kernel modules CPU utilization.....	64
Figure 5.28 CPU utilization of individual functions within tcpip6.sys, TCP over IPv6.....	65
Figure 5.29 UDP/IPv6 kernel modules CPU utilization.....	66
Figure 5.30 CPU utilization of individual functions within tcpip6.sys, UDP over IPv6.....	67
Figure 5.31 TCP over IPv6, CPU utilization by stack overhead category.....	69
Figure 5.32 UDP over IPv6, CPU utilization by stack overhead category.....	70

Chapter 1 Introduction

1.1 Background of the 3Com-Calpoly joint research project

Nowadays network computing has become more and more dominant in the computer applications on the personal computer platforms. The performance of the networking applications depends on a number of factors: (a) physical characteristics of the processor (CPU speed and memory size, disk cache size, etc.), (b) bandwidth of the network connection, (c) efficiency of the application program, and (d) efficiency of the network protocol stack that is used for communication by the application. The last item listed above is the focus of this study. The performance of the protocol stack, together with the behavior of the operating system, greatly affects the efficiency of network applications built on top of it. Investigation of network protocol performance, as well as the evaluation methodology, is a key step of optimizing the performance of the protocol.

Since 1996, 3COM Inc. and California Polytechnic State University have conducted a joint research project to investigate and maximize the performance of network protocol on PC platforms. The protocols that the measurements and development were based on currently include both version 4 and version 6 of TCP/IP for Windows NT 4.0, and TCP/IP version 4 for Red Hat Linux 5.1. As of this writing, three Cal Poly faculty members, Prof. James Harris, Prof. Mei-Ling Liu, and Prof. Chris Scheiman, together with other students in the Electrical Engineering, Computer Science and Computer Engineering major, are actively working on different parts of this research project. 3COM Inc. has supported the lab facilities in room 20-114 on the campus of Cal Poly for the project includes providing NT stations, Ethernet network adapter cards and their drivers, network hubs, and some state-of-the-art development and measurement tools.

One of the goals of this project is to establish a model of the TCP/IP protocol stacks on various platforms. To establish such a model, efforts were undertaken to quantify various aspects of the performance of the stack. This includes measuring latency and throughput of TCP/IP stack (both version 4 and version 6) for NT, version 4 for Linux, and those of the application layer. This thesis is a project report of the performance measurement and evaluation on TCP/IP stack version 6 for Windows NT platform.

1.2 Overview of this study

Protocol performance evaluations can be classified as comparative evaluations and analytical evaluations[1]. This study was mostly an analytical evaluation research. It proposed a source-code based instrumentation model of measuring the latency and throughput of the whole TCP/IP protocol stack for Windows NT operating system, or measuring those within different parts (layers or components) of the stack. The process of research consists of (a) investigating the MSR IPv6 implementation source code, (b) identifying the common execution path through the stack, (c) designing, implementing and embedding measurement code into the stack at probing points, (d) compiling and installing the instrumented version of stack into the Windows NT operating system, (e) setting up the host-target debugging model for instrumentation, (f) running test applications on the target machine for generating timestamp data into the buffer, (g) triggering dumping of timestamps out from the buffer within the stack, (h) charting and analyzing the results, (i) repeating the whole process above for different measurements or for refining the instrumentation model.

1.3 Outline of this thesis

Chapter-1 provides an introduction to the background of the 3COM/Cal Poly joint research project, an overview of this study regarding its objective, method, model, and results. Chapter-2 gives the general information on the area of network systems and protocols performance evaluation. It introduces the concepts of latency, throughput, and CPU utilization, as well as previous studies and current works in the research topic of protocol performance evaluation. Chapter-3 introduces our test object, the prototype implementation of IPv6 protocol stack for Windows NT, developed by Microsoft Research. It talks about the general TCP/IP architecture, answers questions like, why do we need a new IP protocol? What are the major differences between IPv4 and IPv6, and so on. It also summarizes the implementation of data structure and features of functionality that has been implemented as a subset of IETF IPv6 specification. Chapter-4 discusses the details of our instrumentation model for measuring latency and throughput within IPv6. It covers details of the test bed hardware, the software environment including tools to build and debug the instrumented stack, and the instrumentation approach. We discussed two key technologies used in our model: time-stamping the interested point of the stack with CPU tick counter, and outputting timestamps from kernel mode using the NT debugger WinDbg. In Chapter-4 we also highlighted the common execution path through the stack on which lie most of our probing points. Chapter 5 presents the latency, throughput, and CPU utilization measurement result charts,

as well as some analysis of the stack behavior that the charts are representing. Chapter-6 concludes the measurement and evaluation result, and leaves some suggestions to the future researchers on this topic.

Chapter 2 Basics of network performance evaluation

Network performance evaluation can be defined as a process that sets up an effective measurement model on the measured computer network system, collects measurement data, compares or analyzes the result, and provides a conclusion on the degree that the system meets the expectation of designers, implementers, and users. Performance is a qualitative characteristic; it is highly subjective to the needs of the people who are involved with the system [1]. The major work of performance evaluation is to quantify the qualitative characteristics and accurately describe the performance of the evaluated object.

A computer network consists of several primitives: computer nodes (or hosts), network connection media, network structure (topology), networking protocols, and networking applications. A protocol is the “language” that the networked computers used to talk to each other. It’s a set of formal rules describing how to transmit and receive data across the network. From the implementation point of view, a protocol is a program that is implemented according to its network function and interface specifications, it runs on the operating system, as any other programs, to carry out its protocol functionalities. Performance evaluation of a network protocol usually involves the measuring and analysis of the following criteria: (a) latency, (b) throughput, and (c) CPU utilization. Below are more details regarding these 3 criteria.

2.1 What is protocol latency?

Generally, *latency* is the time it takes for a packet to be transmitted across a network connection, from sender to receiver, or the period of time that a frame is held by a network component before it is forwarded [19]. Specifically, protocol latency is the period of time that a packet or datagram is held by a network protocol or its sub-layer for processing before it is forwarded to other protocols or network components. This is the processing overhead that a protocol has added into the efficiency of a network system, especially in the system with a multi-layer protocol hierarchy.

The protocol latency can be categorized into two kinds: start latency and stop latency. *Start latency* is defined as the interval of time from the first bit of payload data reaches the top of the protocol stack, until the first bit of payload data exits from the bottom of the stack. On the other hand, *stop latency* is the interval of time from the first bit of payload data reaches the top of the protocol stack, until the very last bit of payload data

exits from the bottom of the stack. We can compare the protocol latency with the amount of time it takes for some amount of water pass through a water pipe. When payload data stream is processed and passed through a protocol stack, it resembles to water flowing through a piece of water pipe. The start latency is the time it takes for the first drop of water to get into the pipe, until it gets out from the other end of the pipe. Stop latency is the time it takes for the first drop of water gets into the pipe, until there is no more water flows out from the pipe. Obviously start latency is a part of stop latency. Stop latency is needed to calculate average throughput, which can be analogue to the average amount of water flows through the pipe per second.

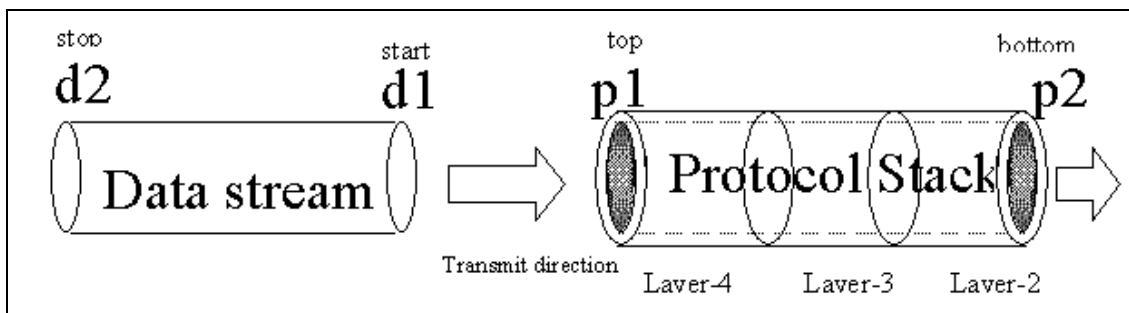


Figure 2.1 Comparing protocol latency model with water pipe model

$$\text{Start Latency} = \Delta t \text{ ((time point for } d1 \text{ reaches } p1), \text{ (time point for } d1 \text{ reaches } p2))$$

$$\text{Stop Latency} = \Delta t \text{ ((time point for } d1 \text{ reaches } p1), \text{ (time point for } d2 \text{ reaches } p2))$$

$$\text{Average Throughput} = (\text{amount of data in the stream}) / (\text{Stop Latency})$$

Start latency can be used to determine the efficiency of the protocol implementation for initializing and preparing for data transmission. On the other hand, stop latency identifies the efficiency of protocol implementation as well as the network connection. Because stop latency includes start latency and the round trip time from packet sending until acknowledgement received. If some of the packets are lost for some reasons, and have not been acknowledged by receiving side, these packets must be resent which results in delay in stop latency. Because of this, stop latency is also affected by efficiency and availability of the network connection beside the protocol performance. The following diagram illustrates the two types of latency.

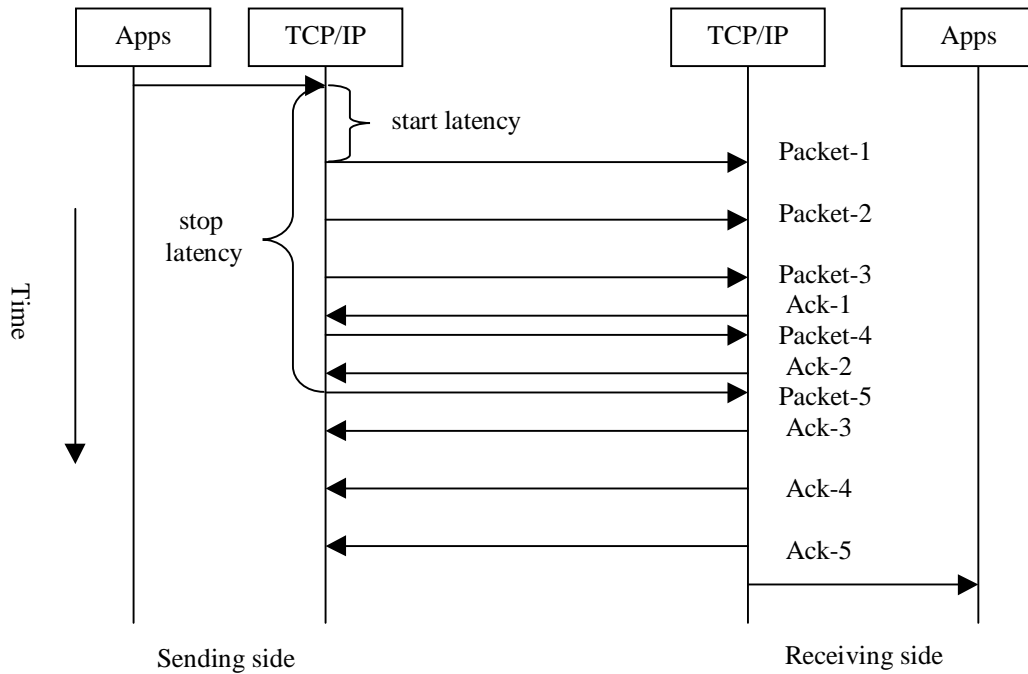


Figure 2.2 Start latency and stop latency illustration

2.2 What is protocol throughput?

Within the area of computer network, *throughput* is defined as the amount of data that a communication channel (or a network component) can carry per time unit. Protocol throughput is the amount of data that a protocol stack can process per time unit (KB/s or Mbps). It differs from different criteria such as size of input data and protocol service that involved (e.g. TCP/UDP). The average throughput of a protocol implementation denotes its data processing capacity for a given network hardware environment. Payload throughput is the part of throughput that devotes for sending payload data, as oppose to that for transmitting gross information including encapsulation (TCP/UDP headers, IP headers, and checksum) and handshakes overhead (e.g. TCP acknowledgements). We calculated the payload throughput using the below formula:

$$\text{Payload throughput} = (\text{size of payload}) / (\text{stop latency})$$

Where size of payload is the amount of data sent by the test application programs such as TTCP6. This is one of the command-line specified parameters when running the tests. Stop latency refers to the measured stop latency from the instrumentation results for that test run.

2.3 What is CPU utilization?

CPU utilization usually refers to the percentage of CPU time it takes when a certain program is running. It denotes the percent of time that a CPU is busy for a process [1]. In a pre-emptive multiple-process operating system, the higher CPU utilization a process has the higher workload it has brought to the system thus the higher efficiency it has. Many tools such as NT performance monitor and VTune can measure a process's CPU utilization. Another study as a part of the 3COM project is to investigate the networking latency, throughput, and CPU utilization behavior on the application layer using Windows NT built-in performance counters [18]. We also measured CPU utilization for the stack module and the functions within the stack. (See chapter 5.4)

2.4 Categorizing processing overheads of TCP/IP

Protocol processing overhead of TCP/IP can be categorized into the following types [5]:

- (a) checksumming overhead performed on TCP/UDP data field and the IP header field (no IP header checksumming in IPv6 protocol);
- (b) Data moving such as user mode to kernel mode copying, kernel to device copying, and cache maintenance;
- (c) data structure operations that manipulate network data structure such as socket buffer, defragmentation queue, and device queue;
- (d) error checking operations such as parameter checking on socket system calls;
- (e) "mbuf" operations that manage the "mbuf" data structure in BSD Unix-based network subsystems;
- (f) operating system overhead includes support for sockets, synchronization overhead and other general operating system support functions;
- (g) protocol-specific operations such as setting header fields and maintain protocol states;
- (h) other overheads which are too small to measure, such as symmetric multiprocessing locking mechanism.

Among of these overhead types, (a), (b), (c), and (g) have their counterparts in the MSR IPv6 protocol stack for Windows NT. We measured and analyzed some of the above protocol overheads in our instrumentation model.

Chapter 3 MSR IPv6, the IPv6 prototype for Windows NT

3.1 TCP/IP general

TCP/IP (Transmission Control Protocol / Internet Protocol) is one of the most commonly used protocol stack for local area networks, wide area networks (e.g. the Internet) and lots of other kind of networks. TCP/IP is the set of protocols providing functions of the Internet and Transport Layer. When mapping the TCP/IP hierarchy to ISO's OSI 7-layer reference model, TCP/IP's Internet layer is corresponding to the Network layer (OSI layer-3) and Transport Layer (TCP) is corresponding to Transport (OSI layer-4) and Session Layer (OSI layer-5)[16]. When data are sent from an application of a host to an application running in another host, they are processed and encapsulated through each layer of the stack from the top to the bottom at the source host, and processed and de-capsulated in the opposite way at the destination hosts. The stack is designed into layers in order to provide independent functionality and clear interface specification for structured implementation.

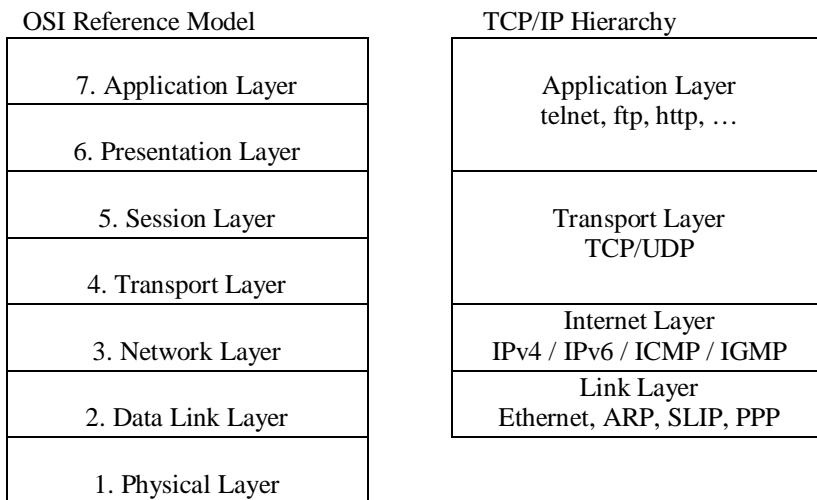


Figure 3.1 TCP/IP Hierarchy and OSI Reference Model

The functions of TCP/IP layers are as below [16]:

- Application: The application layer protocols includes two categories: (a) independent user applications which generate network service requests. Examples include Telnet (a

remote terminal emulation utility) and FTP (File Transfer Protocol, a client/server file transferring utility). (b) upper layer protocols which provide service to TCP/IP application programs that build on top of these protocols. Examples includes SMTP(Simple Message Transfer Protocol, servicing the e-mail client program), DNS (Domain Name System, servicing the nslookup utility program), and SNMP (Simple Network Management Protocol, servicing the network management applications).

- **Transport:** The protocols in this layer provide end-to-end communication services. They are TCP, Transport Control Protocol, servicing the connection-oriented communications, and UDP, User Datagram Protocol, servicing connection-less communications.
- **Internet:** The IP (Internet Protocol) is a connection-less protocol. Both IPv4 and IPv6 belong to the Internet layer. It mainly provides services like packaging, addressing, and routing data to its destination. Also operating on this layer are ICMP (Internet Control Message Protocol), which is controlling and regulating the IP traffic, and IGMP (Internet Group Management Protocol), which provides multicast services.
- **Link:** The link layer protocols are used to facilitate the transmission of IP data over the existing network medium. ARP (Address Resolution Protocol) is an example of the protocols on this layer, which provides an interface between the IP layer and the physical network with a mechanism to translate and cache the IP address into Ethernet address and vice versa.

3.2 Introduction to IPv6 protocol, IPv6 versus IPv4

The traditional TCP/IP protocol stack was designed in the early 1970's when the network was relatively small, slow, and unpopular. The number of hosts connected to the network is limited, and the hosts are also relatively primitive with limited memory, processing capacity and networking throughput. After almost 30 years, the situation has been

greatly changed. The Internet has become much more popular with number of hosts connected is greatly increased. Number of networks in the Internet is doubling every 12 months [17]. More and more versatile and powerful applications have been developed to run on the Internet with TCP/IP protocol suites. But all these changes only happened in the upper layers, which is the user and application layer, while the lower layers, the networking protocol stack itself, remains unchanged.

Because of the shortage of IP address space and other limitations of TCP/IP, the current TCP/IP (known as IPv4) protocol stack will be replaced by its most promising candidate, IPv6 (known as IPng, Internet Protocol of next generation) in the near future. Many big computing companies, including Microsoft, Sun, CISCO, 3Com, etc, have implemented their version of IPv6 protocol stack over their products or platforms according to the IETF's (Internet Engineering Task Force) draft standards [11]. But none of the IPv6 stacks has been delivered to the market for commercial use since the protocol itself has not been finalized. Most people envision that even after IPv6 has come out to reality, IPv4 and IPv6 will co-exist for a rather long period, perhaps 10 to 15 years, to ensure seamless migration. The ease of transition from IPv4 to IPv6 is one of the key points when IPv6 was designed [17].

There are several major benefits achieved by migrating from IPv4 to IPv6. First, the scaling problem of the Internet address space will be solved. IPv6 provides 128 bits (IPv4 use 32 bits) for both source and destination IP address. Even when the efficiency of address assignment policy of IPv6 is the same as IPv4, it could provide more than 1,500 addresses for each square meter of the whole surface of the Earth[17]. With IPv6, greatly expanded routing and addressing capabilities are supported. Some new features like auto-configuration of addresses, Anycast addressing (as oppose to multicast and unicast addressing), are also added. Second, simplification and improvement have been made to IP header and header option for header processing performance promotion. Header processing is in the hot path of packet sending and receiving within the stack, which should be highly optimized. Major simplifications on IPv4 header include: (a) assign a fixed format to all headers, (b) remove the IP header checksum, and (c) remove the hop-by-hop segmentation procedure[8]. These updates will minimize the costs of header processing in IPv6 even though the header length is increased to 40 bytes (20 bytes in IPv4) due to quadruple of IP address. Besides, improved support for header options in IPv6

includes changes in the way that options are encoded to allow more efficient forwarding, less constraints on length of options, and greater flexibility for introducing new options in the future. Third, quality of service capabilities is provided. This new capability, with defining flow label and priority class in IPv6, enable the labeling packets that belong to particular traffic flow for which the sender has requested special handling, such as a non-default quality of service or a real-time service of a guaranteed bandwidth. Fourth, Security can be brought to the IP layer with encryption and authentication. Two security specifications are added to IPv6: authentication header and encrypted security payload [8]. The first one provides an authentication procedure by which the recipient of a packet can guarantee the source IP address is authentic and the packet has not been altered during transmission. The later one guarantees that only legal receiver(s) will be able to read the content of the packet[8]. Encryption algorithm MD5 (computes a 128-bit hash code for arbitrary message length) and DES-CBC (uses 56-bit key) were chosen because they are well-known and widely available algorithms. Once security is provided in the IP layer, it becomes a standard service that all applications can use in all platforms [8]. Cross-platform security-required applications like voting through the Internet can be deployed with great ease.

3.3 Our test subject: MSR IPv6 implementation

The MSR IPv6 implementation was an experimental prototype of IPv6 developed for Windows NT platform by MSR (Microsoft Research). It consists of TCP (Transport Control Protocol), UDP (User Datagram Protocol) and a version 6 IP (Internet Protocol). The TCP and UDP are directly ported from those for IPv4 except the internal interface towards IPv6. It runs on NT 4.0 and NT 5.0 (build 1773 or later). It was not designed to support Windows 95 or 98 platform. MSR has made their implementation publicly available both in source code and binary form for study and investigation purpose. Since it is not intended to be a product release for commercial use (The Windows Networking Group in Microsoft is working on a product-quality IPv6 implementation), The MSR IPv6 does not yet support all the features specified by the IETF IPv6 protocol draft standards. In spite of this, it was considered a good testing and instrumentation subject as a TCP/IP stack. MSR IPv6 was designed and implemented as a separate stack from IPv4 to allow ease of experimentation with IPv6 without affecting the existing IPv4 functionality on the same machine.

3.3.1 Supported functionalities

	MSR IPv6 release 1.1	MSR IPv6 release 1.2
Supports[2]	<ol style="list-style-type: none"> 1. Basic IPv6 header processing 2. Hop-by-hop option header 3. Destination option header 4. Fragmentation header 5. Neighbor discovery 6. Stateless address auto-configuration 7. TCP and UDP over IPv6 8. ICMPv6 9. Multicast listener discovery (IGMPv6) 10. IPv6 over IPv4 (Carpenter/Jung draft) 11. Raw packet transmission 12. Automatic and configured tunnels 	<p>All the supported features in release 1.1 plus:</p> <ol style="list-style-type: none"> 1. Routing header 2. Correspondent node mobility 3. Host and router functionality 4. Ethernet and FDDI media
Not yet supports[2]	<ol style="list-style-type: none"> 1. Mobility in IP 2. Authentication 3. Encryption 4. Routing header 	<ol style="list-style-type: none"> 1. Full mobility support 2. Authentication 3. Encryption
Included test applications	<ol style="list-style-type: none"> 1. NcFTP (client program, ported to IPv6) 2. igmp6test (IGMPv6) 3. ping6 (ICMPv6) 4. ipv6 (stack config status dumper) 5. mldtest (socket calls to add/drop multicast groups) 6. Bloodhound network monitor 7. testsend(send different type of raw IPv6 packets) 8. tracert6 (trace route of IPv6) 9. ttcp6(test TCP/UDP connection) 	<p>All the test applications in release 1.1 plus:</p> <ol style="list-style-type: none"> 1. Fnord web server 2. RAT and SDR (popular multicast conferencing multicast application) 3. Ipv6/IPv4 translator 4. hdrtest (IPv6 header test) 5. hmactest (test the HMAC-MD5 algorithm implementation) 6. md5test (test the MD5 algorithm implementation, derived from RSA)

Table 3a Supported functionality and included test applications in MSR IPv6 1.1 and 1.2

The above table briefly summarized the supported and unsupported functions, as well as the provided test applications, in both release 1.1 (released on March 24, 1998 with the joint efforts of USC/ISI East) and release 1.2 (released in February 1999, which is an incremental release based on 1.1 added with improved or new functionalities and test applications). All of our tests are done on release 1.1 since it's a fully functional TCP/IP stack and the core IP functions have not been changed significantly in release 1.2.

3.3.2 Data structure

The design of MSR IPv6 data structure was closely matched to the conceptual data structures found in the Neighbor Discovery specification RFC 1970[2]. For example, it has a Neighbor Cache (NCE), a Destination

Cache (called Route Cache, RCE in MSR IPv6), a Router List (RLE), and a Prefix List (PLE). The design deviates from the conceptual data structures in two major ways. First, it supports multi-homed hosts (hosts with multiple interfaces), and this complicates the data structures slightly. Second, Route Cache entries cache the preferred source address for destinations as well as caching the next-hop neighbor, path MTU, and other information [2].

Below is the data structure diagram of MSR IPv6:

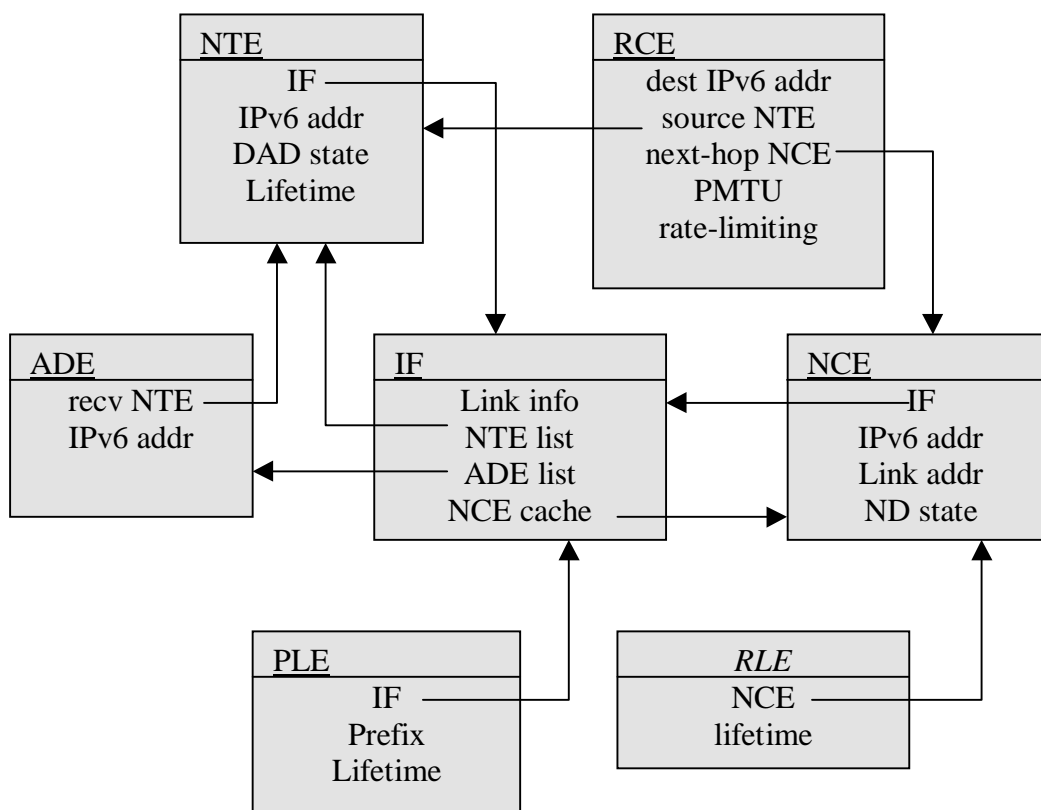


Figure 3.2 MSR IPv6 data structure [2]

No	Terms	Full Name	Annotations and comments
1	NTE	Net Table Entry	NTE represents a unicast address assigned to an interface for use as a source address.
2	DAD	Duplicate Address Detection	In principle, most of the interfaces come out of box with an IEEE 802 address. These addresses should be unique worldwide. Anyhow, DAD is carried out to prevent

			accidental collision of addresses.
3	RCE	Route Cache Entry	RCE caches the results of the next-hop selection algorithm. It maps a destination IPv6 address to a next-hop neighbor (NCE). In addition, the RCE caches the preferred source address (NTE) to use when sending to the destination, the path MTU, and ICMP error rate-limiting information.
4	PMTU	Path Maximum Transfer Unit	The number of bytes in a single IP datagram that can be sent through a physical path. It depends on the link level protocol in use and the media type.
5	ADE	Address Entry	ADE represents a destination address for which the interface can receive packets. A unicast address can have both an ADE and an NTE, but a multicast address will have only an ADE. Each ADE maps to the NTE that logically receives the packets (would be used to reply to the packet if necessary).
6	IF	Interface	The central part of data structure in MSR IPv6 stack that inter-connects and coordinates the other entry tables.
7	NCE	Neighbor Cache Entry	NCE represents a neighboring node on the interface's link. The NCE maps the neighbor's IPv6 address to its link-layer address.
8	ND	Neighbor Discovery	The Neighbor Discovery algorithm manages the state transitions for NCEs. MSR IPv6 uses a simple least-recently-used (LRU) algorithm to manage the cache.
9	PLE	Prefix List Entry	PLE is used to determine if a destination is directly reachable or not.
10	RLE	Router List Entry	RLE represents routing information learned from advertisements sent by routers. Together with PLE it can determine the next hop (NCE) to which a packet should be sent. If the destination address matches the prefix in a PLE, then the destination is assumed to be "on-link," or directly reachable. Otherwise the list of default routers (RLEs) must be consulted to choose a router for that destination.

Table 3b Summary of abbreviations and annotations in the MSR IPv6 data structure diagram [2]

The Interface (IF) is the central part of the data structure [2]. There is one Interface for each network interface card, plus additional Interfaces for logical or virtual links like loop-back, configured or automatic tunneling, and 6-over-4. In addition to link-layer information and configuration information, each Interface has a list of NTE (Net Table Entry), which represent the addresses assigned to the interface that can be used as source addresses; a list of ADEs, which represent the addresses for which the Interface can receive packets; and a cache of NCEs, which represent neighboring nodes on that link [2].

Below is an excerpt of definition code from the "msripv6/tcpip6/ip6/ip6def.h" header file for the IF interface data structure in MSR IPv6 release 1.1:

```
//=====tcpip6/ip6/ip6def.h=====
// Information about IPv6 interfaces. There can be multiple NTEs for
// each interface, but there is exactly one interface per NTE.
//=====
```

```

typedef struct Interface {      // a.k.a. IF

struct Interface *Next;       // Next interface on chain.

//
// Interface to the link layer. The functions all take
// the LinkContext as their first argument. See comments
// in llip6if.h.
//
void *LinkContext;           // Link layer context.
void (*CreateToken)(void *Context, void *LinkAddress,
                    IPv6Addr *Address);
void (*ReadLLOpt)(void *Context, uchar *OptionData);
void (*WriteLLOpt)(void *Context, uchar *OptionData,
                  void *LinkAddress);
void (*ConvertMCastAddr)(void *Context,
                        IPv6Addr *Address, void *LinkAddress);
void (*Transmit)(void *Context, PNDIS_PACKET Packet,
                uint Offset, void *LinkAddress);
NDIS_STATUS (*SetMCastAddrList)(void *Context, void *LinkAddresses,
                                uint NumKeep, uint NumAdd, uint NumDel);

uint Index;                  // Node unique index of this I/F.
uint NTECount;               // Number of valid NTEs on this I/F.
NetTableEntry *NTE;         // List of NTEs on this I/F.
AddressEntry *ADE;          // List of ADEs on this I/F.
NeighborCacheEntry *FirstNCE; // List of active neighbors on I/F.
NeighborCacheEntry *LastNCE; // Last NCE in the list.
int DisableND;              // Is Neighbor Discovery allowed?
uint NCECount;              // Number of NCEs in the list.
uint TrueLinkMTU;           // True maximum MTU for this I/F.
uint LinkMTU;               // May be smaller than TrueLinkMTU.
uint CurHopLimit;           // Default Hop Limit for unicast.
uint BaseReachableTime;     // Base for random ReachableTime (in ms)
ULONGLONG ReachableTime;    // Can assume reachable(in system ticks)
uint RetransTimer;          // NS timeout (in IPv6Timer ticks).
uint DupAddrDetectTransmits; // Number of solicits during DAD.
uint RSCount;               // Number of Router Solicits sent.
uint RSTimer;               // RS timeout (in IPv6Timer ticks).
long Validate;              // RCE validation counter.
uint LinkAddressLength;     // Length of I/F link-level address.
uchar *LinkAddress;         // Pointer to link-level address.
uint LinkHeaderSize;        // Length of link-level header.
KSPIN_LOCK Lock;           // Interface lock.

int MCastSynchNeeded;       // Is SetMCastAddrList needed?
KMUTEX MCastLock;          // Serializes SetMCastAddrList
                             // operations
LinkLayerMulticastAddress *MCastAddresses; // Current addresses
uint MCastAddrNum;         // Number of link-layer mcast addresses
uint MCastAddrNew;         // Number added since SetMCastAddrList
int LoopbackCapable;        // Can we disable loopback of this IF.

} Interface;

#define SentinelNCE(IF) ((NeighborCacheEntry *)&(IF)->FirstNCE)

```

//=====

Within the struct of the Interface, a number of functions that are called by pointers are specified with function prototype. The “void (*transmit)” function was defined in the Interface to call the lower level link layer functions in order to do the actual packet sending work. Three link layer sending routines can be called, either LanTransmit() function which is located in the tcpip6/ip6/lan.c module, LoopTransmit() function located in tcpip6/ip6/loopback.c module, or TunnelTransmit() function located in tcpip6/ip6/tunnel.c module depends on the Interface’s status.

3.4 Current work on MSR IPv6

The MSR IPv6 project was started in late 1996. Currently Microsoft Research has published their release 1.2, both source code and binary code, together with the available ported test applications, on their web site. They claimed that they have participated in interoperability testing at the UNH Inter-Operability Lab and have a machine on the 6bone, an organization that coordinates and provides a testing backbone of IPv6 with sites all over the world.

Release 1.2 does not yet support full mobility, authentication or encryption. The next future release (will be 1.3) will cover the above issues.

Mobility refers to the capability that can provide mobile nodes the ability of “plug-and-play” network access at any remote subnet outside its home subnet. The IPv6 basic mobility scheme involves setting up a secure tunnel between the mobile node and a home agent [8]. The home agent, as known as a “proxy” for the mobile node, can receive packets that were sent to the mobile’s home address, and relay them to the mobile’s current location using IPv6 tunneling [8]. In MSR IPv6 release 1.2 some of the mobility routines are provided in the “tcpip6/ip6/mobile.c” module. Yet it still needs more add-ons to support complete mobility.

The authentication and encryption mechanism included in the IPv6 specification will provide a dramatic improvement in the security of the Internet. In MSR IPv6, the “tcpip6/ip6/security.c” module provided some of the security routines for IPv6. The “tcpip6/algrthms/md5.c” module and the “tcpip6/algrthms/hmac_md5.c” module also provided some routines that implement the MD5 algorithm, which is derived from the RSA Data Security Inc.’s MD5 Message Digest algorithm. It does not yet cover the whole

security encryption in IP transmissions. Future releases will have authentication and security support [2].

Chapter 4 The instrumentation model on NT platform

In general, instrumentation is the process that injects devices into hardware or instructions into software to monitor the operation of a system or component. In our case, it refers to the later one. An instrument model is the hardware and software platform that we can build the instrumented protocol stack and run our test applications. It includes physical machines, network connection, instrument connection (in our case, the kernel debugging connection), operating system, network protocol stack and test applications. The following narratives describe our instrumentation model and technology.

4.1 The test bed

In order to instrument and test the protocol stack, we need a test bed. The MSRIpV6 protocol stack is compiled and installed in the machines of the test bed, while test applications are also run on it to invoke the interested function calls inside the stack by sending and receiving test data through the network. We used an isolated and standalone network for our test bed to eliminate unwanted network traffic.

4.1.1 The hardware of the test bed: host & target

In the 3COM lab at room 114 of building 20, we connected 4 Pentium-200MHz personal computers to the same branch of our 100BaseT fast Ethernet network. But only two of them are involved in the instrumentation and testing of IPv6. They both use the 3COM 3C905 NIC as their network interfaces. Following is the summary of hardware details:

CPU:	Pentium 200MHz
Memory:	SRAM 64MB
OS:	Windows NT 4.0 Workstation Build 1381 with Service Pack-3
Network:	100BaseT Fast Ethernet
NIC:	3COM 3C905 Etherlink-II

- Receive Data connected to Transmit Data
- Ground connected to Ground

In order to setup a functioning target-host connection, the following steps must be taken:

Step-1. Connect the null modem cable between the COM ports of the two machines.

Step-2. Start up the host machine. This is the machine we use to observe, debug and control the other machine and collect the instrumentation data.

Step-3. On the host machine, start the *WinDbg* program from SDK tool kits.

Step-4. In the WinDbg window, click at "option" menu, choose "kernel debugging", make a check on the correct COM port and baud rate. We used 56000 baud and COM1 on both machines. The "kernel debug enable" option must be checked to enable the kernel-debugging mode in the target machine.

Step-5. In the WinDbg window, click on the "run" button to start debugging. It should say "waiting to connect on COM1 @ 56000 baud".

Step-6. Start up the target machine. On the target machine, modify the *boot.ini* file at the root directory of the booting drive. Find out the line

```
"multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows NT Workstation Version 4.00", modify it to:  
"multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows NT Workstation Version 4.00 WinDbg Target  
/DEBUGPORT=COM1 /BAUDRATE=56000"
```

Step-7. Save the changes to *boot.ini* (make a backup copy if it's needed) and restart the target machine. Make sure the "kernel debugging enable" is chosen when it asks for boot option during restart.

Step-8. Since we have started *WinDbg* at step-5, we should be able to see the booting sequence in NT (including kernel loading info and drivers loading info) on the *WinDbg* window of your the machine. It's important that the *WinDbg* must be running before the target is boot. Otherwise the connection won't work.

Once we have a functioning host-target instrumentation model setup, we can start to instrument the stack and run our test applications. This model uses the NT kernel debugging mechanism to output text information directly from the instrumented version of IPv6 stack, which is a kernel mode driver of the NT operating system, to the WinDbg window running on the host machine.

4.2 Software Environment

The Windows NT 4.0 Workstation operating system with service pack 3 is probably the most widely used one in the NT product families. It is a 32-bit, preemptive, multitasking, multi-user, portable operating system designed around a micro-kernel architecture.

Our instrument model is based on this operating system because 3COM is interested in the performance of MSR IPv6 running on NT with their Fast EtherLink NIC cards and drivers. Windows NT has been a good platform for protocol development. It accommodates new protocols, loadable at run-time with ease. The kernel debugger WinDbg provides a good source-level debugging environment [2]. Although Microsoft Research said MSR IPv6 stack also works well on Windows NT 5.0, we have only tested it on NT 4.0 Build 1381 with SP3 in all of our test experiments.

4.2.1 General implementation of a protocol stack in Windows NT

7. Application Layer	Windows API Applications		TCP/IP apps, telnet, ftp, ...
6. Presentation Layer	NetBIOS		Windows Socket
5. Session Layer	NetBIOS over TCP/IP		
4. Transport layer	TDI Interface		
3. Network Layer	NWLink	NetBEUI (NBF)	TCP/IP
2. Data Link Layer	NDIS Interface		
	Network Adapter Driver		
1. Physical Layer	Network Adapter (NIC)		

Figure 4.2 Windows NT Networking Protocol Architecture

In the Windows NT networking architecture, the TCP/IP protocol stack is designed to interface upward with the TDI (transport device interface) layer servicing the Windows socket or NetBIOS. It interfaces downward with NDIS (Network Driver Interface Specification) layer, which is the Microsoft/3COM specification for the interface of network device driver.

Network protocols in Windows NT are dynamically loadable device drivers, much like any other device drivers that are usually located in the “WINNT/system32/drivers/” folder. In order to add a new protocol into NT, two new components must be generated and installed: a kernel-mode driver (it’s called *tcpip6.sys* for the MSR IPv6 stack) and a user-level helper routine library (*wshipv6.dll* for MSR IPv6 stack) to support the access to the driver via the socket layer [2].

The MSR IPv6 protocol stack was implemented as a standalone stack, as opposed to an integrated implementation with both IPv4 and IPv6. This approach makes it possible to instrument and test the IPv6 stack while having the conventional IPv4 stack working in the same machine without affecting normal network traffic. However when coming to a real product usage, the hybrid stacks approach might be superior for most scenarios[2].

4.2.2 The MSR IPv6 source code, implementation hierarchy

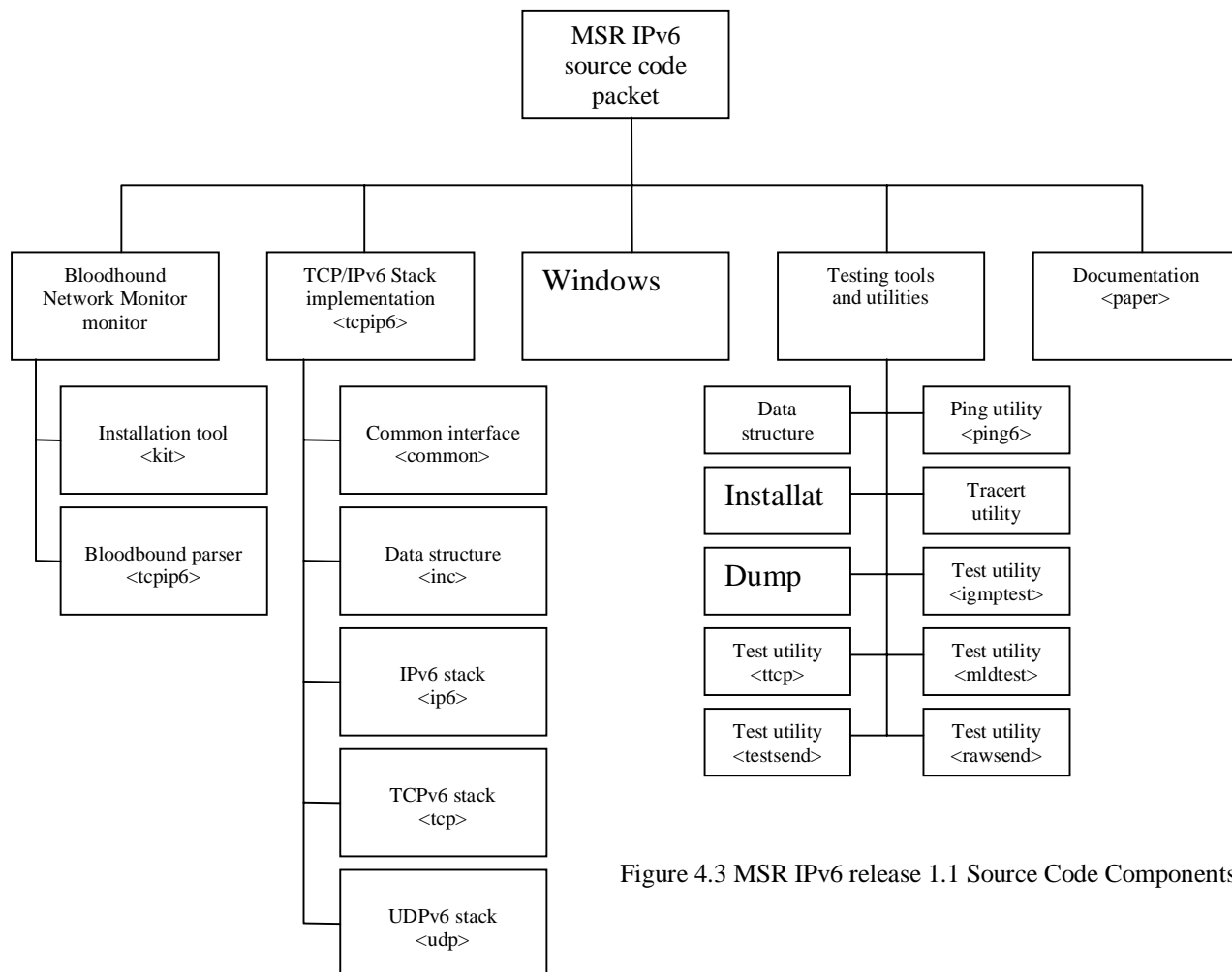


Figure 4.3 MSR IPv6 release 1.1 Source Code Components

In mid-98, Microsoft Research released their first prototype implementation of IPv6 protocol stack for Windows NT platform (known as MSR IPv6). They put the stack as well as the source code for the stack in the public domain for testing and research investigation. In February 1999 they released a version of 1.2 with updates to 1.1 with adding mobility and security functionality. All the instrumentation of this study is based on the source code from MSR IPv6 release 1.1, which has no major difference with release 1.2 in the core part of the stack.

The entire MSR IPv6 implementation source code (release 1.1) consists of several components, each with its own directory listed as below:

- *Tcpip6*, the core IPv6 stack with transport protocols TCP and UDP built on its top;

- *Wship6*, the Windows socket helper DLL routine library;
- *Netmon*, the bloodhound network monitor;
- A number of test utilities including ping6, tracert6, igmp6test, mldtest, rawsend, ttcp, ipv6, and the installation kit used to create an install disk for the new protocol stack driver.
- *Paper*, the documentation of the implementation of MSR IPv6 protocol stack.

All the source modules for building the IPv6 stack driver *tcpip6.sys* are located within the directory “/tcpip6”. It includes the TCP and UDP modules (basically the same as those of IPv4) for transport layer, the IPv6 core modules for the Internet layer and some of the transmit modules for data link layer. They are separated into 5 directories:

- *Common*: the common TDI interface modules.
- *Inc*: high level specification for all other kernel modules.
- *Ip6*: software modules for network layer IPv6 components and data link layer support modules.
- *Tcp*: TCP transport layer modules.
- *Udp*: UDP transport layer modules.

The total size of release 1.1 implementation source code is about 36,000 lines of C code with 36 source files, for release 1.2, it's about 42,800 lines with 39 source files. These files do not include any test applications, not even the Windows socket helper routine library (*wship6.dll*) module. The executable module generated for MSR IPv6 is *tcpip6.sys*, which can be found in the “/tcpip6/common/obj/i386/free” or “/tcpip6/common/obj/i386/checked” directory, and it should be installed into the “winnt/system32/drivers/” directory as one of the kernel mode drivers. The below diagram shows its interface to the other Windows NT network components. In this diagram, both *tcpip6.sys* and *wship6.dll* are the newly added-in modules.

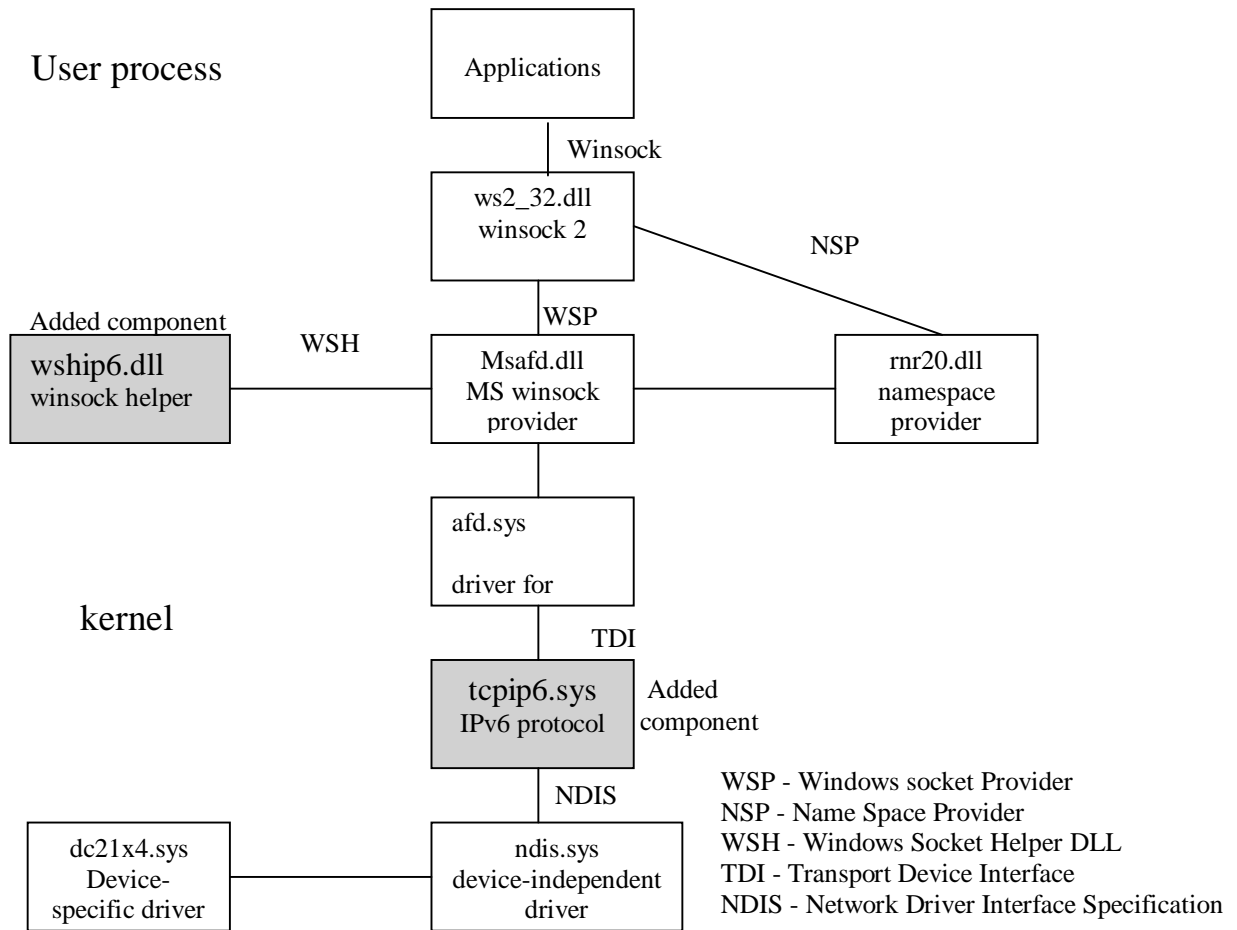


Figure 4.4 MSR IPv6 protocol components' interfaces in the NT networking architecture

4.2.3 SDK WinDbg utility

The Windows NT debugger (WinDbg) is a 32-bit application that, along with a collection of DLLs, is used for debugging the Kernel, the device drivers, and applications. The WinDbg utility is the standard debugger for Windows NT driver programmer. It can be used to debug both user mode and kernel mode drivers. It is a part of the NT Software Development Kit (SDK).

Since the instrumented IPv6 stack `tcpip6.sys` is running in NT kernel mode, it has no direct contact with any I/O process. In other words, outputting instrumentation results can not be done through direct I/O routines such as `printf()`, `fprintf()`, or `sprintf()`. We had experimented with outputting the instrumentation results

into a disk file in kernel mode, but we found that it's not worth the implementation complexity and the processing overhead.

A quick and easy solution to this problem is utilizing the WinDbg utility provided in NT SDK. WinDbg's command window allows kernel programmer to print debugging string information directly from kernel mode. This process requires a host-target connection between two NT machines. During the instrumentation process, WinDbg is just like a string information receiver for getting the instrumentation data (time stamps) out from the target machine in kernel mode. The debugging print function we used to get things out to WinDbg on the host machine is "DbgPrint()".

The WinDbg provides an easy-to-use GUI for debugging and controlling the target object. It has 8 windows within the application canvas, namely they are:

- (a). Watch window: Monitors the variables and expressions set by the Watch Expression command.
- (b). Locals window: Shows value of the local variables within the function currently being stepped through.
- (c). Registers window: Shows current contents of the CPU registers.
- (d). Disassembly window: Shows the assembly instructions being debugged.
- (e). Command window: Allows to enter debugger commands and display debugging information.
- (f). Floating-point window: Shows current contents of floating-point registers and stack.
- (g). Memory window: Shows current contents of memory.

In our instrumentation model, only the Command window of WinDbg is used. Furthermore, no debugging command needs to be sent to target from the Command window because it's important to minimize the debugging impacts to the target system where latency and throughput are measured.

4.2.4 Enable target machine's kernel debugging in NT

In order to allow the target machine to output instrumentation data (string information) directly from kernel mode IPv6 stack driver, its kernel-debugging mode of OS has to be enabled. To enable the target machine's kernel debugging mode, the booting initialization file *boot.ini* of Windows NT located at the boot drive must be modified by adding the `DEBUGPORT=debugport` and `BAUDRATE=baudrate` parameters into the boot control string under the "[operating system]" section. For example, below is a *boot.ini* file for a Windows NT 4.0 machine without kernel debugging enabled:

```
[boot loader]
timeout=5
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(2)\WINNT="Windows NT Workstation Version 4.00"
multi(0)disk(0)rdisk(0)partition(2)\WINNT="Windows NT Workstation Version 4.00 [VGA mode]"
/basevideo /sos
C:\ = "Microsoft Windows"
```

After enabling the kernel-debugging mode, the file should look like:

```
[boot loader]
timeout=10
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(2)\WINNT="Windows NT Workstation Version 4.00 WinDbg Target" /DEBUGPORT=COM1 /BAUDRATE=56000
multi(0)disk(0)rdisk(0)partition(2)\WINNT="Windows NT Workstation Version 4.00 [VGA mode]"
/basevideo /sos
C:\ = "Microsoft Windows"
```

Enabling kernel-debugging mode is only done in the target machine, not the host machine.

4.2.5 Compiling the IPv6 stack with NT DDK

To be able to build the MSR IPv6 stack, the following tools are required in the build machine:

Microsoft Windows NT Version 4.0 Device Driver Kit (DDK), which requires Windows NT 4.0, the Platform Software Development Kit (SDK) for Win32 and Visual C++ 5.0 (or an equivalent 32-bit C compiler). NT DDK is a device driver development toolkit for Windows NT operating system provided by Microsoft. We used DDK to build the protocol driver for the IPv6 stack, because the make file that is shared by all the driver components of the Windows NT DDK greatly simplified the compiling process.

We also tried to compile the IPv6 stack with Microsoft Visual Studio version 5. It takes some effort to integrate the NT DDK header file directory tree into Visual Studio at the searching directory setting. This is done under the "Tools-Options-Directories" menu item of Visual Studio. The IPv6 source files needs to reference these DDK header files (e.g. ntddk.h, ndis.h, tdi.h, and a lot more) to build up the TDI and NDIS network interface. Some extra effort is also needed to create a multiple-directory project for the stack in Visual Studio because in the tcpip6 directory tree, conflicted file names are found in different branches of the directory

tree (E.g. “init.c” is found in “common”, “ip6” and “udp” sub-directories under “tcpip6” directory representing different files). We took advantage of the browse file created by Visual Studio after compiling the stack and used it as a MSR IPv6 source code browser, since useful features such as caller graphs, calling graphs, global class (function) listing and user-defined keyword referencing are provided in Visual Studio. For practically building an executable of the stack, NT DDK is more convenient and faster. We built all our instrumented versions of IPv6 stack using NT DDK.

To start compiling the whole source and building the stack, either “Checked build” or “Free Build” utility window has to be opened. Check Build is used to build a debugging version of the stack which includes all the debugging symbols and error checking code (e.g. the ASSERT macro) for debugging. Free Build is the building tool to build a release version of the compiled software, which will exclude the extra debugging code from the binary. Both utilities are provided in the DDK to compile and build binary products for Windows NT including executables, dynamic link libraries, and drivers. The window of the build utility is very much similar to the MSDOS console command prompt window. DOS commands can also be executed in this window. After entering the proper directory path for the version of IPv6 stack which is to be built (typically .../msripv6/tcpip6/), key in the command “build” or “build -cef” to start building the stack. The “-cef” switch will force re-compile of every module in the source tree. If it’s omitted, nmake.exe invoked by build will only re-compile those modules, which have been modified since the last build.

Before running either the Checked Build or Free Build utility, the path to the compiler must be set in order to direct the build utility to invoke nmake.exe, a utility that actually compiles the source according to the make files. This can be done by running a batch file named “vcvars32.bat” located under the “/bin” directory if Visual Studio is used as the default C compiler.

4.2.6 Checked build versus free build

There are two versions of executable for the Windows NT kernel modules, namely the free build version and the checked build version. The free build version of Windows NT is the release version that end users receive in their installation disks. The Free Build version is built with full optimization and contains minimal debugging symbols. The checked build version of NT is shipped only with the DDK, it is used in

debugging drivers and other system code. The Checked Build binaries provide error checking, argument verification, and system debugging code not present in the Free Build binaries.

Generally, using the WinDbg debugging utility requires the target machine to be run on the Checked Built version of the Windows NT kernel executable (e.g. ntoskrnl.exe and hal.exe) which is much slower in speed than the normal Free Build version. Fortunately, our instrumentation model doesn't require extensive involvement in the debugging functionality. Only the kernel mode printing call DbgPrint() is used to output string information on the command window of WinDbg debugger. Because of this, we can still use the normal Free Build version of Windows NT for our IPv6 stack instrumentation. This is very important since we are measuring short execution latency within the IPv6 stack and we don't want extra overhead involved from the Checked Build of NT that would allow the measuring latency to be meaningless numbers.

On the other hand, when we build our instrumented version of MSR IPv6 stack driver, we can either use NT DDK free-build tool to build a "slim" version of tcpip6.sys, which has no debugging symbols in it; or we can use the checked-build tool to build a "fat" version of tcpip6.sys. The selection of the above two depends on the instrumentation stage. We began with a checked build of the stack first. Once we are pretty sure about the probing points of the stack and the interpretation of data, we switched to free build since it's closer in operation to the actual stack without instrumentation. Most of our instrument results are obtained from a free build of the stack.

4.2.7 Creating an installation kit for a new build of IPv6 stack

Once we have built a version of the executable of IPv6 stack, we need to create an installation kit which will be used to install the protocol stack into NT. The stack executable is located at the path of `"/msripv6/tcpip6/ common/obj/i386/checked/tcpip6.sys"` if checked build is used, or at `"/msripv6/tcpip6/common/obj/i386/free/ tcpip6.sys"` if free build is used.

In the downloaded MSR IPv6 source tree, there is a command script called "mkkit.cmd" located in `"/msripv6/kit/"` directory. This DOS command script will copy the necessary executable files as well as the documentation files from the source tree into a destination directory and create the installation kit. The files that need to be copied include:

- oemsetup.inf The setup file for Windows NT 4.0

- netip6.inf The setup file for Windows NT 5.0
- license.txt The license text document
- license.htm The license hypertext document
- ReadMe.txt The read me text document
- tcpip6.sys The MSR IPv6 protocol stack driver
- wship6.dll The Winsock helper routine dynamic linking library
- msafd.dll The Winsock provider routine dynamic linking library(bug fixed)
- ipv6.exe The IPv6 status information dumping utility program
- ping6.exe The IPv6 ping utility program
- tracert6.exe The IPv6 trace route utility program
- ttcp6.exe The IPv6 test TCP connection utility program
- mldtest.exe The IPv6 multicast listener test utility program
- testsend.exe The IPv6 test sending utility program
- nslookup.exe The IP nslookup utility program(bug fixed for IPv6)

After placing the above files into your installation directory, manually or by the script, you can install the stack into the NT operating system.

4.2.8 Install or upgrade the new build of IPv6 stack in NT

This task is facilitated by Windows NT's run-time install/uninstall ability for protocol drivers. To install the MSR IPv6 on an NT machine for the first time, simply make a right click on the desktop icon "Network Neighborhood" and choose "Properties" from the menu. Click on the "Protocols" tab on the top, here you can see a list of network protocols currently installed in your machine. Click on the "Add..." button (make sure you have the system administrator's privilege to do that) and select "Have disk..." option, then browse to the drive and directory where the installation kit is located. Then the IPv6 stack should be able to install itself. To update the existing protocol driver with a newer version, follow the same procedure as above. But instead of clicking on "Add...", the button "Update" should be used.

The above install procedure will copy the files from the install kit into the appropriate places and add entries to the system registry for IPv6 configuration. If you later modify IPv6 components, you can replace just the affected file or files without having to uninstall/install IPv6 again. The protocol stack itself (tcpip6.sys) is installed in the “winnt\system32\drivers” directory. The Winsock helper dynamically linked library for the INET6 address family (wship6.dll) and all the user applications and utilities (ipv6.exe, ping6.exe, tracert6.exe, ttcp6.exe, etc) are located in the “winnt\system32” directory.

After installing the stack, the system needs to be rebooted before the new setting is takes effect. In our host-target instrumentation model, the host machine was installed with the free build version of MSR IPv6 which was directly downloaded from MSR, while the target machine was installed with a free or checked build of a instrumented version of the IPv6 stack, with measurement code embedded inside it.

4.3 Instrument technology

Generally, our approach of measuring latency and throughput inside the protocol stack was rather straightforward. First we studied the source code of MSR IPv6 for TCP/IP and UDP/IP functionality, then we identified the path of execution for sending and receiving. We embedded our instrumentation code which will obtain a timestamp of the current system time at the tops and ends of the functions, or before and after any interested points in the source code along the path. When the IPv6 test applications are run, time stamps are collected from the interested functions or locations, and stored in the buffering memory. To minimize the instrumentation overheads, printout of time stamps from the stack are triggered at the end of each test run by utilizing the host-target debugging connection model. By calculating the time elapsed between different interested probing points of the stack, we can come up with the latency through various parts of the stack for any given payload size. Furthermore, we can come up with the payload throughput according to the following formula: $T = P / L$, where T is the payload throughput, P is the payload size and L is the stop latency for the payload data to be transmitted through the stack.

Since Windows NT is a preemptive, multi-process, multi-thread kernel, the absolute time elapsed for the stack to process the payload may not be the time that CPU spends on running the stack process. Actually, most likely only a portion of the elapsed time is used for CPU to run the stack process. To minimize this measurement deviation caused by CPU utilization, we isolated our test network to avoid any concurrent sending

and receiving on IP or above layers. We also minimize the number of concurrent user processes running in the target machine. However, some extraneous packet traffic remains on the Ethernet test bed on the data link layer, e.g. SMB (Server Message Block) traffic for Windows NT network file systems. This unwanted traffic might cause context switches in the target machine while measuring, and add deviations into our absolute latency result. To have a better analysis about how this affects our instrumentation result, we monitored the network traffic on our test bed using NetXray, a network traffic monitoring and statistic tool that can capture and decode packets on Ethernet. We also used VTune3 to analyze the CPU time dispersion among different kernel and user process. More discussion on the CPU utilization issue can be found in section 5.4.

4.3.1 Instrumentation code, general consideration

In order to obtain accurate data, it is important to minimize the latency overhead introduced by the instrumentation itself. The following issues have been taken into consideration when designing our instrumentation method:

- **Efficiency:** The embedded code for obtaining timestamps and marking locations must be as efficient as possible in terms of execution time. Short code doesn't guarantee short execution time. We have to measure the execution time of the measuring code itself in order to decide if it's efficient enough. See section 4.3.5 for more details.
- **Buffering:** The timestamps must be buffered into memory that can be directly accessed by the protocol stack. We used global scope arrays added to MSR IPv6 stack's data structure to buffer all the timestamps before they are dumped out from kernel mode. The size of the buffer is large enough to hold 64K timestamps.
- **Deferred Printing:** Printing of the timestamps from target to host with the serial connection is a time-consuming operation. In-line printing whenever a timestamp has been obtained will incur extreme heavy overhead to the stack and will slow down the whole system. We have to separate timestamp printing from timestamp collecting and buffering. In our instrumentation implementation, timestamps are buffered in pre-allocated array buffers and a "ping6" utility command is used to trigger the printing of timestamps after the test run has been finished.

4.3.2 Obtaining timestamps

One of the critical parts of our instrumentation is obtaining accurate system timestamp. We first tried to use the Windows NT's kernel API calls provided in the DDK library to obtain current system time as our measurement timestamps. The API calls include:

- `Void KeQuerySystemTime (OUT PLARGE_INTEGER CurrentTime);`
Obtains the current system time. It actually obtains the count of 100ns intervals since January 1, 1601.
- `Void KeQueryTickCount (OUT PLARGE_INTEGER TickCount);`
Obtains a count of the interval timer interrupts that have occurred since the system was booted.
- `ULONG KeQueryTimeIncrement ();`
Returns the number of 100-nanosecond units that are added to the system time each time the interval clock interrupts.
- `Void NdisGetCurrentSystemTime (PLARGE_INTEGER pSystemTime);`
This function call is defined to be the same as KeQuerySystemTime() in ndis.h of DDK library.

After a few tests we figured out that the above kernel mode API calls can only provide a time resolution of up to about 10 milliseconds (ms). KeQueryTimeIncrement() always return 100144 on our Pentium-200 CPU based target machine. 100144 is the number of 100-nanosecond intervals that is added into the program tick counter for each time interrupt (while KeQueryTickCount increments by 1). The result from KeQuerySystemTime() always advances in steps of 100144 program ticks which is $100144 * 100 \text{ ns} \approx 10 \text{ ms}$. With this resolution of 10ms, most of the stack latency which is in the microseconds level will not be visible to the instrumentation code. So basically the above NT kernel mode API calls do NOT work with our instrumentation model for the given measurement task in MSR IPv6.

Because of this, we have to make our own timestamping function call with better time resolution. We took advantage of the RDTSC (Read Time Stamp Counter) assembler instruction, which is available in the later version of Intel x86 Pentium processor family, to obtain the current system time from the Pentium CPU hardware tick counter. As our target machine is running on 200MHz CPU clock, the time resolution with RDTSC could be as fine

as 1/200MHz, which is in step of 5 nanosecond. Two 32-bit variables were passed to receive the high 32-bit and the low 32-bit of the timestamp correspondingly when the function is called. Below is the function listing:

```
//=====start of getCPUtime=====
void getCPUtime(unsigned long int *highc,
                unsigned long int *lowc)
{
    unsigned long int    h, l;
    _asm {
        _emit 0x0f    ; RDTSC
        _emit 0x31
        mov l, eax
        mov h, edx
    }
    *highc = h;
    *lowc  = l;
}
//=====end of getCPUtime=====
```

This timestamping approach has proved to work very well in our instrumentation model. Whenever the function is called, it returns the current Pentium CPU tick count with two unsigned long integers. The returned value multiplied by 5 will be the absolute system time in nanoseconds. In other word, this absolute clock advances in an increment of 5ns, which is a much better time resolution than that provided by the NT kernel API calls.

4.3.3. Buffering timestamps

As we mentioned in section 4.3.1, buffering timestamps is essential for reducing instrument overhead to a minimum. We added the following global scope definition into the file `ntdisp.c` located in `/tcpip6/common/` directory for timestamp buffering data structure:

```
//=====timestamp buffering data structure=====
LARGE_INTEGER Xptime[XPMAX];           // timestamp array
unsigned int   XPindex = 0;             // current write index in buffer
unsigned int   XPlocation[XPMAX];      // location array
unsigned int   XPthread_id[XPMAX];     // thread id
```

```
unsigned int  prntTSi=0;                // printing index for dumping TS
//=====
    The constant XPMAX was #define'd to 65535 (64K) in the "tcpip6/common/oscfg.h" module. At each
MSR IPv6 module where we want a timestamp from, the above definition is repeated with "extern" specifying.
The timestamp buffering arrays can be accessed from anywhere of the source modules of the stack with the
external specifying global scope data structure.
```

For example if we want to get a timestamp at the function IPv6Send0() in the module send.c, which is located at tcpip6/ip6/ directory, we define the timestamp arrays as external variables as below:

```
//=====local timestamp buffering definition=====
extern LARGE_INTEGER Xptime[XPMAX];    // timestamp array
extern unsigned int  XPindex;          // current write index
extern unsigned int  XPlocation[XPMAX]; // location array
extern unsigned int  XPthread_id[XPMAX]; // thread id
//=====end of local definition=====
```

At the timestamp collecting point, instrumentation code is embedded as below:

```
//=====added in function local declaration=====
static int XPi=0;
int XPk=++XPi;
//=====added in function's head=====
if (XPindex==XPMAX)
    {DbgPrint("IPv6Send0(0): Array Full! New timestamp dropped!\n");}
else
    {
    getCPUtime(&Xptime[XPindex].HighPart, &Xptime[XPindex].LowPart);
    XPlocation[XPindex] = 18; // Assign 18 to this location
    XPthread_id[XPindex]=XPk; // store the id with timestamp
    XPindex++;                // increase index
    }
//=====end of instrument point=====
```

We used in-line instrumentation code embedding instead of calling a probing function, because it helps to reduce the instrumentation overhead to the lowest level. For each timestamp, four items are buffered: high 32-bit of timestamp, low 32-bit of timestamp, location number corresponding to a physical location in the source model, and thread identification number (See section 4.3.6). The timestamp data stays in the buffering array until they have been output and cleared, or the NT operating system has been shut down. If for some reason the buffer arrays are full, newly arrived timestamps will be discarded with a warning string printing to the WinDbg command window on the host machine.

4.3.4 Outputting timestamps

After each time we ran the test applications, the collected timestamps need to be output and the buffer needs to be cleared for next use. This process also depends on how many timestamps are generated to the buffer for a test run. Since there are 64K positions in the buffer, usually it's quite enough for one or several test runs.

The kernel debugging API call `DbgPrint()` was used to print the timestamps to the WinDbg command window running on the host machine. `DbgPrint` is specified as an NT DDK library routine:

- `ULONG DbgPrint(PCH pchFormat, ...);`

`DbgPrint()` writes a string to the WinDbg KD Command window. The parameters of `DbgPrint` are the same as those used by the C runtime function `printf()`. Only kernel-mode drivers can call `DbgPrint`. The `ntddk.h` file defines a `KdPrint` macro that is equivalent to `DbgPrint`, except that it is a no-op in free builds and outputs a string only in checked builds.

We put our timestamp output code at the end (right before the last return statement) of the function `TCPDispatch()` in the module `ntdisp.c` located in the “`/tcpip6/common/`” directory. This point of the program is not run by the normal data-sending functionality, while the utility command “`ping6`” can trigger the stack running through it. Because of this we can easily trigger the time stamp dumping by entering a `ping6` command on the target machine's MSDOS console.

The actual code for dumping timestamps is straightforward. When dumping is finished, we clear all the buffers by resetting the write index to zero. The embedded code is listed as below:

```
//===Printing timestamps. Time consuming!=====
for (prntTSi=0; prntTSi < XPindex; prntTSi++) {
    DbgPrint("%i: tid=%u, t.H=%u, t.L=%u, d=%i, location=%i\n",
            prntTSi, XPthread_id[prntTSi], XPtime[prntTSi].HighPart,
            XPtime[prntTSi].LowPart, (XPtime[prntTSi].LowPart -
            XPtime[prntTSi-1].LowPart), XPlocation[prntTSi]);
}
XPindex = 0; // reset writing index, clear buffers
//=====
```

Before triggering timestamp dumping, the WinDbg command “`.logopen filename.txt`” (there is a dot before the “`logopen`” keyword as part of the command) can be used to start scripting text from screen to file.

After the timestamp dumping on the WinDbg window is finished, the WinDbg command “.logclose” will close the scripting file and save it in disk.

The time for printing a thousand timestamps, for example, is about 20-30 seconds depending on the speed of the serial connection between the target and host machine (we used 56000 Baud). During dumping, the target machine is in a “hanged-waiting” mode. All the processes are postponed until the dumping is finished. This process is automatically controlled by the kernel debugging mechanism of NT.

4.3.5 Measurement code overhead

Since the embedded instrument code will add overhead to the stack and increase the latency to be measured, time overhead of the instrument code itself must be eliminated for more accurate results. Before any instrument measurement test, we built a special version of the stack dedicated for measuring time consumption of our embedded instrumentation code. We collected “doubled” and “tripled” timestamps from different locations of the stack. “Doubled” timestamping means two timestamps are collected consecutively with no code running between them. “Tripled” means three are collected in the same way. The time differences between these doubled or tripled timestamps are the time needed for getting absolute time with RDTSC and putting it into buffer, which is the instrumentation overhead added to the stack. By averaging the time differences we got:

- Averaged instrumentation overhead: 230 CPU ticks = 1150 ns

The above time of overhead is added to the measurement result each time when the instrument code (shown in section 4.3.3) is run for collecting and buffering a timestamp. When calculating the time difference between absolute time points, the above instrument overhead multiplied with one plus number of timestamps between the two interested points should be eliminated. For example:

```
!5000
0:  tid=145, t.H=16539, t.L=3745545802, d=-549421494,
location=2
```

1: tid=145, t.H=16539, t.L=3745551694, d=5892,
location=3

2: tid=146, t.H=16539, t.L=3745552966, d=1272,
location=2

3: tid=146, t.H=16539, t.L=3745555390, d=2424,
location=3

4: tid=147, t.H=16539, t.L=3745556059, d=669,
location=2

5: tid=147, t.H=16539, t.L=3745556382, d=323,
location=3

6: tid=148, t.H=16539, t.L=3745717129, d=160747,
location=2

7: tid=149, t.H=16539, t.L=3745790503, d=73374,
location=2

8: tid=12, t.H=16539, t.L=3745810681, d=20178,
location=18

9: tid=12, t.H=16539, t.L=3745932364, d=121683,
location=19

10: tid=150, t.H=16539, t.L=3746810944, d=878580,
location=2

11: tid=151, t.H=16539, t.L=3746866414, d=55470,
location=2

12: tid=152, t.H=16539, t.L=3746909431, d=43017,
location=2

13: tid=153, t.H=16539, t.L=3746951749, d=42318,
location=2

14: tid=154, t.H=16539, t.L=3746993311, d=41562,
location=2

15: tid=155, t.H=16539, t.L=3747058339, d=65028,
location=2

16: tid=155, t.H=16539, t.L=3747063265, d=4926,
location=3

The above timestamps were collected when a payload size of 5000 bytes was sent with UDP mode by test application ttcp6.exe from target to host. Timestamps are collected at location 2, 3, 18, 19. (Please reference to Table 4a instrumentation location numbering) where 2 and 3 are the head and the end of function “TCP Dispatch Internal Device Control()” in tcpip6/common/ntdisp.c module, and 18 and 19 are the head and the end of function “IPv6 Send Fragments()” in tcpip6/ip6/send.c module.

To calculate the absolute time it takes to executes the IPv6SendFragments() function which achieved fragmenting and sending all the fragments, we calculate the time difference between the two timestamps from location 18 and 19, which is $\Delta(T1, T2) = 121683$ CPU ticks, and subtract with the measurement overhead, which is $(1+0)*230 = 230$ CPU ticks (here 0 is added because there is no timestamping between location 18 and 19), then we get the actual latency in this case = $\Delta(t1, t2) = (121683-230) = 121463$ CPU ticks = 607315 ns. (5ns/tick). The measurement overhead is $230/121436=0.19\%$ of the measuring result. In most of the cases, our measurement overhead is less than 1% of the measured latency.

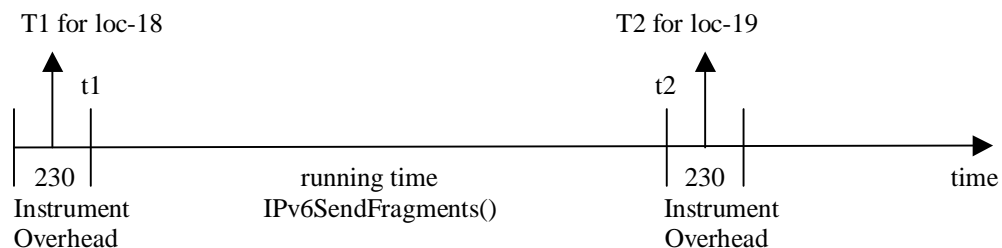


Figure 4.5 Subtracting measurement overhead from measured latency

Actual latency = measured latency – measurement overhead,

$$\Delta(t1, t2) = \Delta(T1, T2) - 230 \text{ ticks}$$

4.3.6 Identifying functions called by different kernel threads

Since Windows NT is a multi-process multi-thread kernel, occasionally we observed that timestamps for the same function (both head and end) called by different threads interleave with each other in our instrument result. In order to be able to identify the calls made by the same thread and pair-up the function-head timestamps with the function-end timestamps, we need to use a mechanism to identify them.

We declared a static integer variable `XPi` and a non-static integer variable `XPk` at the instrumentation point, usually at the head of instrumented function. Both variables are local scope variables. The static variable `XPi` was initialized to be zero at the first time when the function was called, and the other integer variable `XPk` was assigned with the actual value of `XPi` plus one.

```
static int XPi=0;
int XPk=++XPi;
...

XPthread_id[XPindex]=XPk; // store the id with timestamp
```

The static variable will be assigned with a permanent space in the memory. It can keep its initialized value of zero and increments it by one for each time when the instrumented function was called. At run time, this value was unique within the instrumented function's scope, thus can be used to identify the timestamps from that particular call of the function head and end. With this number we can identify the head timestamp and the end timestamp for the same function called by the same thread, because they will have the same `XPk` number.

4.3.7 Test utilities and applications

There are several test applications with their source code that come along with MSR IPv6 release 1.1 source download: `ping6`(ping using ICMPv6 on IPv6), `ttcp6`(test TCP/IPv6 connection), `ipv6`(dumping the stack

configuration and status info), mldtest(), igmpstest, rawsend, testsend, and tracert6. In release 1.2, which came out in February 1999, some new test utilities are added. They include md5test, hmactest and hdrtest. These utilities can be used for testing different functionality and performance of the IPv6 stack. There are not many fancy network applications, which are implemented to run on IPv6 for Windows NT, are included in release 1.1, except the ported NcFTP client application. In release 1.2, MSR added two more ported applications to the package: the Fnord web server application, and RAT with SDR, the multicasting audio conference application.

In our experiments, we mainly use the ttcp6 to generate our test traffic, and use the ping6 utility as a special command to trigger timestamp dumping. The ttcp6 program was a test application originally developed for testing BSD 4.x TCP socket connections. It has been modified to run on MSR IPv6 for the same functions. It set up a connection on port 5001 and transfers fabricated buffers or data copied from standard input (stdin). By default ttcp6 sends a total of 16MB payload data (2048 buffers of 8KB in size) to its receiver, but the buffer length and the number of buffer to be sent can be arbitrarily specified in the command line arguments. This greatly facilitated our testing traffic generation.

Following is an example of using ttcp6 to initiate transferring of a single payload buffer of 2048 bytes to the destination host ::129.65.26.70 using TCP protocol.

On sender's screen:

```
C:\>ttcp6 -t -n1 -l2048 ::129.65.26.70
ttcp6-t: buflen=2048, nbuf=1, align=16384/+0, port=5001 tcp ->
::129.65.26.70
ttcp6-t: socket
ttcp6-t: connect
done sending, nbuf = -1
ttcp6-t: 2048 bytes in 1 real milliseconds = 2000 KB/sec
ttcp6-t: 1 I/O calls, msec/call = 1, calls/sec = 1000, bytes/call = 2048
```

On receiver's screen:

```
C:\pxie>ttcp6 -r
ttcp6-r: buflen=8192, nbuf=2048, align=16384/+0, port=5001 tcp
ttcp6-r: socket
ttcp6-r: accept from ::129.65.26.23
ttcp6-r: 2048 bytes in 1 real milliseconds = 2000 KB/sec
ttcp6-r: 2 I/O calls, msec/call = 0, calls/sec = 2000, bytes/call = 1024
```

The ping6 utility works similarly as its IPv4 counterpart does. It sends ICMPv6 packets to the command argument specified host and checks for the reply message. The round trip time is also timed and printed. In our experiments, the way we used ping6 is different from its normal usage. We identified a hot path for ping6 within the stack's generic dispatcher which is located at the physical end of the function TCPDispatch() in tcpip6/common/ ntdisp.c file. We put the timestamp dumping code at this place. When we ran the ttcp6 to generate our test traffic, this spot of the stack was not run across. After the test runs finish and all the timestamps are stored in the buffer, we ran the ping6 utility to make the timestamp dumping code to be run through, which will trigger the dumping timestamps and clearing of buffer. This is a relatively simple and feasible way to trigger timestamp dumping compared to some other implementation approaches.

4.3.8 Identifying the common path through the stack

One of our objectives is to measure the start and stop latency through the whole TCP/IPv6 or UDP/IPv6 stack. Identifying the common path in the source code for sending and receiving a packet is a key step in the measurement. By definition, a common path is the execution path without involving any error or exception handling functions. This is the only code or functions that need to be executed in a normal case.

For the lack of design documentation of MSR IPv6, we have to do some experiments to find out the common path. We built a special version of the stack so that each of the major functions involved in packet traffic handling will print a line to the WinDbg window with identification of itself both when it was called and exited. The idea was to get the actual execution tracks of the function calls in the stack. Of course this function-tracking build of the stack is not efficient in traffic handling because of the in-line direct printing from the kernel using DbgPrint(). However the printing result gives us a good source of information to figure out the hot path through the stack.

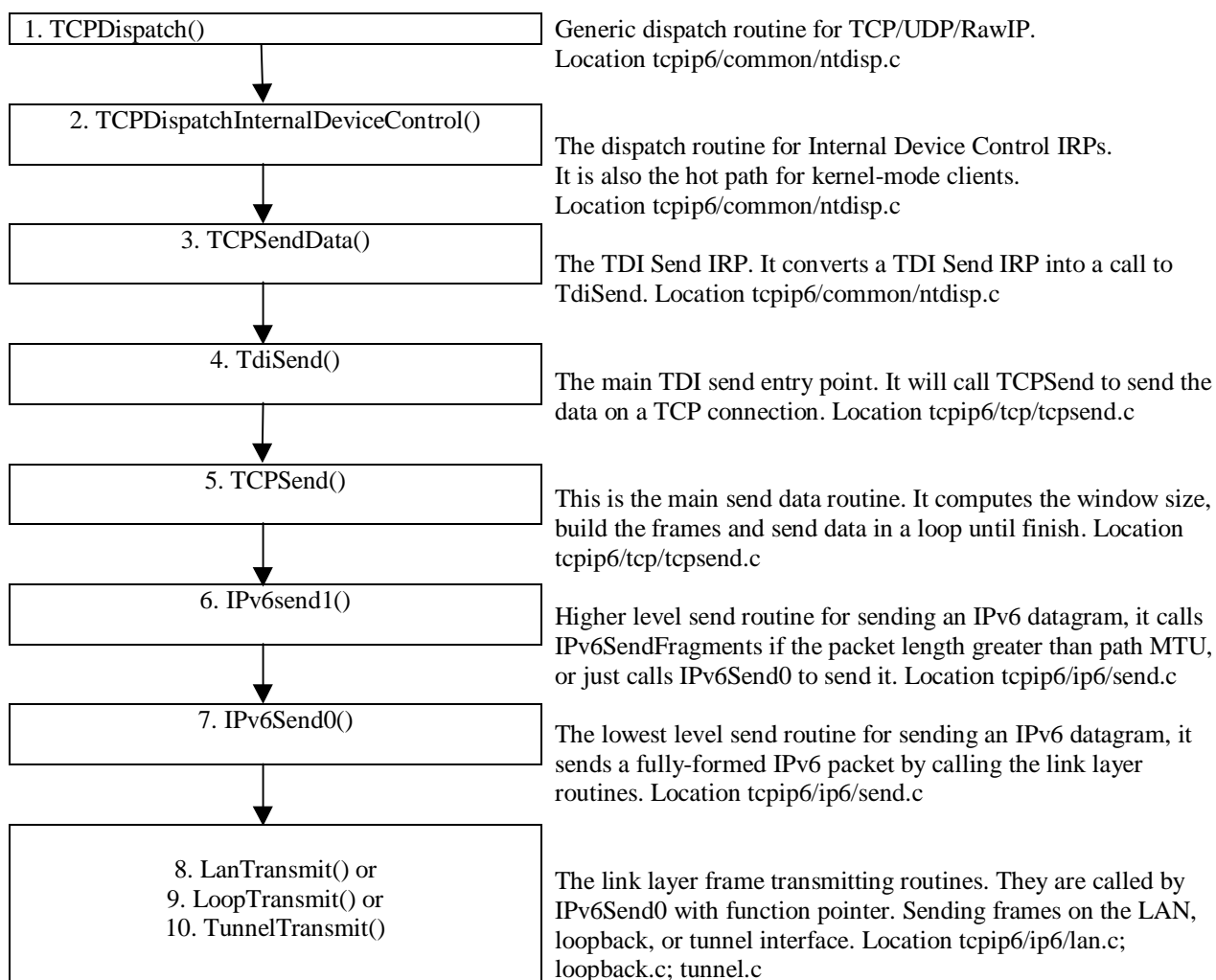


Figure 4.6 Common call path through MSR TCP/IP6 stack on the sender's side (downward)

Referring to Figure 4.6, functions 1 and 2 in the diagram belong to the TDI layer within Windows NT networking architecture (please reference to section 4.2.1). Functions 3, 4, and 5 closely match the TCP transport layer. Functions 6 and 7 belongs to the IP layer while functions 8, 9, and 10 belongs to the NDIS layer which is corresponding to the data link layer in the OSI model.

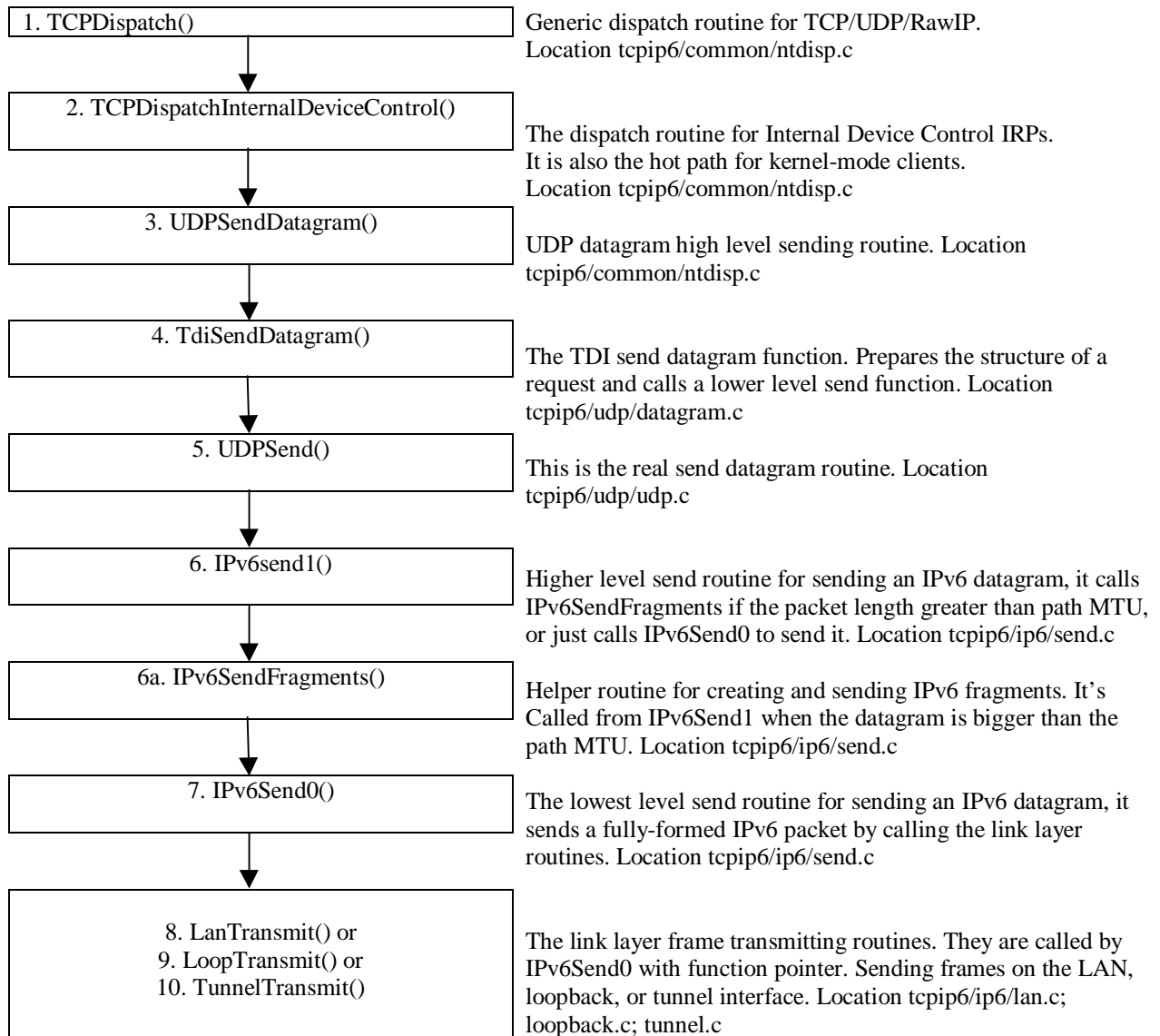


Figure 4.7 Common call path through MSR UDP/IPv6 stack on the sender's side (downward)

We found the similar hierarchy in UDP as that in TCP. In figure 4.7, functions 4, 5, and 6 work similarly in the hierarchy as their counterparts in TCP. Function 6a IPv6SendFragments() only shows up in UDP's common path because in TCP, fragmentation in IP layer is avoided by using the path MTU as its maximum segment size. The IP routine just needs to take the TCP segments, encapsulate with IP headers, and pass them to the NDIS routine calls.

With the help from the browse file feature of Visual Studio, the path can be verified with the caller graphs and calling graphs in the source hierarchy. But the caller/calling graphs can not replace the path-finding

test runs. Because some functions in the path are called by pointers in MSR IPv6 source code, these functions will not show up in the caller graph or calling graph from the Visual Studio browse file since the actual function called will not be known until run time.

4.3.9 Time-stamping locations

A time-stamping location is a place within the MSR IPv6 stack source code where the instrumentation code is embedded and a time stamp was collected whenever it is hit. We decided our time-stamping locations along different functions in the common path (section 4.3.8) according to what type of latency we were measuring. To avoid measurement overhead of string copying, each time-stamping location is hard-coded with a location number (as oppose to a location string) in the embedded instrumentation code, and this location number is stored together with the timestamps of which it has collected. The table below summarizes some of the instrumented locations.

Location number	Directory path	File	Function's Name	Instrumentation Position
0	Tcpip6/common/	Ntdisp.c	TCPDispatch(0)	Function head
1	Tcpip6/common/	Ntdisp.c	TCPDispatch(1)	Function end
2	Tcpip6/common/	Ntdisp.c	TCPDispatchInternalDeviceControl(0)	Function head
3	Tcpip6/common/	Ntdisp.c	TCPDispatchInternalDeviceControl(1)	Function end
4	Tcpip6/common/	Ntdisp.c	TCPSendData(0)	Function head
5	Tcpip6/common/	Ntdisp.c	TCPSendData(1)	Function end
6	Tcpip6/tcp/	Tcpsend.c	TdiSend(0)	Function head
7	Tcpip6/tcp/	Tcpsend.c	TdiSend(1)	Function end
8	Tcpip6/tcp/	Tcpsend.c	TCPSend(0)	Function head
9	Tcpip6/tcp/	Tcpsend.c	TCPSend(1)	Function end
10	Tcpip6/ip6/	Send.c	IPv6Send1(0)	Function head
11	Tcpip6/ip6/	Send.c	IPv6Send1(1)	Function end
12	Tcpip6/ip6/	Send.c	IPv6Send0(0)	Function head
13	Tcpip6/ip6/	Send.c	IPv6Send0(1)	Function end
14	Tcpip6/ip6/	Lan.c	LanTransmit(0)	Function head
15	Tcpip6/ip6/	Lan.c	LanTransmit(1)	Function end
16	Tcpip6/ip6/	Loopback.c	LoopTransmit(0)	Function head
17	Tcpip6/ip6/	Loopback.c	LoopTransmit(1)	Function end
18	Tcpip6/ip6/	Send.c	IPv6SendFragments(0)	Function head
19	Tcpip6/ip6/	Send.c	IPv6SendFragments(1)	Function end
20	Tcpip6/ip6/	Lan.c	LanReceive(0)	Function head
21	Tcpip6/ip6/	Lan.c	LanReceive(1)	Function end
22	Tcpip6/ip6/	Subr.c	ChecksumPacket(0)	Function head
23	Tcpip6/ip6/	Subr.c	ChecksumPacket(1)	Function end

Table 4a Summary of instrumentation location numbers

Chapter Summary:

In this chapter, we discussed our instrumentation model from design to implementation, as well as the considerations of the approach. The key components of this model include time-stamping using the Pentium CPU tick counter and outputting timestamps from kernel mode by utilizing the NT WinDbg kernel debugging mechanism. Our experiments have proved that this instrumentation model works very well with the MSR IPv6 stack in the NT4 environment. As any source code-based instrumentation, its effectiveness also relies on the correctness of positioning the probing points within the stack source code. This requires intensive study and in-depth understanding of the stack's implementation.

Chapter 5 Instrumentation Results Analysis

In this chapter, we present some of our instrumentation measurement results as well as the corresponding analyses. All data are collected from the tests run on our testbed which consists of two 200MHz Pentium machine interconnected with a 100 Mbps Ethernet. The testbed's subnet is isolated to avoid unwanted traffic.

5.1 Latency for the whole stack IPv6

Latency for the whole stack refers to the time period for the data to pass through the whole processing path through the TCP/IPv6 or UDP/IPv6 layers in the stack. They are categorized into start latency and stop latency (see section 2.1). For start latency, we measured the time between (1) when the nearest TCPDispatchInternalDevice function (location 2) was called, and (2) when the first IPv6Send0 (location 12) which is for payload data (not for handshaking IP datagrams or acknowledgements) was called. For stop latency, we measured the time between (1) when the nearest TCPDispatchInternalDevice function (location 2) was called, and (2) when the Last IPv6Send0 (location 13) was finished for payload data. Stop latency may include some of the round trip transmission time if the payload size exceeds the Ethernet's MTU multiplied with TCP sliding window size. The latency for the whole stack is an indication of the protocol stack's efficiency in terms of the elapsed time for a given payload data size on a network physical characteristics. It can be used to compare the performance with a different protocol stack, e.g. IPv4. It can also be used to determine the characteristics of the measured stack in terms of average throughput and latency.

5.1.1 Latency for TCP/IPv6 of varying packet size

The first set of experiments was carried out to study the latency of the whole stack with TCP over IPv6. The test application that drives the sending and receiving of payload data is the public available utility TTCP ported to IPv6 by MSR. All the tests were run as point-to-point transmission with unicast destination address specified as `ttcp6.exe`'s command line argument.

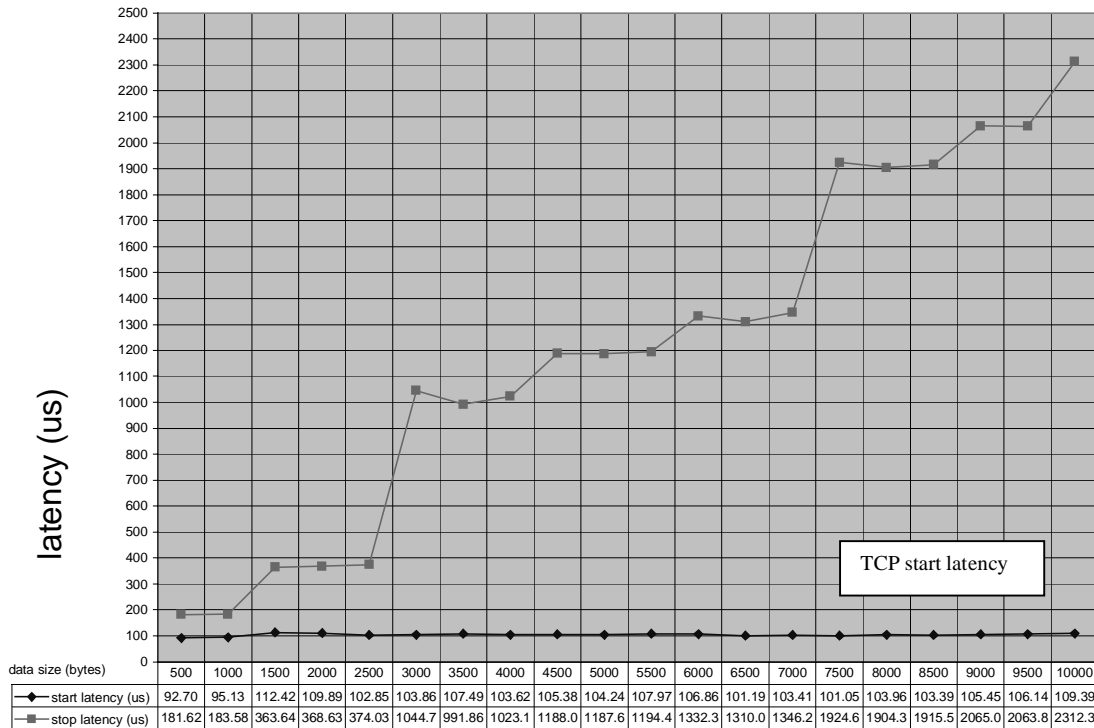


Figure 5.1 TCP/IPv6 start latency and stop latency through the whole stack, payload buffer size starts from 500 bytes to 10,000 bytes in steps of 500 bytes.

Figure 5.1 presents the results compiled when latency is measured while varying the payload buffer size from 500 bytes to 10,000 bytes. In TCP transactions, the start latency, which is the processing time in the stack before the first byte of payload data is sent, remain quite stable at around 100 micro-second as payload size was increased from 500 bytes to 10,000 bytes. On the other hand, the stop latency increases significantly with payload size. Whenever the payload size reaches a multiple of the adjusted path MTU, which is 1420 bytes ($= \text{path MTU} - \text{IPv4 header size} - \text{IPv6 header size} - \text{TCP header size} = 1500 - 20 - 40 - 20 = 1420$ bytes), the stop latency increases in a stepwise pattern due to the processing overhead associated with multiple fragments. The overall trend line of the stop latency is linear.

Figure 5.2 presents a chart describing the measured start and stop latency with more intensive measurements for TCP over IPv6. Payload data ranging from 100 bytes to 20,000 bytes, with even increments

of 100 bytes. Again the same behavior was observed as in the chart of Figure 5.1. The start latency of TCP remain around 100 microseconds while the stop latency increases linearly as follows:

$$y = 192.57x + 246.98$$

while x is the number of 1000 bytes in payload data, y is the expected stop latency in micro-seconds.

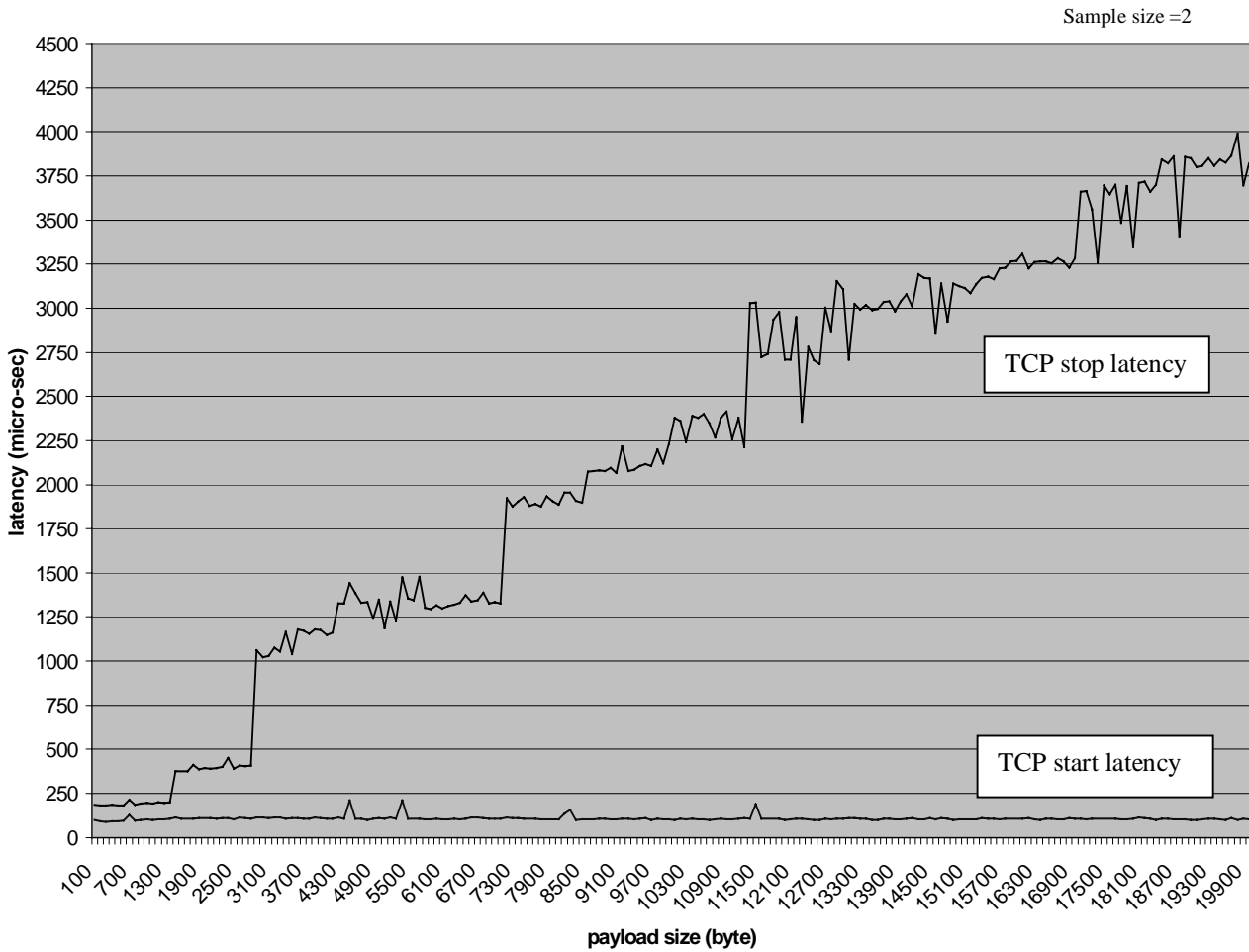


Figure 5.2 TCP/IPv6 start latency and stop latency through the whole stack; payload buffer size from 100 bytes to 20,000 bytes in steps of 100 bytes.

5.1.2 Latency for UDP/IPv6 of varying packet size

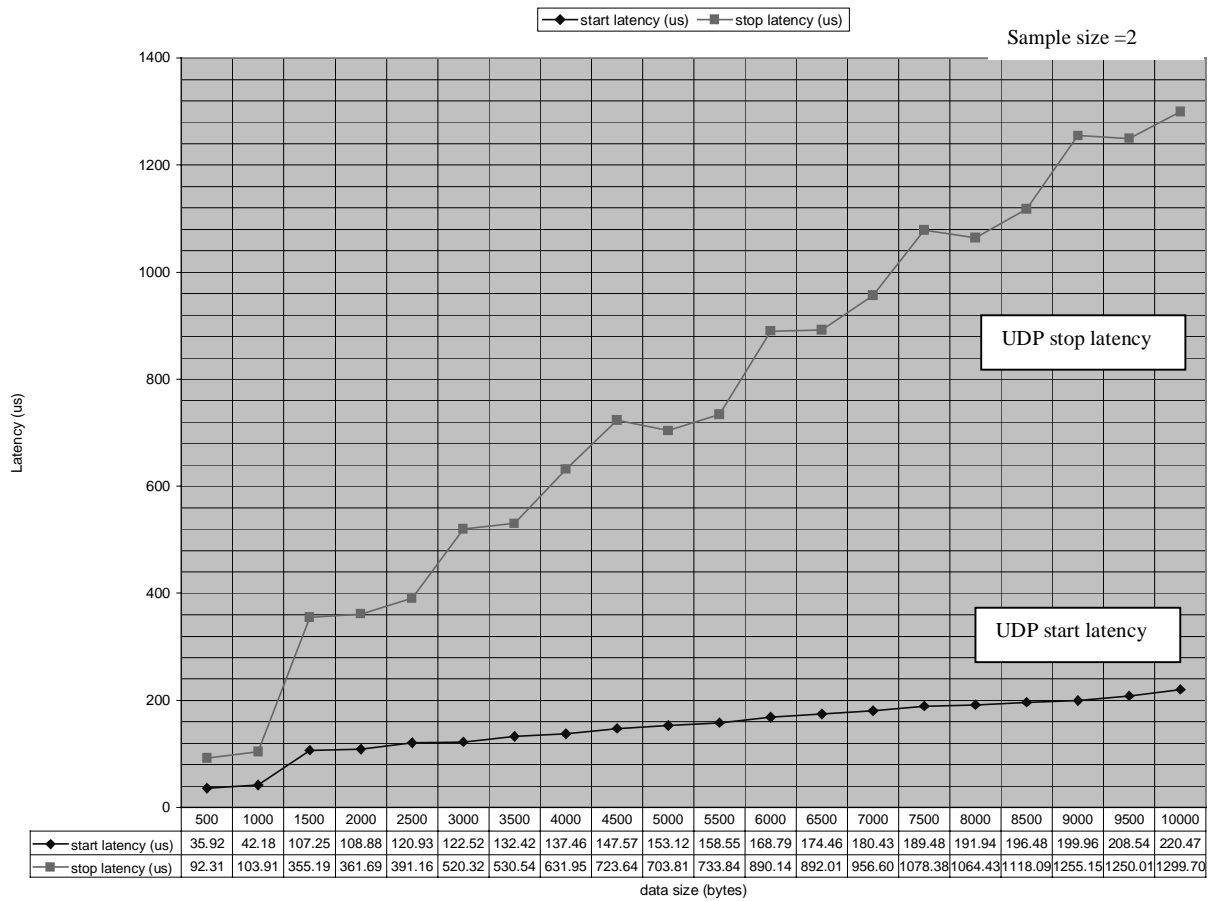
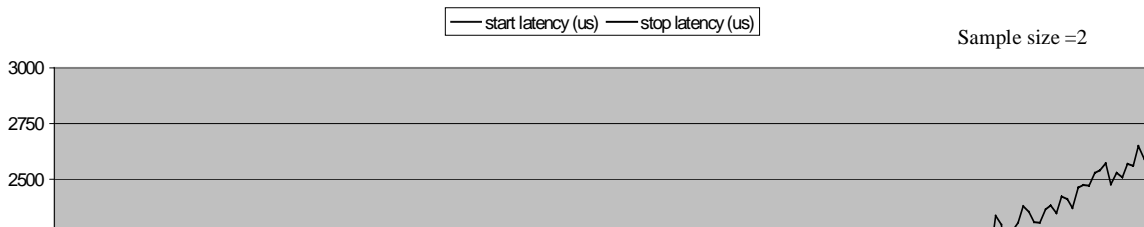


Figure 5.3 UDP/IPv6 start latency and stop latency through the whole stack, payload buffer size from 500 bytes to 10,000 bytes in steps of 500 bytes.

The same experiment reported in the previous section was repeated for the UDP over IPv6 protocol. Figure 5.3 presents the measurement results in a chart. In the UDP transactions over IPv6, the start latency increases slightly with the increase of the payload size. It starts at 36 microseconds when payload is 500 bytes, and reaches 220 microseconds for the payload of 10,000 bytes. This is unlike the start latency of TCP, which stays around 100 microseconds. The stop latency behaves roughly the same as that of TCP. It increases with payload size, but at a slope smaller than that of TCP. For example, the UDP stop latency for payload size of 10,000 bytes is around 1,300 microseconds, while TCP is about 2,300 microseconds for the same payload size.



UDP stop latency

UDP start latency

Figure 5.4 UDP/IPv6 start latency and stop latency through the whole stack; payload buffer size from 100 bytes up to 20,000 bytes in steps of 100 bytes

From the chart in Figure 5.4 we can clearly see the linear behavior for both the start and stop latency of UDP over IPv6. The trend line for UDP start latency can be formulated as $y_1 = 12.877x_1 + 84.154$ while x_1 is the number of 1,000 bytes in payload size, and y_1 is the expected start latency in microseconds. Likewise, the trend line for UDP stop latency is formulated as $y_2 = 128.54x_2 + 49.114$ while x_2 is the number of 1,000 bytes in payload size, and y_2 is the expected stop latency in microseconds.

5.1.3 Comparison between TCP and UDP over IPv6

TCP and UDP are the two major transport protocols running on top of IPv6, their comparative performance on our testbed, as described in the previous two sections, are summarized in the table below:

	Start Latency Trend Line	Stop Latency Trend Line
TCP over IPv6	$y = 0.243x + 109.2$	$y = 192.57x + 246.98$
UDP over IPv6	$y = 12.877x + 84.157$	$y = 128.54x + 49.117$

Remark: x = number of 1000 bytes in payload data; y = expected latency in microseconds

Table 5a Summary of trend-line formulas of TCP and UDP; start and stop latency through the whole stack

Theoretically, the start latency with a zero payload size is the minimum processing time needed by the stack, which is 109µs for TCP/IPv6 and 84 µs for UDP/IPv6 according to the above formulas. While the payload size increases, the start latency of UDP increases faster than that of TCP, and exceeds TCP’s start latency at the payload size of 1982 bytes. We can expect a 12.877-microsecond increase in start latency for every thousand bytes of additional payload data in UDP, while the start latency of TCP remains quite stable around 100 microseconds with little increments. For the stop latency, the expecting increase for every thousand bytes of additional payload is 192.57µs with TCP, and 128.54µs with UDP.

Comparison of TCP and UDP latencies

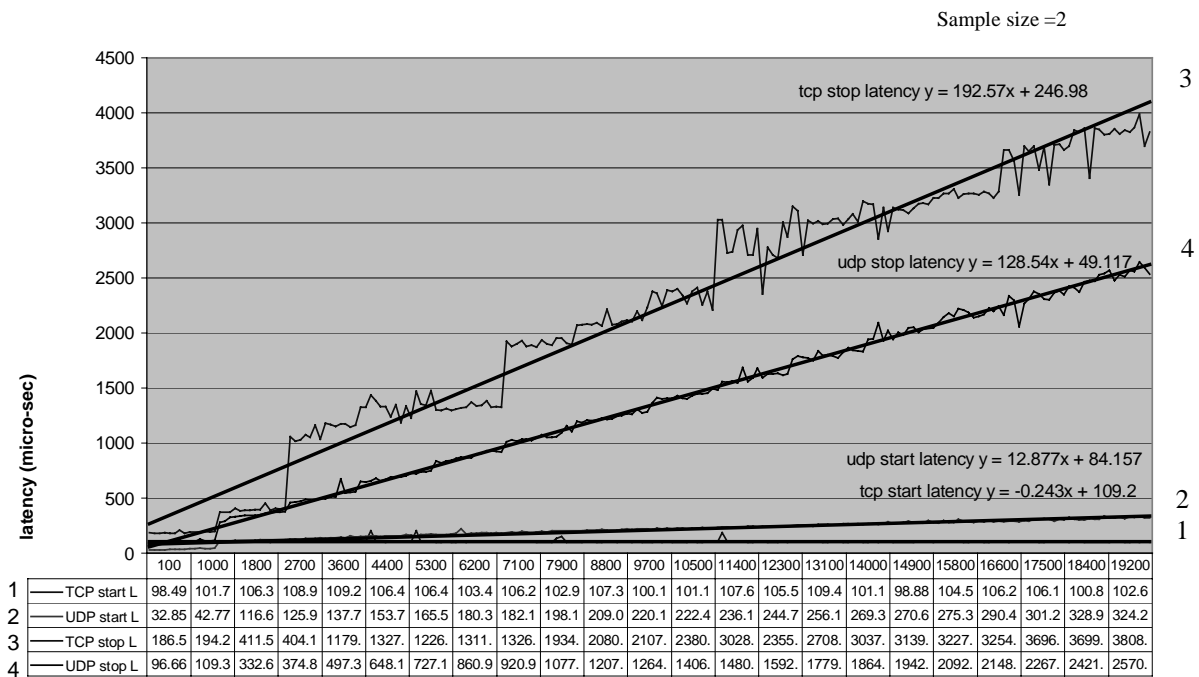


Figure 5.5 Comparison of TCP/UDP for start/stop latency through the whole stack

Generally, UDP over IPv6 consumes more time on start latency than TCP. One reason for this is that in UDP there is a payload fragmentation involved in the IP layer, whereas TCP segments its payload within the TCP layer using the path MTU (Maximum Transfer Unit, maximum amount of data that is allowed to send on the link layer path in a packet) as the segmentation size. We observed that the function IPv6SendFragments() in tcpip6/ip6/send.c module is called in UDP when the payload size is greater than path MTU, while this same function is never called in TCP with any payload size.

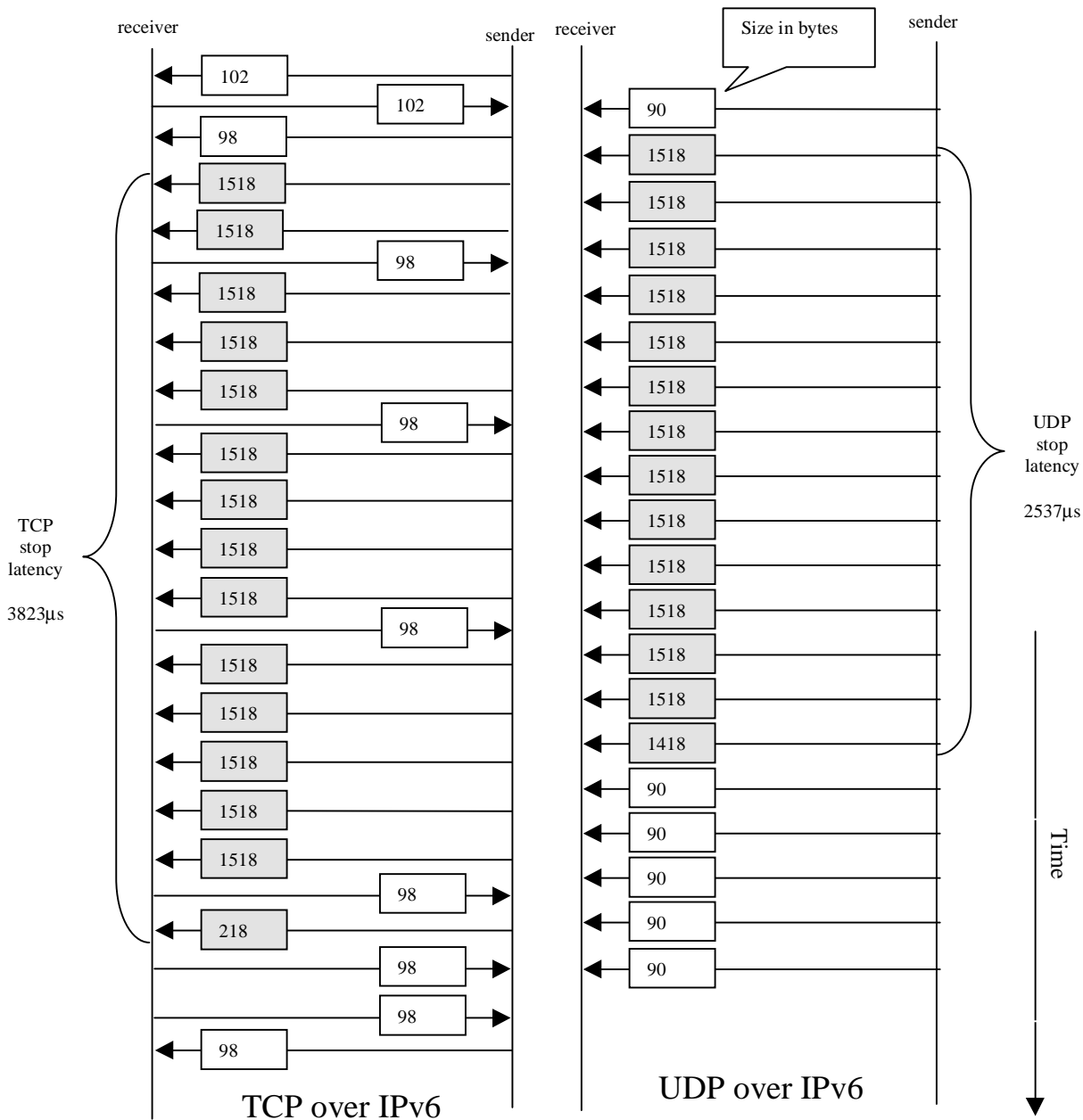


Figure 5.6 Message diagrams for both TCP and UDP of sending payload size of 20,000 bytes

On the other hand, UDP's stop latency is about one-third smaller than that of TCP. This is mainly because UDP uses a connectionless transmission scheme while TCP uses a connection-oriented one with backward acknowledgements. In UDP measurements, we observed no backward acknowledgement datagram. Figure 5.6 is the message diagrams recorded by NetXray during our instrumentation tests for the payload size of 20,000 bytes sent by both TCP and UDP over IPv6. In Figure 5.6, the sender using TCP/IPv6 protocols sends out 18 frames (15 of them are carrying payload data) and receives 7 acknowledgement frames for sending a 20,000-byte payload buffer. The same sender using UDP/IPv6 sends out 20 frames (14 of them are carrying data) and doesn't need to wait for any acknowledgement frames. The stop latency for UDP with 20,000 bytes payload is 2537 μ s which is only about 66% of that of TCP (3823 μ s). The same conclusion (stop latency of UDP is about 2/3 of that of TCP) can be found in the slopes of the trend lines for stop latency of TCP and UDP listed in Figure 5.5 above.

5.2 Latency for the interested parts

Our instrumentation model can be used to measure the processing latency for some specific parts within the protocol stack. The objective is to obtain a breakdown of the latency in different categories of functionality of the stack. In the following sections, some of our measurement results of the latency for the data checksumming and NDIS buffer management operations, are presented and analyzed.

5.2.1 checksumming for TCP/IPv6 data

Unlike in IPv4, checksumming in IPv6 is only performed on TCP and UDP data part, and no longer on the IP header part. In the implementation of the MSR IPv6 stack, function ChecksumPacket() located in the "tcpip6/ip6/subr.c" module is called to calculate the checksum whenever a packet is received or is to be sent.

This function is called in all the core sending and receiving routines for both TCP and UDP such as TCPSend(), UDPSend(), TCPReceive(), UDPReceive(), ICMPv6Send(), ICMPv6Receive(). Within the function ChecksumPacket(), it loops to call the assembler version of NT common checksum calculating routine, Cksum() or the same as tcpsum() located in the common/i386/xsum.asm module, to do the actual checksumming computations.

The total checksum processing latency is the cumulative time it takes to compute checksum for all the packets including payload and handshaking packets with the function ChecksumPacket(). The payload checksum latency is the part of the total checksum latency that is only for the packets with payload data. Figure 5.7 presents the measurement of these two checksum latencies for TCP transactions over IPv6.

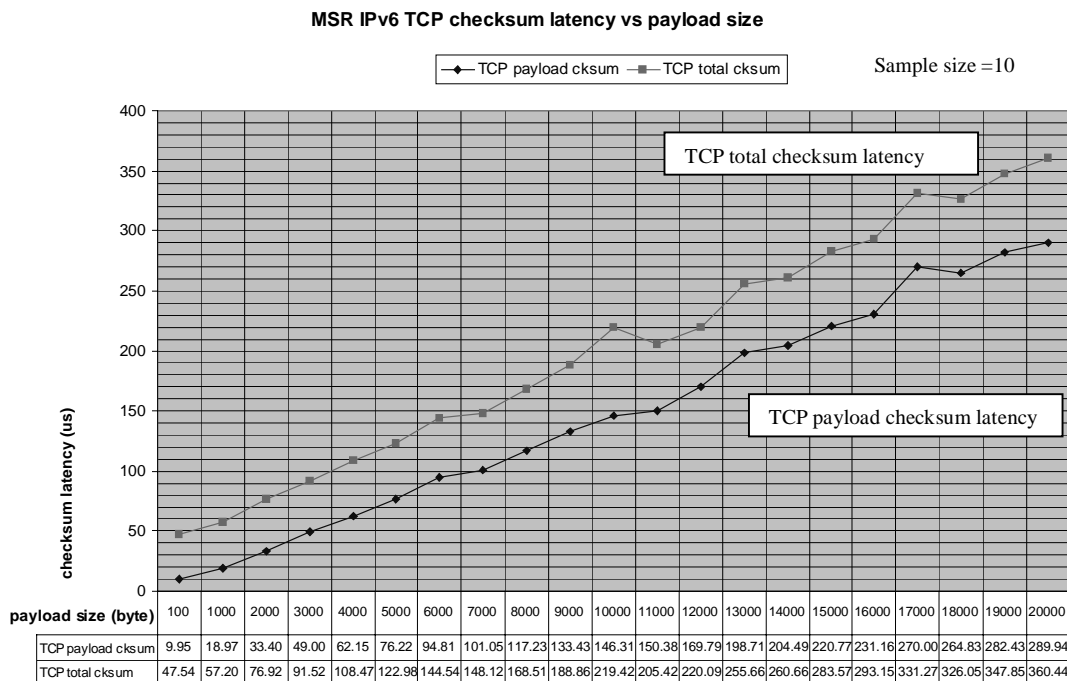


Figure 5.7 MSR TCP/IPv6 Checksuming latency versus payload size from 1k to 20k step 1k bytes

In MSR IPv6 stack, each TCP/IPv6 datagram is checksummed separately. In figure 5.7, The total/payload checksum latency increase linearly with the payload size. The payload checksum and the total checksum latency start at approximately 10 μ s and 48 μ s for a 100-byte payload, finally reach about 190 μ s and 360 μ s for a 20,000-byte payload data. Their trend lines can be formulated as:

$$y_1 = 14.494x_1 - 10.622 \text{ (for payload checksum latency in TCP)}$$

$$y_2 = 15.888x_2 + 28 \quad (\text{for total checksum latency in TCP})$$

where x is the value of payload in thousand-byte, and y is the checksum latency in microseconds. According to the formula, each additional 1000-byte payload data will add 15.8 μ s of total checksuming time to the stack's processing overhead.

When we compared the payload/total checksum latency with the stop latency of the same payload size, we found the percentages of checksum latency to stop latency are mostly less than 10%. They jump to the highest of 9.8% for payload checksum and 29% for total checksum when the payload size is around the adjusted path MTU (1420 bytes), and become stable around 7% and 9% while payload size increases (see Figure 5.9).

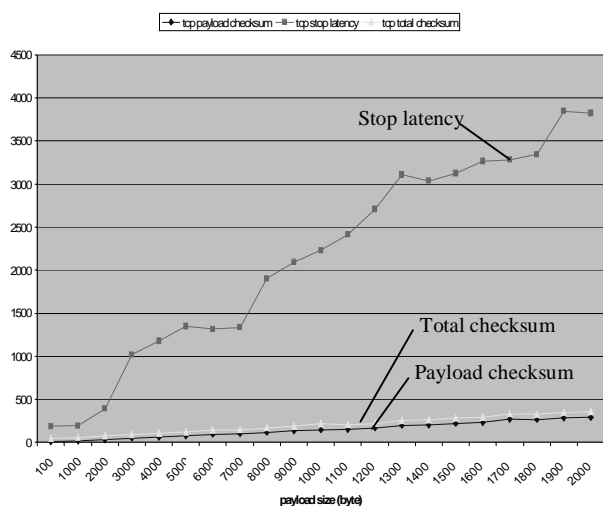


Figure 5.8 The payload/total checksum latency and the stop latency in TCP/IPv6

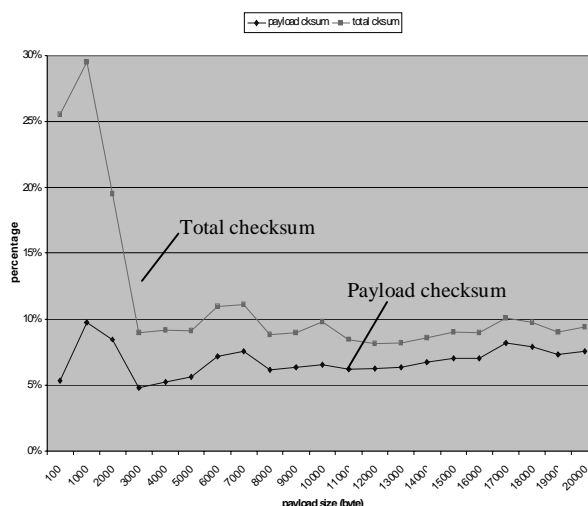


Figure 5.9 Percentage of the payload/total checksum latency to the stop latency in TCP/IPv6

Figure 5.8 compares payload/total checksum latency and stop latency. Figure 5.9 shows the percentage of those latencies relative to stop latency in TCP/IPv6. Previous studies on the TCP/IP protocol stack on the DEC Ultrix 4.2a platform have reported that checksum processing alone consume nearly 50% of the total stack processing time for large messages [5]. By comparison we find that on average in TCP the payload checksum processing overhead is roughly 7% of the stop latency. Total checksum processing is found to be 11.5% of the stop latency. This is when measuring only small and medium size payloads (ranging from 100 bytes to 20,000 bytes). One issue that may explain this relatively small percentage

is that by definition the stop latency includes part of the round trip time on the network. Thus, while we used an isolated, quiet 100Mbps fast Ethernet to minimize the round trip time, the stop latency still includes time for protocol processing plus an additional time corresponding to the round trip period from sending to acknowledging.

Generally speaking, the total checksum processing in TCP contributes about 11.5% of the stop latency.

5.2.2 Check-summing for UDP/IPv6 data

In the UDP transactions, ChecksumPacket() is called only once for each UDP datagram regardless of the payload data size. We employed the same measurement as described in the previous section for UDP over IPv6 and charted the results in Figure 5.10. The payload/total checksum processing latencies we measured shown in Figure 5.10 increase linearly with payload size. The payload checksum and the total checksum latency start at 6.5µs and 35µs for a 100-byte payload buffer, and reach 241µs and 272.5µs for a 20,000-byte payload. Their trend lines can be formulated as:

$$y_1 = 11.428x_1 + 27.756 \quad (\text{for total checksum latency in UDP})$$

$$y_2 = 11.383x_2 - 0.8453 \quad (\text{for payload checksum latency in UDP})$$

where x is the payload size in thousand-byte, y is the checksum latency in microseconds.

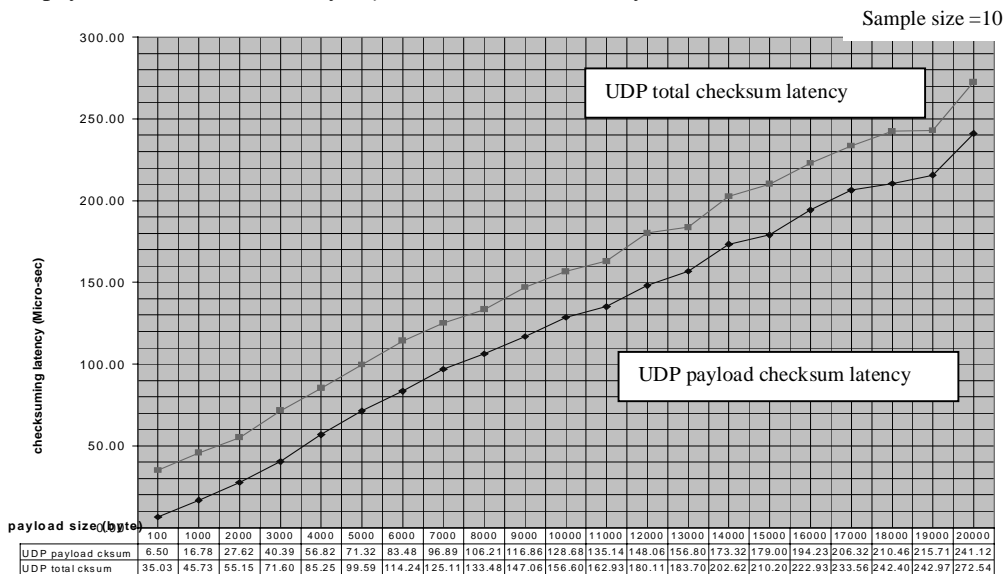
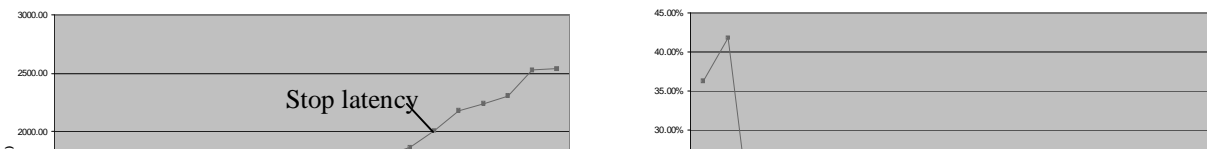


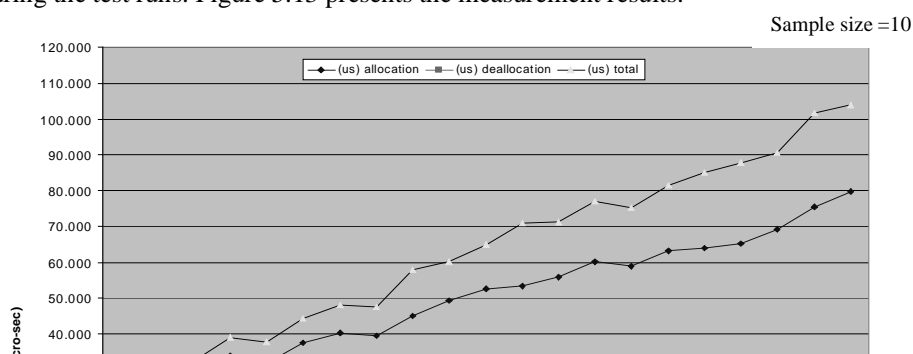
Figure 5.10 Checksum latency of MSR UDP/IPv6 versus payload size



According to the formula of the total checksum latency's trend line, for each additional 1000-byte payload data, there is a 11.4 μ s processing time added to the total checksum overhead, which is about 28% smaller than that of TCP (15.8 μ s per 1000-byte payload). Generally, checksum processing for data sent with UDP consumes a larger percentage of the stop latency time than that with TCP. The average total checksum processing time is 14.66% of the stop latency. It reaches the peak value of 41.82% when the payload size is around the adjusted path MTU (1420 bytes), and stays around 10% of the stop latency while payload grows. The average payload checksum processing time is 9.5% of the stop latency. It peaks at 15.34% for the adjusted-MTU payload size and remains stable at roughly 9% of the stop latency while the payload size increases. Generally speaking, the total checksum processing in UDP contributes about 15% of the stop latency.

5.2.3 Buffer allocation and de-allocation in TCP/IPv6

Buffer management is another major source of the latency in the protocol stack. There are many places in the MSR IPv6 implementation's source code which involves the NDIS buffer management API calls to allocate and de-allocate buffer memory. These API calls include: NdisAllocatePacket(), NdisAllocatePacketPool(), NdisAllocateBuffer(), NdisAllocateBufferPool(), ExAllocatePool(), NdisFreePacket(), NdisFreePacketPool(), NdisFreeBuffer(), NdisFreeBufferPool(), ExFreePool(). We placed instrumentation code at these NDIS buffer management calls, measured and collected the time for these operations during the test runs. Figure 5.13 presents the measurement results.



According to our finding, buffer allocation operations generally consumes more processing time than the buffer de-allocation operations. 71.6% of the total buffer management overhead comes from buffer allocation, while the rest of 28.4% comes from buffer de-allocation. They both increase linearly with payload size. The total buffer management overhead starts at 16.8μs with a 100-byte payload size, and reaches 103.8μs with a 20,000-byte payload size. The formulas of the trend lines are shown in Table 5b.

	Formula for linear trend line	Percentage
Buffer allocation operations	$y_1 = 2.9118x_1 + 17.03$	71.6%
Buffer de-allocation operations	$y_2 = 1.1567x_2 + 0.6727$	28.4%
Total buffer management overhead	$y = 4.0685x + 17.703$	100%

Table 5b Formulas for buffer management latency trend lines in TCP/IPv6

In Table 5b, x , x_1 or x_2 is the number of payload size in thousand bytes; y , y_1 or y_2 is the latency in microseconds. The formulas show that we can expect a 4-μs increase to the total buffer management overhead for TCP with every additional 1000-byte payload data.

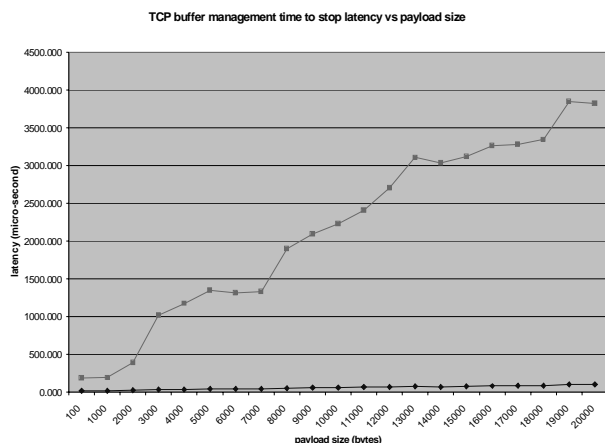


Figure 5.14 TCP: Buffer management time to stop latency

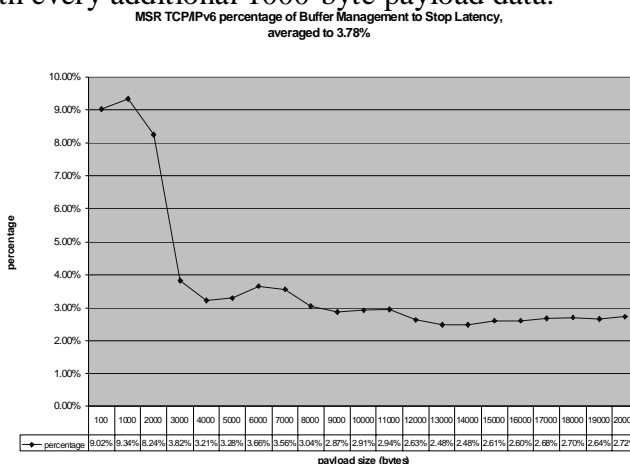


Figure 5.15 TCP: percent of buffer management time to stop latency

When compare to the stop latency of TCP, buffer management operation overhead is relatively small. The percentage for buffer management peaks at 9.34% when payload size is around the adjusted path MTU of Ethernet (1420 bytes), and stays at around 3% thereafter. It averages 3.78% of the total stop latency of TCP.

5.2.4 Buffer allocation and de-allocation in UDP/IPv6

Buffer management operations in UDP transactions consumes a larger percentage of the total stop latency than that of TCP. On average, the total buffer management consumes 11.68% of the UDP stop latency time. Similar to TCP, buffer allocation operation creates more overheads than that of the de-allocation operations. 60.9% of the UDP’s buffer management overhead comes from buffer allocation, the rest of the 39.1% comes from de-allocation. Unlike TCP, UDP’s buffer latency distribution pattern is non-linear. Instead, it is distributed as the payload size increases.

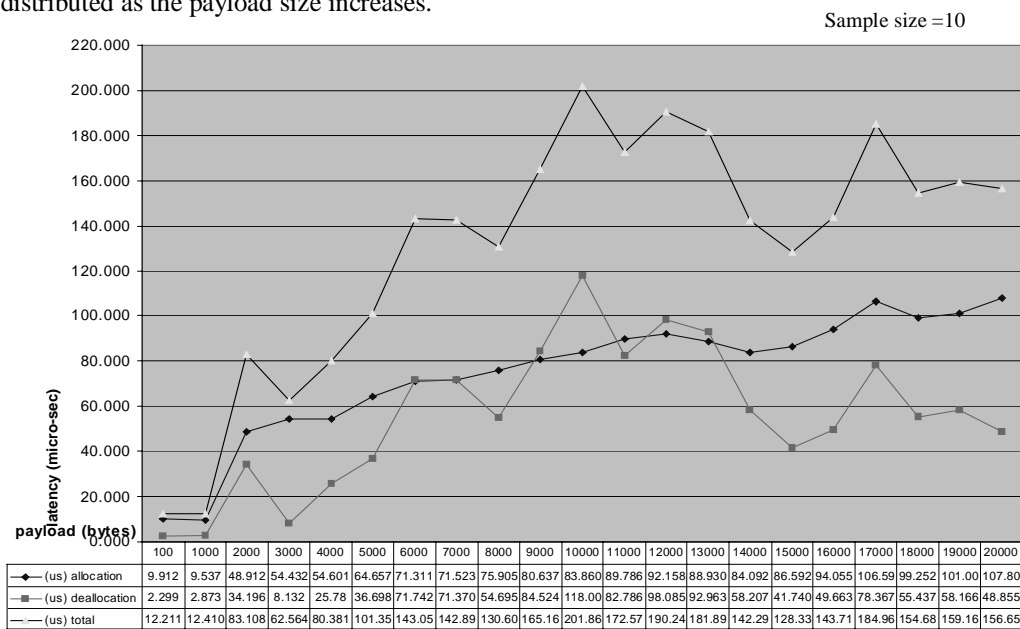
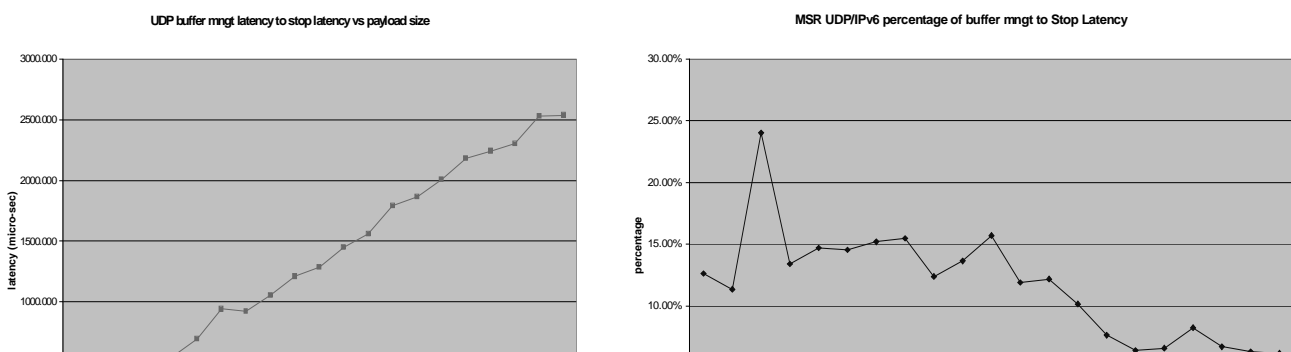


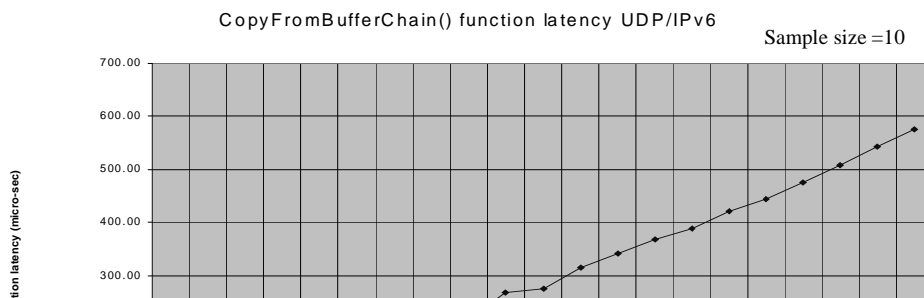
Figure 5.16 MSR UDP/IPv6 buffer management latency versus payload size from 1k to 20k step 1k



The buffer allocation overhead in UDP increases when the payload size increases. But the buffer de-allocation overhead doesn't follow this rule, it shows a discrete pattern when payload size increases. We observed in the instrumentation data that the number of de-allocation function calls increase when payload size increases, but the accumulated time they consumed is not linear. The de-allocation time starts at 2.3 μ s for a 100-byte payload, jumps up to the highest of 118 μ s when payload size is 10,000 bytes, and slides down to 48.8 μ s for a 20,000-byte payload. The total buffer management time averages 130.96 μ s for the payload ranging from 100 bytes up to 20,000 bytes, and it contributes 11.68% to the stop latency in UDP on average.

5.2.5 Data moving operation in UDP/IPv6

Data moving operation is defined as the operations of the stack that actually copy the packet data from one buffer to another. This is needed for fragmenting UDP datagrams or TCP segments in IP layer. We observed a quite large portion (>60%) of the stack's CPU utilization was spent on a data moving function CopyFromBufferChain() during the UDP transmission. (See section 5.4.2 CPU utilization for UDP over IPv6). This function is called in IPv6SendFragments() to copy data from a buffer chain to a "flat buffer" when the UDP datagram or TCP segment is greater than the path MTU. We built an instrumented version of the stack and measured the processing time for the function CopyfromBufferChain() with payload size ranging from 100 bytes to 20,000 bytes. Figure 5.19 shows the measurement results.



The processing latency of the data-moving function CopyFromBufferChain() is quite linear in UDP, and it can be formulated as:

$$y = 28.775x - 36.468$$

where x is the number of payload data in thousand bytes, and y is the latency in microseconds. When the payload size is less than path MTU, the measured latency is zero because of no fragmentation. When payload is 2,000 bytes it takes 50.55 μ s for this function to finish copying. When the payload size is 20,000 bytes, the processing latency reaches 575 μ s. This is a relatively long latency since the whole UDP stop latency for the same amount of payload is only 2537 μ s which is about 5 times as it. This data-moving function is a relatively short function (with 27 lines of C code, located in “tcpip6/ip6/subr.c” module). It could be a potential bottleneck in the implementation that would slow down the whole processing if it’s not fully optimized. Figure 5.20 presents the comparison between the data-moving latency and the stop latency of UDP. Figure 5.21 shows the percentage of the data-moving latency to the total stop latency with varying payload size.

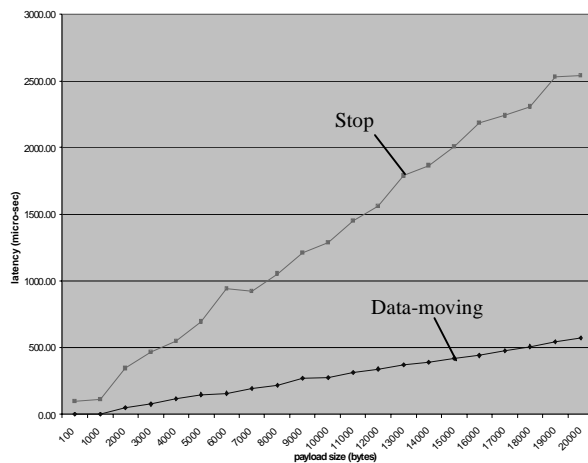


Figure 5.20 UDP data move latency to stop latency

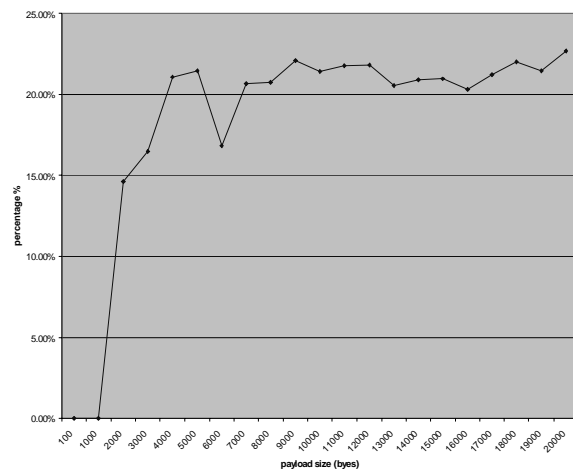


Figure 5.21 UDP/IPv6:percent of data move latency to stop latency

The data move function `CopyFromBufferChain()` in UDP generally takes about 18.5% of the UDP stop latency on average. It starts at 14% of the UDP stop latency when payload size is 2,000 bytes, and stays at about 22% of the stop latency when payload is equal or greater than 4,000 bytes. With such a relatively high percentage, we believe that the stop latency of UDP could be reduced significantly if this function is removed or fully optimized.

5.3 Payload throughput for the whole stack IPv6

Payload throughput is defined as the amount of payload data that is transmitted through the whole stack per time unit. The payload throughput is calculated using the formula $T = P/L$ where T is the payload throughput, P is the size of the payload data in bytes or kilobytes, and L is the stop latency for that payload size.

5.3.1 Payload throughput for TCP/IPv6 versus packet size

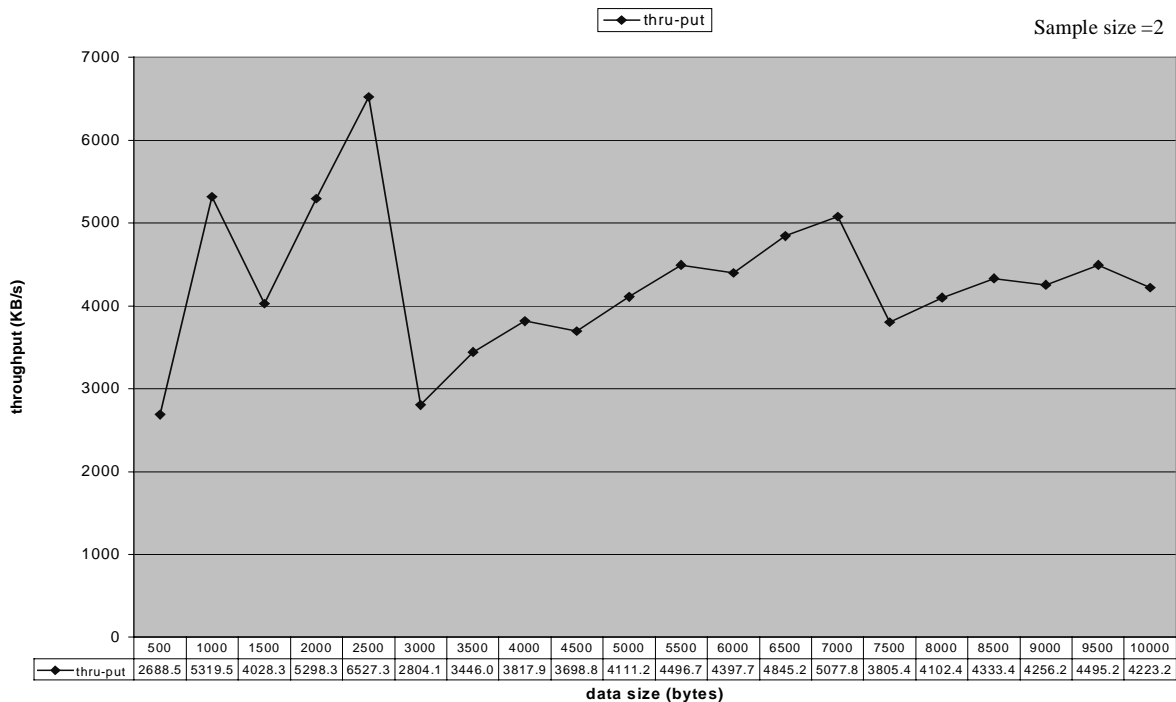


Figure 5.22 TCP/IPv6 payload throughput versus payload buffer size, from 500 to 10K step 500 bytes

Figure 5.22 shows the throughput of TCP/IPv6 for payload size ranging from 500 bytes to 10,000 bytes in step of 500 bytes. The spurs that can be observed occur whenever the payload size reaches the multiple of the adjusted MTU, when an additional fragment is required. The additional fragment results in a sharp increase in the stop latency, which inversely cause a sharp drop in the throughput.

Figure 5.23 provides a finer grain measurement of TCP/IPv6 throughput. While the payload size increases, the added round trip time of an extra fragment becomes less significant, and the throughput becomes more stable around the trend line, which can be formulated with a power equation as:

$$y = 2258.1x^{0.1475}$$

where x is the number of 100-byte in payload size (since the measurement is carried out at increments of 100 bytes), and y is the payload throughput in kilobyte per second.

Sample size =2

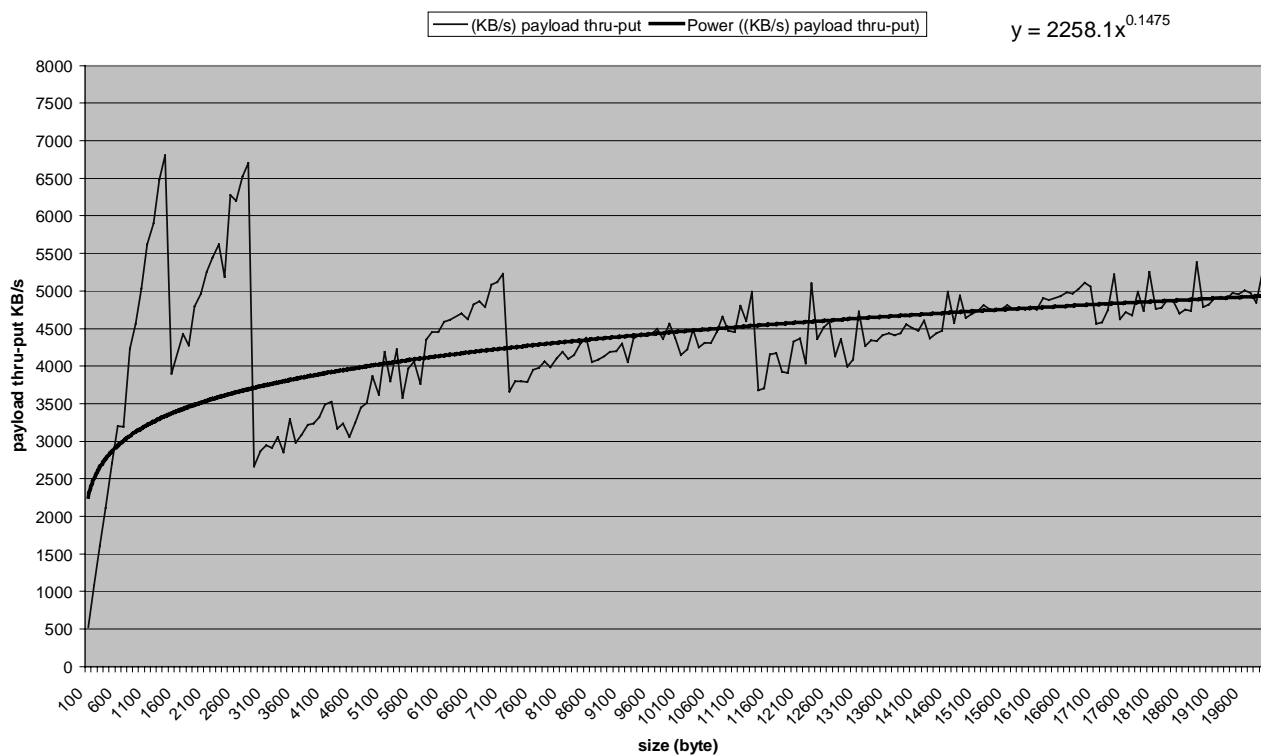


Figure 5.23 TCP/IPv6 payload throughput versus payload buffer size, from 100 to 20K step 100 bytes

The maximum TCP/IPv6 throughput reaches 6800 KB/s (53.125Mbps) when the payload size is 1400 bytes. The second highest throughput occurs at 6703KB/s (52.37Mbps) when the payload size of 2800 bytes. While payload size increases, the throughput stabilizes at roughly 5000 KB/s (39.1Mbps). This is the throughput within the TCP/IPv6 protocol stack for payload data solely, excluding any acknowledgements and handshake frames. The payload throughput is approximately 40% of the maximum bandwidth of the testbed network, which is 100Mbps.

5.3.2 Payload throughput for UDP/IPv6 versus packet size

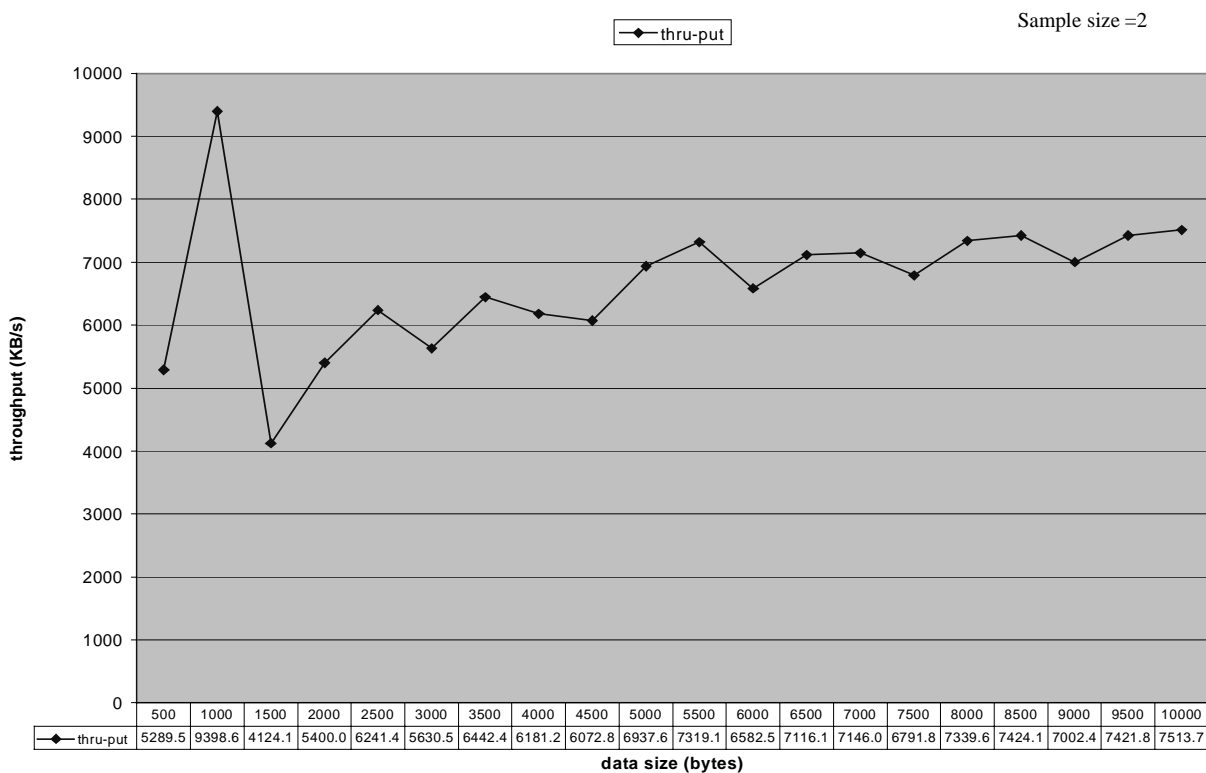


Figure 5.24 UDP/IPv6 payload throughput versus packet size from 500 to 10,000 bytes step 500 bytes

We observed a very similar behavior in the payload throughput with UDP as that in TCP. The throughput fluctuates for the first few multiples of the MTU, and gradually stabilizes as payload size increases. UDP generally achieves a higher throughput than TCP because of its advantages of using connectionless transmission with no acknowledgement required. Figure 5.25 shows a finer measurement of UDP throughput with payload size ranging from 100 bytes up to 20,000 bytes in step of 100 bytes.

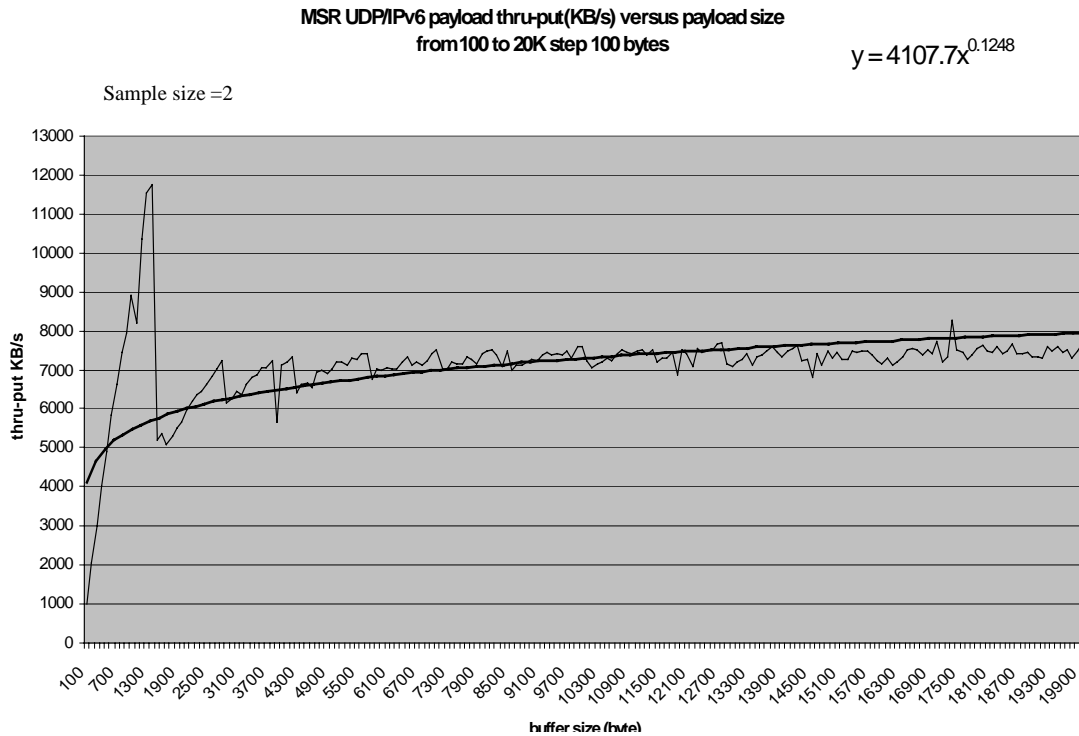


Figure 5.25 MSR UDP/IPv6 payload throughput versus payload size from 100 to 20k step 100 bytes

The maximum of UDP/IPv6 payload throughput reaches 11736.5KB/s (91.7Mbps) when payload size is 1400 bytes. The general trend line of throughput can be formulated as:

$$y = 4107.7 x^{0.1248}$$

where x is the number of 100-byte in payload (since the measurement is carried out at increments of payload size of 100 bytes), y is the latency in microseconds. While the payload size increases, the throughput number stabilized at about 8000 KB/s (62.5Mbps), which is 62.5% of the maximum bandwidth of our test bed network. We found a 60% of increase in the average throughput of UDP (62.5Mbps) compares to that of TCP (39.1Mbps).

5.4 CPU utilization

CPU utilization refers to the percentage of time distribution that the CPU had spent on different programs or processes during a measurement period. This metric provides a measurement of the CPU usage requirement (0-100%) for a given module or a process of a software. This definition can be further extended to the CPU usage requirement for the functions within a module. The measurement is obtained by sampling the currently running process in the operating system in a very short interval during a measurement period. When the sampling is finished, the number of the presenting times for each module or function that had been run during the measurement period are accumulated, and the CPU utilization percentage for a particular module or function can be calculated using the accumulated presenting number divided by the total sampling number.

For our test subject MSR IPv6, it would be interesting to find out the percentage of the CPU time spent on processing the stack (tcpip6.sys) among other kernel mode modules, or the percentage of the CPU time spent on different functions within the tcpip6.sys module. We used VTune 3.0, a Windows systems performance measurement tool made by Intel Inc., to measure the CPU utilization percentage during the TCP and UDP test runs. We used the ttcp6.exe utility that came with the MSR IPv6 source to trigger repeated sending of payload buffer of 20K bytes in TCP or UDP, and started the VTune measurements on the sending machine during the test process. We instructed VTune to take a sample for every 1

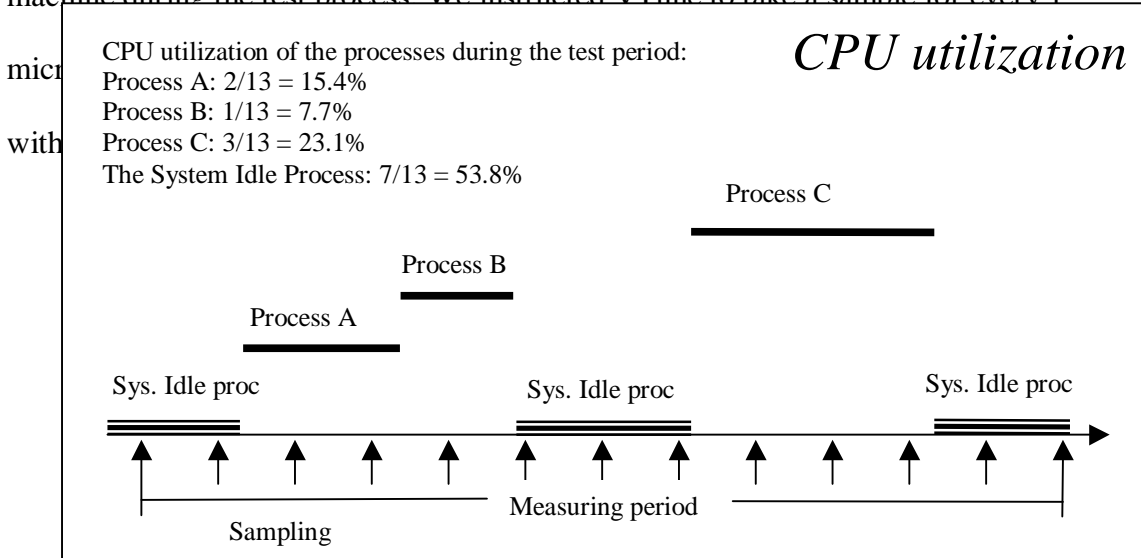


Figure 5.26 Illustrate the CPU utilization measurement

5.4.1. CPU utilization for TCP over Ipv6

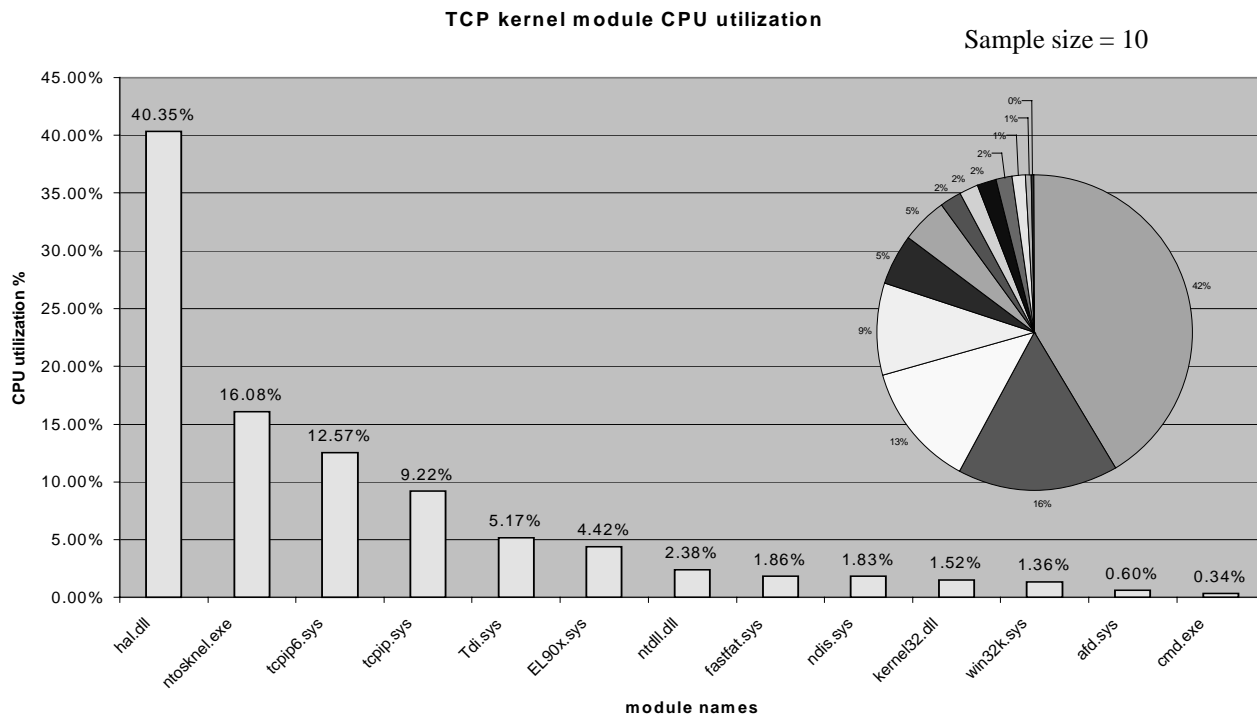


Figure 5.27 TCP/IPv6 kernel modules CPU utilization

Figure 5.27 presents the CPU utilization of the modules during a UDP test-sending period of 5 seconds. During the repeated sending of 20K data buffers, about 40% percent of the CPU time was spent on the system idle process (hal.dll). The Windows NT OS kernel (ntosknl.exe) module occupies about 16% of the CPU time. Only about 12.5% of the CPU time was spent on running the processes within the IPv6 stack module (tcpip6.sys). Since we are using a 6-over-4 mechanism, which embeds IPv6 datagram into IPv4's, to do the transmissions on the test bed, the CPU also spends some time (9.22%) on the TCP/IP version 4 stack module (tcpip.sys). The TDI driver (tdi.sys) and the 3COM Fast EtherLink NIC driver (el90x.sys) consume 5.17% and 4.42% of the CPU time respectively.

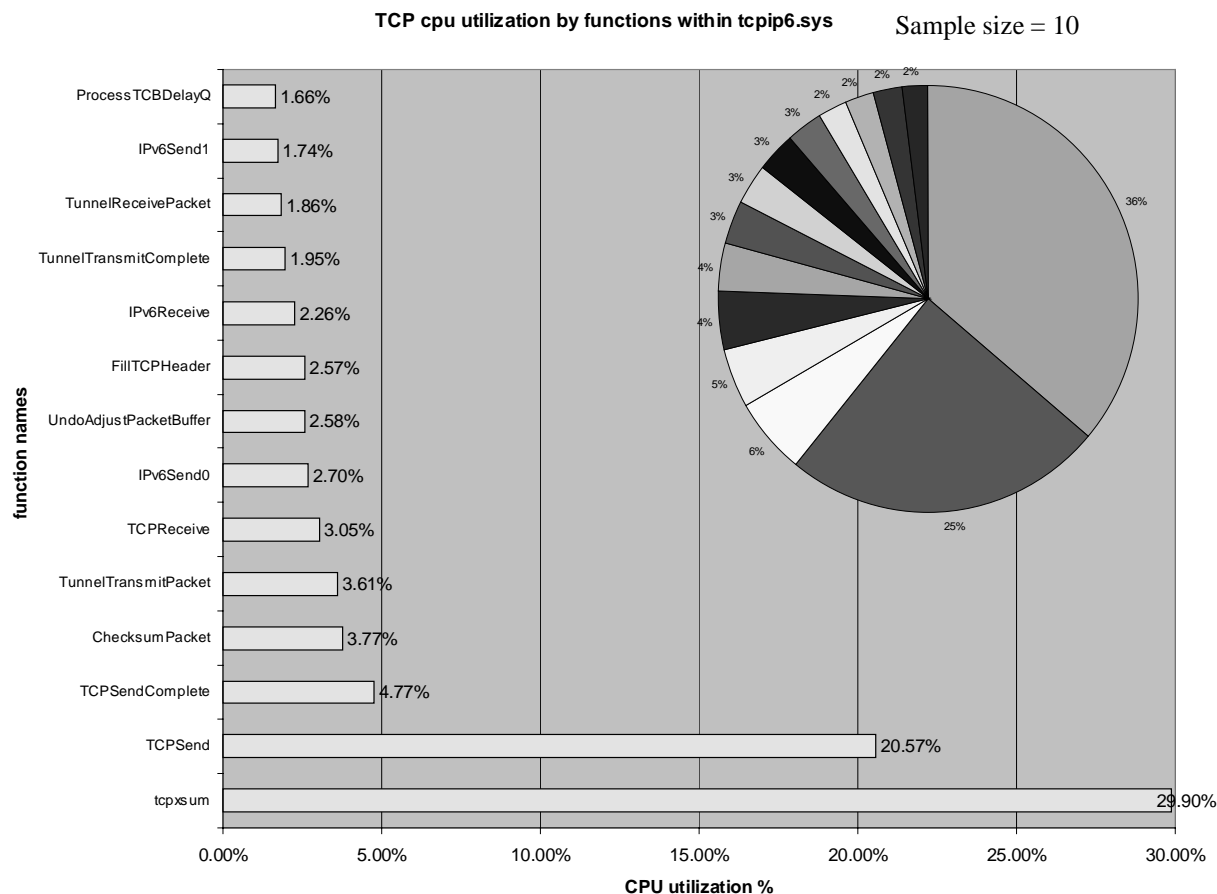


Figure 5.28 CPU utilization of individual functions within tcpip6.sys, TCP over IPv6

Figure 5.28 shows the breakdown of the CPU utilization of the individual functions within the IPv6 stack module tcpip6.sys. Within the 12.5% CPU time that spent on the stack for TCP sending, about 30% was spent on the function tcpsum(), which computes the checksum for both the outgoing and receiving packets. This function is called by ChecksumPacket() whose latency is measured and reported in section 5.2.1. About 21% of the stack's CPU utilization was spent on the function TCPSend(), which handles the TCP protocol specific sending functionality. About 5% was spent on calling the TCPSendComplete() function, which is called at the IP layer to complete a TCP session. The function ChecksumPacket(), which is a high level caller of the tcpsum() function for checksum computation, take 3.77% of the stack's CPU utilization. The IP layer sending functions IPv6Send1() and IPv6Send0() take 1.74% and 2.7% of the stack's CPU utilization. The link layer

level send functions TunnelTransmitPacket() and TunnelTransmitComplete() take 3.61% and 1.95% respectively.

5.4.2 CPU utilization for UDP over IPv6

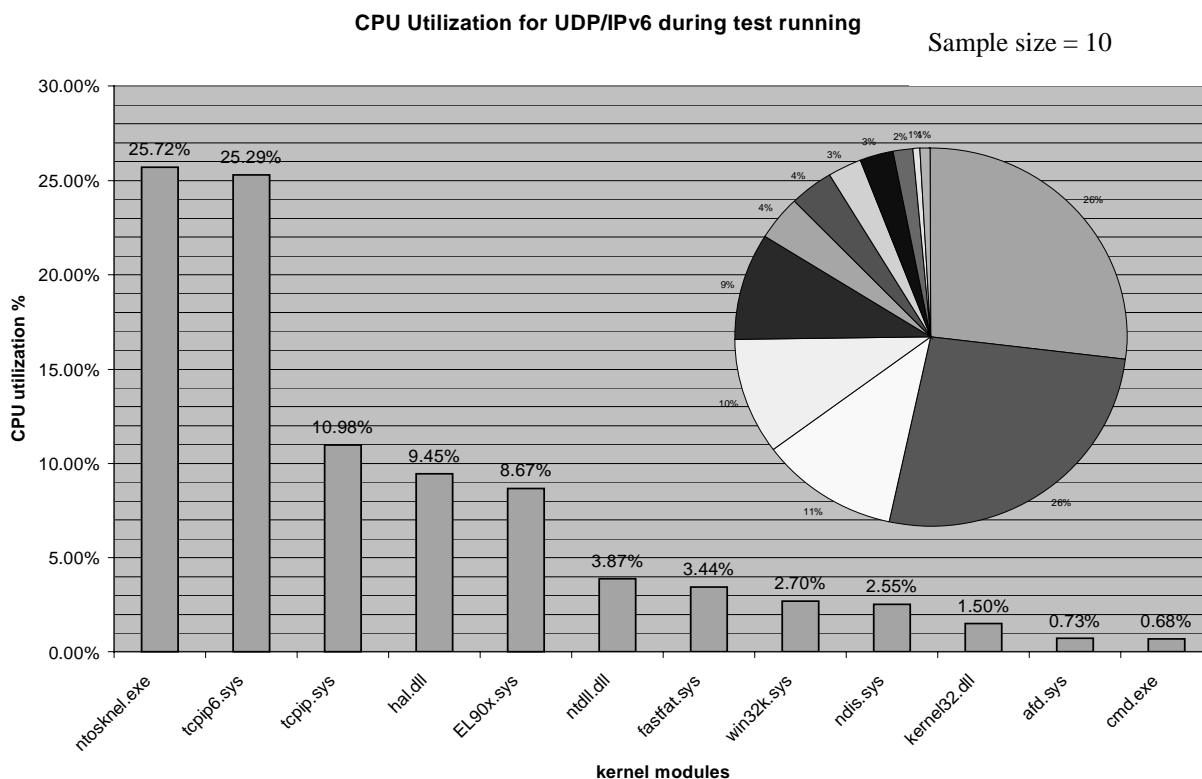


Figure 5.29 UDP/IPv6 kernel modules CPU utilization

In UDP over IPv6 sending, the CPU spends much less time on the system idle process in hal.dll (only 9.45%) as compared to TCP (40%), which means the CPU is much busier in UDP than in TCP during the repeated sending of 20K bytes buffer. This is because the CPU does not have to be suspended for any acknowledgements in UDP mode, thus the CPU is occupied more fully (higher CPU utilization percentage). The IPv6 stack's CPU utilization in UDP mode (25.29%) is almost double as that in TCP (12.57%). The Windows NT kernel module ntosknl.exe takes 25% of the CPU time, which is about the same as the time that the CPU spent on the IPv6 stack module. The version 4 TCP/IP stack module tcpip.sys and the 3COM NIC driver el90x.sys consume about 11% and 9% of the CPU time respectively.

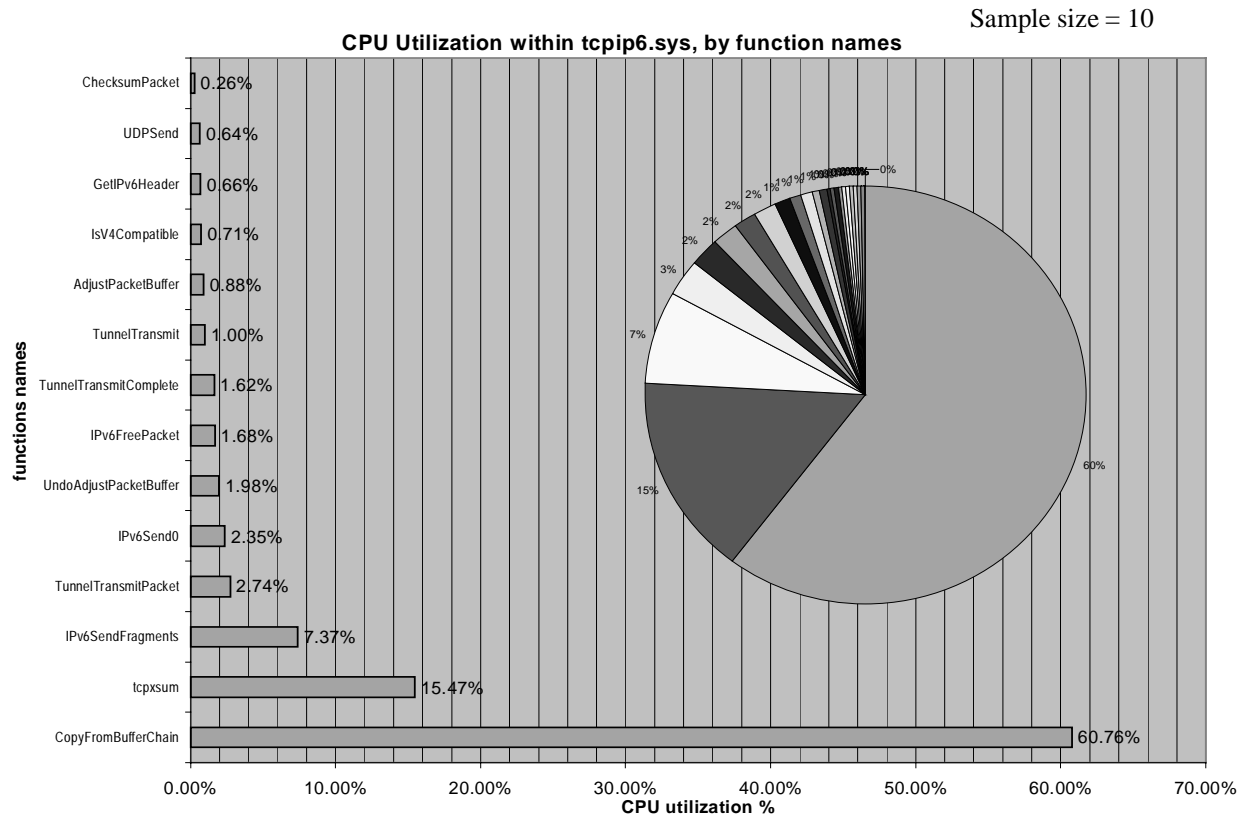


Figure 5.30 CPU utilization of individual functions within tcpip6.sys, UDP over IPv6

Within the tcpip6.sys module in UDP mode, the stack spends quite a large amount of its CPU utilization (61%) running on the data-moving function `CopyFromBufferChain()` in the tcpip6/ip6/subr.c source module. This function is not observed by VTune in TCP mode, which means it takes no more than 0.1% of the stack's CPU time when in TCP mode. It is called by the IP layer function `IPv6SendFragments()` to copy data buffer into fragments before the actual sending. It could be a system bottleneck if it's not fully optimized. In other words, the performance of UDP over IPv6 could be significantly increased if this function has been optimized.

Of the stack's CPU utilization, 15% was spent on the checksum calculation function `tcpsum()`. The fragmentation function for IPv6, `IPv6SendFragments()` takes up another 7%. The IP layer send routine `IPv6Send0()` takes up 2.35%. The link layer level transmit routines, `TunnelTransmit()`, `TunnelTransmitPacket()`, and `TunnelTransmitComplete()` take 5.36% of the stack's CPU utilization altogether. Same with the TCP measurements, the CPU utilization for UDP over IPv6 is measured by Vtune based on the sampling at a 1-microsecond interval during a 5-second period.

5.4.3 Summary of CPU utilization for TCP and UDP

With the measurement results obtained from VTune, we can categorize the CPU utilization within MSR IPv6 protocol stack. Table 5c gives a summary of the CPU utilization of the key functions within the stack. We classified the overhead into these categories: (a) checksum operations, (b) Data moving operations, which copy data between buffers, (c) fragmentation operations, which fragment data in IPv6 layer, (d) buffer management operations, (e) protocol-specific sending operations, (f) protocol-specific receiving operations. (g) protocol-specific data structure manipulating operations. This categorizing scheme is simplified from that discussed in chapter 2.4, as some of the overheads are protocol implementation dependent. For example, we classified the processing overheads for all the MSR IPv6 sending functions into the protocol-specific sending overhead category.

There are many other functions whose CPU utilization is less than 1% of the stack's CPU time. These functions together take up 10.46% and 5.03% of the stack's CPU time in TCP and UDP. For simplicity, these functions are excluded from the table below.

TCP/IPv6 Function calls	CPU utilization	Overhead category
<code>Tcpsum</code>	29.90%	Checksum
<code>TCPSend</code>	20.57%	Protocol specific, sending
<code>TCPSendComplete</code>	4.77%	Protocol specific, sending
<code>ChecksumPacket</code>	3.77%	Checksum
<code>TunnelTransmitPacket</code>	3.61%	Protocol specific, sending
<code>TCPReceive</code>	3.05%	Protocol specific, receiving
<code>IPv6Send0</code>	2.70%	Protocol specific, Sending

UndoAdjustPacketBuffer	2.58%	data move
FillTCPHeader	2.57%	Protocol specific, data structure
IPv6Receive	2.26%	Protocol specific, Receiving
TunnelTransmitComplete	1.95%	Protocol specific, Sending
TunnelReceivePacket	1.86%	Protocol specific, Receiving
IPv6Send1	1.74%	Protocol specific, Sending
ProcessTCBDelayQ	1.66%	Protocol specific, data structure
Net_long	1.54%	Protocol specific, data structure
LanReceive	1.49%	Protocol specific, Receiving
TunnelTransmit	1.27%	Protocol specific, Sending
FindAddressOnInterface	1.24%	Protocol specific, data structure
AdjustPacketBuffer	1.02%	Buffer operation
Others	10.46%	- -

Table 5c Functions with CPU utilization greater than 1% in TCP/IPv6

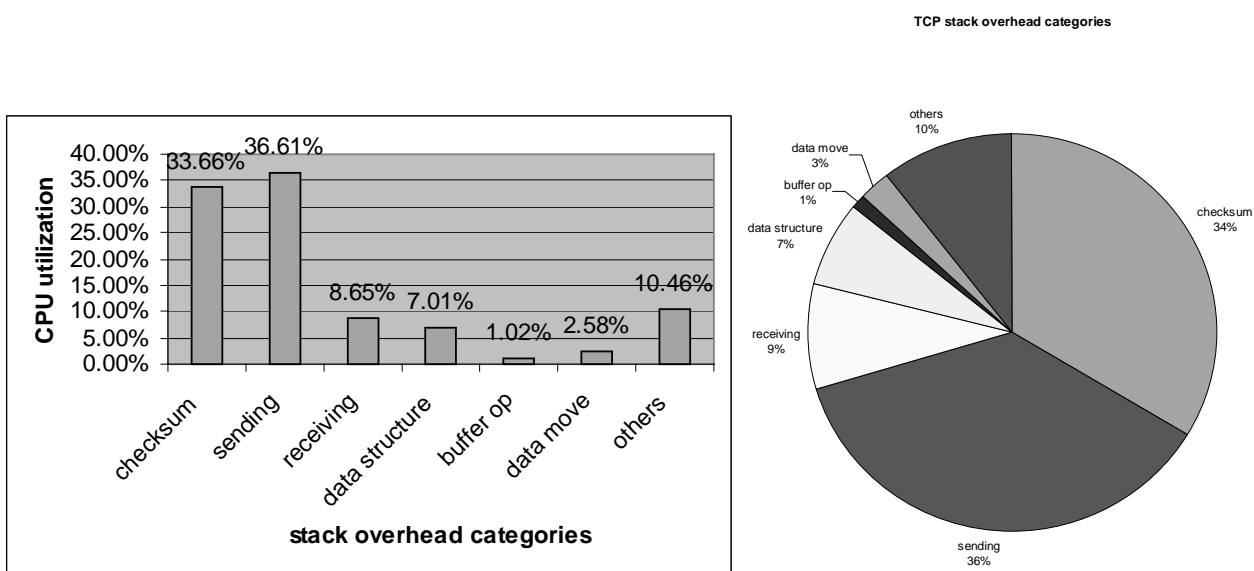


Figure 5.31 TCP over IPv6, CPU utilization by stack overhead category

In TCP over IPv6, the stack spends the largest portion of its time in performing protocol-specific sending operation, which takes up 36.61% of the stack’s CPU time. The checksum calculation operation consumes 33.66% of the stack’s time. The protocol specific-receiving operation (for handshaking and acknowledging) takes up 8.65%. The data structure management operations take 7.01%. The buffer management operations and data moving operations take 1.02% and 2.58% separately, while the rest of the functions, those which consumes less than 1% of the stack’s time and are not evaluated, accounts for 10.46% of the stack’s CPU time.

UDP/IPv6 Function calls	CPU utilization	Overhead category
CopyFromBufferChain	60.76%	Data move
TcpXsum	15.47%	Checksum
IPv6SendFragments	7.37%	Fragmentation in IPv6
TunnelTransmitPacket	2.74%	Protocol specific, Sending
IPv6Send0	2.35%	Protocol specific, Sending
UndoAdjustPacketBuffer	1.98%	Buffer operation
IPv6FreePacket	1.68%	Buffer operation
TunnelTransmitComplete	1.62%	Protocol specific, Sending
TunnelTransmit	1.00%	Protocol specific, Sending
Others	5.03%	- -

Table 5d Functions with CPU utilization greater than 1% in UDP/IPv6

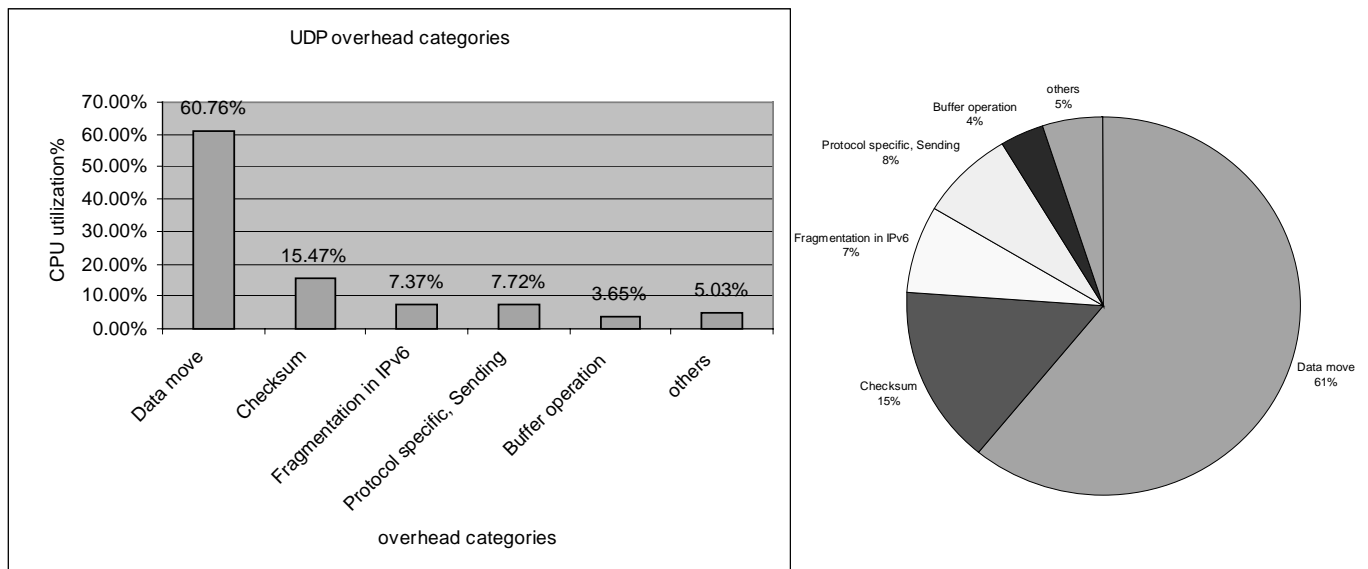


Figure 5.32 UDP over IPv6, CPU utilization by stack overhead category

In UDP over IPv6, the stack spends the largest portion of its CPU utilization on data moving operation, which consumes 60.76%. This is the CPU utilization of function `CopyFromBufferChain()` called by `IPv6sendFragments()` during UDP sending. This percentage is much larger than the data-move operations of TCP (2.58%). The fragmentation overhead in IPv6 layer for UDP takes about 7.37% of the stack's CPU utilization, while almost no fragmenting overhead is observed in TCP (`IPv6SendFragments` is only called in

UDP, not in TCP). This is because throughout our test sending with `ttcp6.exe`, TCP/IPv6 takes the MTU of 1460 bytes as the MSS (Maximum Segment Size). The small MSS waives TCP from doing fragmenting and copying data to fragment buffers, and it greatly reduces the data moving and fragmentation overheads in TCP transmissions.

In UDP over IPv6, the protocol-specific sending operations take up 7.72% of the stack's CPU utilization. Checksum calculation operation and buffer management operations take up 15.47% and 3.65% respectively.

Chapter Summary

X: number of 1000-byte payload data, Y: latency in μ s

	TCP over IPv6	UDP over IPv6
Start latency through the whole stack(μ s)	$y = 0.243x + 109.2$	$y = 12.877x + 84.157$
Stop latency through the whole stack(μ s)	$y = 192.57x + 246.98$	$y = 128.54x + 49.117$
Peak payload throughput (for payload range from 100 to 20K bytes)	53.125Mbps	91.7Mbps
Average maximum payload throughput	39Mbps	62.5Mbps
Total checksum latency	$y = 15.888x + 28$	$y = 11.428x + 27.756$
Average percent of total checksum latency to stop latency	11.49%	14.66%
Buffer allocation and de-allocation latency	$y = 4.0685x + 17.703$	* max 201.86 μ s, min 12.21 μ s
Average percent of buffer management latency to stop latency	3.78%	11.68%
IP Data moving latency	**	$y = 28.775x - 36.468$
Average percentage of data moving latency to stop latency	**	18.53%
CPU utilization of the stack module	12.57%	25.29%

Table 5e Summary of Chapter 5

*: Non-linear, please reference to Figure 5.16

**: Not observed in TCP over IPv6

Chapter 6 Conclusion and future study

6.1 Conclusion

We designed and performed source-based instrumentation on a testbed with Pentium machines running on the Windows NT 4 SP3 platform to measure the performance of the MSR IPv6 protocol stack. The measurements were obtained on the latencies of data transmission within the stack of TCP and UDP over IPv6. In our instrumentation model, the time-stamping technique and the data outputting mechanism have worked well for achieving the measurement objectives. The only drawback is that, in order to correctly obtain and interpret the data, the probing points within the source code of the protocol stack must be carefully established, based on in-depth understanding of the implementation hierarchy of the MSR IPv6 stack, as well as the Windows NT networking architecture.

The MSR IPv6 release 1.1 implementation, on which our experiments were performed, is a preliminary but functioning stack that complies with the IETF IPv6 standards. It is built as a separate stack from IPv4. TCP and UDP protocols that run on top of it remain mostly the same as those for IPv4 from which they are ported. Based on our testbed with Pentium 200MHz NT stations and 100Mbps fast Ethernet, the payload throughput (without any encapsulation overheads) of TCP and UDP over IPv6 for small and medium size (100 bytes to 20,000 bytes) packets trend-lined to a maximum of 39Mbps and 62.5Mbps. These are relatively low throughput numbers, reflecting the substantial overheads imposed by the protocol stack on our testbed. The 6-over-4 implementation mechanism also degrades the performance of the MSR IPv6 stack on our testbed. We observed that the start and stop latency of both TCP and UDP over IPv6 increase linearly with payload size. Generally, TCP has shorter start latency, longer stop latency and lower payload throughput than UDP because of its connection-oriented nature. Within the stop latency, the overall checksum processing time takes up 11.5% in TCP and 14.7% in UDP; the buffer management overhead takes up 4% in TCP and 12% in UDP; the data-move overhead takes up 18.5% in UDP.

We also measured the CPU utilization for the stack module as well as those for the functions within the stack using VTune. The stack module of MSR IPv6 utilizes the CPU at 12.6% in TCP and 25.3% in UDP. We observed that checksumming, protocol-specific sending, receiving, and data structure manipulating are the major

functions in TCP over IPv6 that dominate the stack's CPU utilization; while data moving, checksumming, fragmenting, and protocol specific sending are the dominating functions in UDP over IPv6.

Since MSR IPv6 release 1.1 is not a fully completed and optimized version, the above performance of MSR IPv6 may not reflect that of Microsoft's official release. It is suspected that some of the algorithms used in the implementation, such as data moving operations for IP fragmentation, still have room for optimization.

In spite of this, MSR IPv6 release 1.1 is a good starting point of studying and understanding the Windows NT network protocol implementation on the TDI-stack-NDIS architecture. The whole implementation of this MSR IPv6 release are well organized in a modular structure with detailed in-line comments in the source code, without which our instrumentation efforts would have been much more difficult.

6.2 Future study

This study is just a start of the source-code based instrumentation and performance evaluation on IPv6 protocol for Windows NT. Many aspects of this interesting research can be further expanded both in scope and in depth. Confining the study to the Windows NT platform, we can extend the study to include the following:

- (a) Automating the analysis of raw data. We utilize Excel 97 to do most of our calculation and analysis on the raw data semi-manually. If the tests need to be repeated for large number of times, automation of the calculation and analysis on raw data will be more efficient.
- (b) Performing a study on the upward call path on the receiving side. So far we have only examined the downward path through the stack which is on the sending side. We need to repeat the research and measurement on the receiving side to have a complete picture of the performance of the stack's functionality.
- (c) Performing a study on release 1.2. For this study, all our tests and measurements are run on release 1.1 for consistency. Future tests and measurements on release 1.2 may reveal different performance behavior or bottlenecks with the IPv6 add-on functionalities such as routing headers, encryption and mobility processing,
- (d) Comparative evaluation on MSR IPv6 with IPv4 on the same platform. If the source code of TCP/IPv4 implementation for Windows NT is accessible, the comparative tests and evaluations will definitely help and contribute to profiling and categorizing the processing overheads, and identifying throughput bottlenecks.

Beyond the NT operating system, the study can be extended to comparative evaluation with IPv6 implementations on the other platforms, such as Linux, and the other Unix platforms. With the development and deployment of IPv6, cross-platform comparative evaluation of the IPv6 protocol stack will be more and more important in aiding of profiling protocol processing overheads and protocol standardization.

References

- [1] Mei-Ling Liu, A look at Network Performance Benchmarks, Draft, Cal Poly University, SLO, August 1997. Annotation: A synopsis on the use of benchmarking as a network performance evaluation tool.
- [2] Richard P. Draves, Allison Mankin, Brian D. Zill, Implementing IPv6 for Windows NT. Proceedings of the 2nd USENIX Windows NT Symposium, Seattle, WA, August 3-4, 1998.
<http://www.research.microsoft.com/msripv6/usenixnt/paper.htm>
Annotation: Direct comments on the MSRIp6 protocol from the programmers in Microsoft Research who actually implemented the stack for Windows NT platform.
- [3] David D. Clark, Van Jacobson, John Romkey, Howard Salwen, An Analysis of TCP Processing Overhead. IEEE Communications Magazine, June 1989. Annotation: A very useful classical research paper on TCP performance. It said TCP is in fact not the source of the overhead often observed in packet processing, and it could support very high speeds if properly implemented.
- [4] Ramon Caceres, Peter B. Danzig, Sugih Jamin, Danny J. Mitzel, Characteristics of Wide-Area TCP/IP Conversations. ACM SIGCOMM 1991. Annotation: This paper characterized WAN applications that use TCP protocol, described a new way to model WAN traffic generated by a stub network. Could be useful in studying congestion control, routing algorithms and resource management schemes.
- [5] Jonathan Kay, Joseph Pasquale, Profiling and Reducing Processing Overheads in TCP/IP. ACM Networking Journal, 1996. Annotation: This paper presents detailed measurements of processing overheads for the Ultrix 4.2a implementation of TCP/IP network software running on a DEC station 5000/200. The performance results were used to uncover throughput and latency bottleneck.

- [6] Henrik F. Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon W. Lie, Chris Lilley, Network Performance Effects of HTTP/1.1, CSS1, and PNG. June 24, 1997.
<http://www.w3c.org/Protocols/HTTP/Performance/Pipeline.html>
Annotation: Research notes on performance of HTTP, CSS1 and PNG
- [7] Saurab Nog, David Kotz, A Performance Comparison of TCP/IP and MPI on FDDI, Fast Ethernet, and Ethernet. Dartmouth College Technical Report, Nov.22, 1995, Revised Jan.8, 1996.
<ftp://ftp.cs.dartmouth.edu/TR/TR95-273.ps.Z>
Annotation: Lots of charted data of latency, bandwidth and contention, obtained from tests for 2 protocols (TCP/IP and MPI-Message Passing Interface) and 3 types of networks (FDDI, 100Mbps Fast Ethernet and 10Mbps Ethernet).
- [8] Christian Huitema, IPv6 The New Internet Protocol, Second Edition, Prentice Hall PTR, 1997. ISBN 0-13-850505-5. Annotation: A complete reference on IPv6 standard specification
- [9] Stewart S. Miller, IPv6 The Next Generation Internet Protocol, Digital Press, 1997. ISBN 1-55558-188-9.
Annotation: A well-explained text on IPv6 protocol suites. Focus on protocol functionality and implementation
- [10] Matthew G. Naugle, Network Protocol Handbook, McGraw-Hill, Inc. 1994. ISBN 0-07-046461-8.
Annotation: A complete reference of network protocols, including Ethernet, token ring, IEEE 802.2, XNS, Novell Netware, TCP/IP, AppleTalk, DNA, and Local Area transport.
- [11] Robert Hinden, IPng Standardization Status. A set of web links of IPv6 RFC draft standards. Last updated January 25, 1999. <http://playground.sun.com/pub/ipng/html/specs/standards.html> Annotation: This is a set of nice web links that point to the draft standards of IPv6 protocols.

- [12] Andrew S. Tanenbaum, Computer Networks, the third edition, Prentice Hall PTR 1996 ISBN 0-13-349945-6. Annotation: A very good textbook on computer network architecture, TCP/IP and OSI seven layers reference model.
- [13] A Complete RFC list sorted by RFC numbers, IETF web page. Last visited in March 1999
<http://www.ietf.cnri.reston.va.us/rfc/> Annotation: RFC specifications, Input: RFC number, e.g. RFC-1388, Output: full text of that RFC.
- [14] Lampros Kalampoukas, Anujan Varma, K.K. Ramakrishnan, Improving TCP Throughput over Two-way Asymmetric Links: Analysis and Solutions, ACM Metric 1998, Performance Evaluation Review p.78-p.89
- [15] Lampros Kalampoukas, Anujan Varma, K.K. Ramakrishnan, Two-Way TCP Traffic over Rate Controlled Channels: Effects and Analysis, IEEE/ACM Transactions on Networking, Vol.6, No.6, Dec 1998, p.729-p.742
- [16] Craig Zacker, TCP/IP Administration, IDG Books Worldwide, Inc. 1997. Annotation: A complete reference of TCP/IP protocols stack from design to administration.
- [17] Rober M. Hinden, IP Next Generation Overview, May 14, 1995
<http://playground.sun.com/pub/ipng/html/INET-Ipng-Paper.html> Annotation: A good introduction to IPv6 as well as the overview of IPv6 including header formats, extensions, addressing, routing, QoS capability, security, and transition mechanisms.
- [18] Siu-Ming Lo, Performance Measurements of Network Applications, June 1999, California Polytechnic State University, Master Thesis of Computer Science.