COMPARING TCP-IPV4/ TCP-IPV6 NETWORK PERFORMANCE

A Thesis Presented to the Faculty of the Graduate School
University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

HARSHIL SHAH

Dr. Gordon K. Springer, Thesis Advisor

DECEMBER 2013

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled

COMPARING TCP-IPV4/ TCP-IPV6 NETWORK PERFORMANCE

Presented by

Harshil Shah

A candidate for the degree of

Master of Science

And hereby certify that in their opinion it is worthy of acceptance.

_____

Dr. Gordon K Springer

_____

Dr. Dmitry Korkin

_____

Dr. Justin Legarsky

ACKNOWLEDGEMENTS

I would like to acknowledge and thank, with gratitude, the following people who helped me throughout my studies and completion of my project.

First and foremost, my debt of thanks to my advisor, Gordon K Springer. I would like to thank him for his expert guidance, tutelage and confidence. I would also like to thank him for his patience entrusted upon me to complete the project, while living hundreds of miles away, employed in full-time job. The project would not have been successful without his motivation.

Secondly I would like to thank to my committee members for taking time from their busy schedules and contribute valuable insights.

Lastly I would like to thank to my friends and my family for their support and collaboration in completion of this project. My honors and achievements are dedicated to all of these people.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The Internet Protocol version 4 (IPv4) has been the backbone of the Internet since its inception. The growth and success of the Internet has accelerated the consumption of the IPv4 address space and hence its exhaustion is predicted very soon. Despite the use of multiple hidden and private networks to keep things going, a newer version of the protocol, Internet Protocol version 6 (IPv6), is proposed to solve this issue along with many other improvements as part of a better, newer design. For smoother transition and given the decentralized nature of the Internet, both of the protocol stacks, namely IPv4 and IPv6, are expected to be supported by the hosts and hence co-exist for a period of time. Many application programs, especially those involved in large data transfers, currently use the TCP/IP protocol suite. However, there have not been many attempts to leverage the existence of both Internet Protocol versions over a TCP connection.

This thesis, through a prototype, is an attempt to improve the network utilization by using either an IPv4 or an IPv6 protocol for a TCP connection based on end-to-end measured performance between two hosts. A measurement tool, named *netaware*, is developed as part of this thesis to measure the end-to-end network performance for both IPv4 and IPv6 protocols within a single tool. The tool measures two performance parameters, namely the bandwidth and the latency in a multi-threaded environment. The tool utilizes a simple middleware application, also built as part of this thesis, to create and use

socket connections for interprocess communication across the network between the two hosts. The middleware application is used as an intermediate level application to take care of creating IPv4 or IPv6 connections between the hosts, needed to transmit measurement and control data while measuring the performance parameters. The use of middleware application facilitates the construction of network applications by having an application developer to deal with minimal code to use either IP protocol. The network application may be a file transfer application or any other network application. The middleware application also provides the capability for TCP applications to switch between IPv4 and IPv6 network protocols on the fly, without impacting the application's functionality. The input values for the parameters of the middleware application functions internally control the IP protocol to operate on and the switching technique for an application program. The aim is to enhance the network utilization by having an option of switching between the two IP protocols in order to provide better performance at some point of time. The prototype measurement tool measures the network performance to help decide on the preferred protocol. The preferred protocol can then be used to notify the application program using the middleware application to switch to the preferred protocol while in execution. The combination of the measurement tool to measure the performance for both IP protocols within a single tool and the middleware application's ability to switch between the IP protocols at any point of time, based on measured performance, can help provide better network utilization.

Chapter 1 – Introduction

In today's information age, data is accessed through various forms of media, be it a paper based newspaper or a digitally stored book. The use of data through various sources has grown exponentially, especially during last decade. These sources can be broadly classified as either digital or non-digital sources. The Internet can undoubtedly be termed as a major contributor to the growth as well as use of digital data. The backbone of the Internet can be attributed to the Internet Protocol version 4 (IPv4) and its success. IPv4 has been in existence for more than fifty years. Due to exponential increase in the number of devices interconnected worldwide, the exhaustion of the remaining pool of IPv4 addresses is predicted soon. The Internet Assigned Numbers Authority (IANA) recently assigned the last block of IPv4 addresses on February 3rd 2011 [1]. The Internet Protocol version 6 (IPv6) with an extended address space is proposed to meet this addressing shortage. This new version is not a simple derivative of IPv4, but an improvement over the previous version, while keeping many of the characteristics of the IPv4 protocol. IPv6 is designed to have many additional features such as optional IP headers, class and flow labels, jumbo datagrams and fragmentation-less data transfer to name a few [4] [5]. Thus, the aim is to replace the older version of the IPv4 protocol, to meet the increasing demand for IP addresses and also to use the new features offered by the new version.

However, due to the vast success and wide spread use of the World Wide Web, the monetary cost and time involved, the transition, to IPv6 is occurring gradually as opposed to a sudden conversion. The two protocol stacks are expected to coexist for an extended period of time and to be supported by almost every host. Although, the past few years have witnessed a global scale deployment of IPv6 networks among IPv4 networks especially in Europe and the United States. This support for both the protocols means, a host can be reached by both the stacks, IPv4 and IPv6. Both of the network stacks may or may not follow the same network paths based on the underlying network infrastructure. Even though IPv6 nodes have increased in recent years, there has not been a corresponding increase in applications using or switching to the IPv6 protocol. With relatively light traffic load on IPv6 and abundant IPv6 backbone bandwidth, there is a high probability of greater IPv6 Bandwidth availability than IPv4 [2]. Additionally, there are still large IPv6-over-IPv4 tunnels widely in use where native IPv6 connectivity is not available [3].

In any public service network, a continuous performance improvement and the offering of a better Quality of Service (QoS) to an end user has always been a key challenge. At any given point of time, the performance over a particular network path with the same underlying infrastructure is not constant. The presence of cross traffic from a wide range of digital sources adds to the dynamics of the changing traffic patterns, especially when the data is bursty in nature. Hence, the QoS offered to an application by either IP protocol, the network path utilization can vary over a given period of time. Thus we have a

choice between IPv4 and IPv6 network protocol that an application could choose. This may help to gain better performance and network utilization at a given point of time based on QoS offered.

Many transport layer protocols, namely the Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Stream Control Transmission Protocol (SCTP) rely on the Internet Protocol IP as its underlying network layer protocol. Of the above mentioned transport layer protocols, TCP is most widely used protocol. Many application layer protocols like the HTTP, BGP, FTP and SMTP depend on TCP due to its rich collection of features like congestion control, reliable data transfer, and reordering of data packets for sequential transfer [35]. TCP is a connection-oriented protocol. A connection is established between two hosts prior to transmitting the application data, and each data packet travels over the same network path. Thus for TCP, choosing between IPv4 and IPv6 as its underlying IP protocol can drastically change how an application performs.

This thesis focuses on helping to choose between the IPv4 and IPv6 protocol for a better performance and network utilization for an application at a given point of time. A simple prototype middleware software foundation is built, which not only helps in developing applications to use either IPv4 or IPv6 protocol with ease but also provides a switching capability between either IP protocols, on the fly without impacting the application's functionality. A demonstration application is built to demonstrate its usefulness and its capability to switch between either the IPv4 or IPv6 protocol. The same middleware

foundation/application is also used to construct a measurement tool named *netaware*, to measure end to end performance parameters for both the IPv4 and IPv6 protocols within a single tool. The parameters measured include bandwidth and latency, but can be further extended to measure other performance parameters like packet loss and CPU utilization. This is done in a multi-threaded environment utilizing algorithms explained briefly in subsequent chapters. Here the aim is to provide an option to use either Internet protocol version to enhance the network utilization and leverage the existence of dual protocol stacks for better performance. With the combined use of the middleware application and the measurement tool, an application can choose to use either IP protocol by doing one time performance measurement using the measurement tool or choose to switch to either protocol version while in execution. This application can be a simple file transfer application or a distributed application involving large data transfers.

This thesis is organized into seven chapters. In the next chapter, a detailed description of the IPv4 shortcomings, the IPv6 protocol and its enhancement is provided. This chapter helps the reader understand how an application built using the IPv6 protocol may benefit utilizing its rich new features. It also discusses the current transition mechanism from IPv4 to IPv6. For brief descriptions of IPv4 headers and addressing mechanism, refer to Appendix A. Chapter 3 outlines different types of performance parameters, also known as metrics, and different types of bandwidths. It also provides details on the basics of two of the measurement techniques implemented in the measurement tool

netaware built as part of this thesis. Chapter 4 and 5 provides a detailed description of the design and implementation of the middleware application and measurement tool built respectively. These chapters also detail the algorithms used to measure different types of bandwidths and latency. Chapter 6 discusses the measurement performed for both IPv6 and IPv4 protocols between two hosts on the Research Network (Rnet) at the University of Missouri to demonstrate the utility of the measurement tool. Chapter 7 concludes the thesis by summarizing the work done as part of this thesis. It also discusses potential enhancements and future expansion of the middleware application and the measurement tool.

Chapter 2 – Internet Protocol Version 6

The next generation protocol, Internet Protocol version 6 (IPv6), is designed to overcome some of the limitations of its predecessor IPv4. IPv6 has many useful features added as an enhancement over the existing architecture, based on years of experience with IPv4, to offer better services, but at the same time retain the benefits of the earlier version. An application using these features over an IPv6 protocol may provide better performance in terms of bandwidth and latency as opposed to the performance when using an IPv4 protocol, over the same network path or different network paths followed. In order to leverage the benefits of IPv6, the present connectivity and support for the IPv6 protocol between the communicating hosts (client and server) also plays a significant role. Different transition mechanisms from IPv4 to IPv6 are outlined in Section 2.8. Given the nature of the application and present network conditions for IPv6 support, it can be crucial as well as beneficial to choose an IP protocol that offers better QoS that suits an application. This QoS can be in terms of connectivity, bandwidth offered, latency or a combination of all these factors.

This Chapter starts by describing the basics of the Internet Protocol layer. Citing some of the shortcomings of the IPv4 protocol follows the initial descriptions. The discussion then moves on to describing the IPv6 protocol.

2.1 Internet Protocol Layer (IP)

The Internet architecture evolved out of experiences with an earlier packet-switched network called the ARPANET. It is also called the TCP/IP architecture due to the extensive use of both, TCP and IP in a large number of applications. The hierarchical layer of this architecture is as shown in Figure 2.1. This is in contrast to Figure 2.2 that shows the traditional Open Systems Interconnection (OSI) architecture. As seen from Figure 2.1, the IP (Network Layer or Layer 3) serves as the focal point. A common protocol of exchanging the packets among the networks forms what looks like an hourglass. [4]. The IP layer can be termed as the pivotal protocol of the TCP/IP protocol suite as all the Transport (Layer 4) protocols namely TCP, UDP, SCTP to name a few, data gets transmitted as IP datagrams. IP, by itself offers a connectionless and potentially unreliable service.

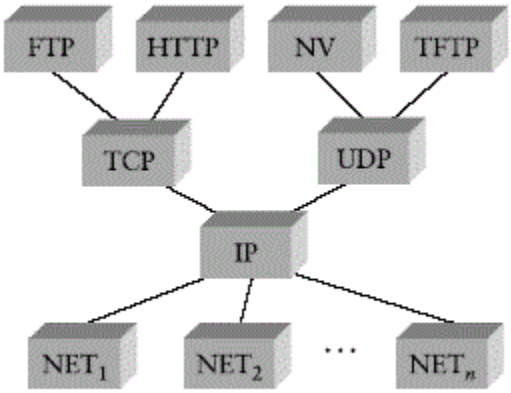


Figure 2.1: TCP/IP Architecture [4]

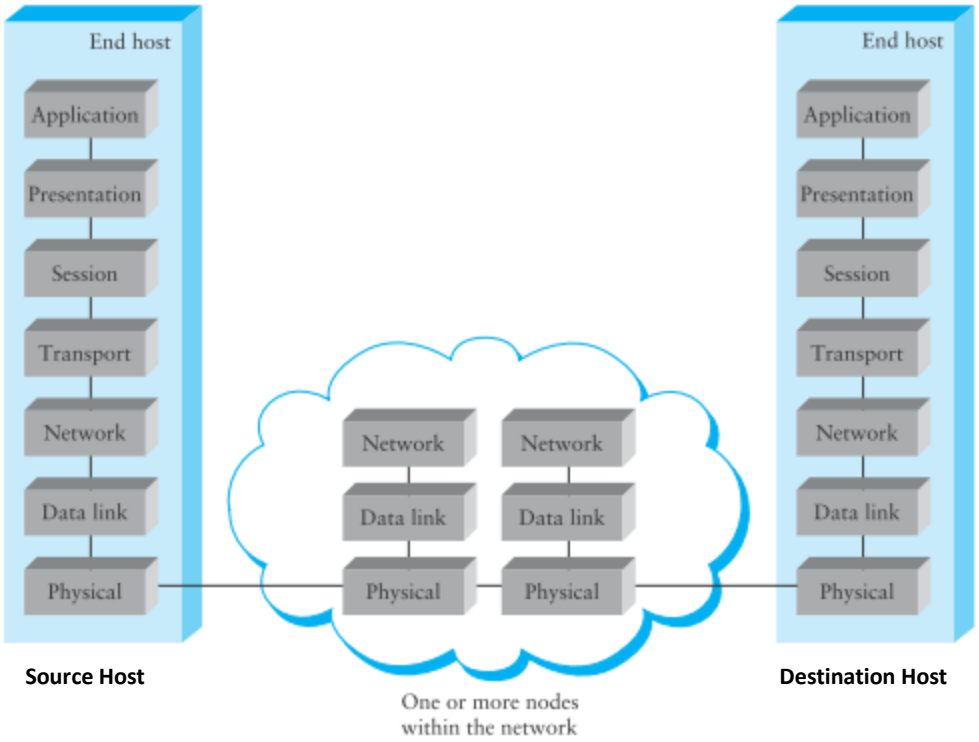


Figure 2.2: OSI Network Architecture [4]

The connectionless oriented service offered by IP is due to the fact that there is no logical connection or advance setup between the communicating hosts. Every datagram contains enough information to be correctly delivered to the destination. Neither the source nor the destination maintains any state information about any packet received from the upper layers. Each datagram is treated as a separate entity and handled separately from all other datagrams. Thus, a datagram may be delivered out of sequence or even multiple times or not at all.

IP offers unreliable service, as it does not guarantee that an IP datagram reaches its destination successfully. It also is called a 'best effort service.' The

best effort means, if a datagram is lost, mis-delivered or corrupted, the network does nothing. It simply drops the datagram, but may send an ICMP message back to the source. That is all it can do. The network simply made a best effort attempt to deliver the datagram. The layers above IP are supposed to handle any required reliability by providing the feedback in the form of acknowledgments to the sending host (example TCP). The philosophy behind this IP service model is to make it undemanding enough so that any network protocol or technologies built on top of the IP layer would be able to provide the necessary services [8].

2.2 IPv4 Shortcomings

The IPv4 protocol has served as the Internet backbone and has been quite robust and resilient for many years. The IPv4 protocol was designed in an early period of the Internet development when worldwide proliferation of Internet use was not anticipated. For many years, the architects and designers believed in the principle, "If it ain't broke, don't fix it." However, in the late 1980's it became apparent that new methods, perhaps a newer design, had to be developed to deal with the scaling problems due to the massive Internet growth. The techniques like Classless Inter-Domain Routing (CIDR) and sub-netting did ease the address allocation and helped conservation, but it was apparent that it would not suffice to contain the growing demand of IP addresses. Also, years of experience brought lessons and additional knowledge to indicate it would be better to build a newer protocol. Internet Protocol version 6 (IPv6), not only aims to prevent IP address

exhaustion, but also to provide a better Internet infrastructure. Before proceeding to IPv6 design features, let us look at some of the problems of IPv4 for better understanding as part of this section.

The first and the foremost issue is the *IP Address Exhaustion*. To identify a host on a network, it must have a unique IP address. The 32-bit address structure of IPv4 provides up to 4.3 billion unique addresses. However, as mentioned in preceding sections, the Internet's rapid growth has caused a shortage of unique IP addresses leading to its exhaustion. It is predicted that these additional addresses will run out sometime in the near future [8]. Moreover, large numbers of IP addresses are unusable as these are registered by IANA (Internet Assigned Numbers Authority) for only local or special uses while a considerable number of addresses are wasted due to the use of IP address classes. Thus, this indicates that the IP address space is much more constrained and there is need for newer protocol that provides much larger address space.

The second issue is regarding large routing tables. Before discussing details about this issue, please refer to IPv4 addressing and concept of class described in Appendix A, if needed. A separate routing table entry is needed for each network. If a network has slightly more hosts than a particular class, then it requires either two IP addresses of the same class or move up to the next class of IP addresses. For example, let us consider that a network has 280 hosts. Class C provides unique addresses for no more than 254 hosts (2^8-2) as 0 and 255 are not usable, whereas a class B network provides unique addresses for 65,534 (2^{16}-2) hosts.

Thus, this network requires either two class C address spaces or one class B address space. If a class B address space is assigned, a large number of host IP addresses are left unutilized. This wastes address assignment efficiency of only 0.42% (280/65535). If two class C address spaces are assigned, the efficiency increases to 54% (280/255*2), but it also increases the number of entries in the Internet backbone routing table and causes early exhaustion of the class C address space. Apart from the inefficiency in address assignment and growing routing tables, the routing process also needs to be simplified. The following are the points that need simplification and improvement.

1. Number of fields in the packet header.
2. Packet fragmentation at router.
3. Computing the packet checksum at every router for integrity.
4. Computing the Time To Live (TTL) field at every intermediate router.

More details on how simplification is achieved on each of the above factors can be found in Appendix A.

The third major issue is security. Security is the key to many applications and it has gained increasing importance over a period of time. Security is optional in IPv4 and that is one of its limitations. Quality of service (QoS) is another major factor that is optional and implemented as a type of service field in its header. Packets from different services, application need different priorities while processing at intermediate routers. For example, a packet body can be encrypted, but QoS will still function because the header portion containing the QoS instructions is not encrypted. These packets may also need more bandwidth. As

these features are not directly integrated in the protocol, most of the router systems may ignore these fields. Thus the IP layer needs to have these features to provide better services.

Apart from these limitations, new features are also needed. For example, improved multicast capabilities, auto-configuration and plug and play needs to be accommodated. Taking into consideration the need for new features along with the limitations of the existing protocol, the Internet Protocol 6 is designed as explained in next section.

2.3 Internet Version 6 (IPv6)

The entire IPv6 packet is as shown in Figure 2.3. The IPv6 header is always present and is a fixed size of 40 bytes. The Extension header varies in length and may or may not be present. Its details are described in Section 2.5. The Upper Layer Protocol Data unit (PDU) consists of an upper layer protocol header along with its payload. For example, the PDU of the Transport layer protocols like TCP consists of TCP header and its payload. The IPv6 header is then added to this PDU when it is transferred down to the network layer.

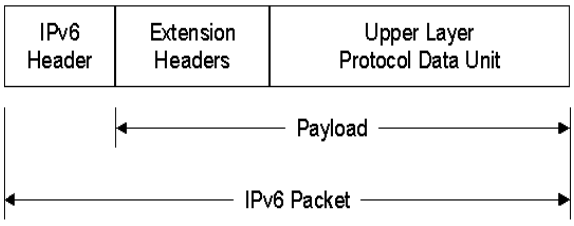


Figure 2.3: IPv6 Packet [10]

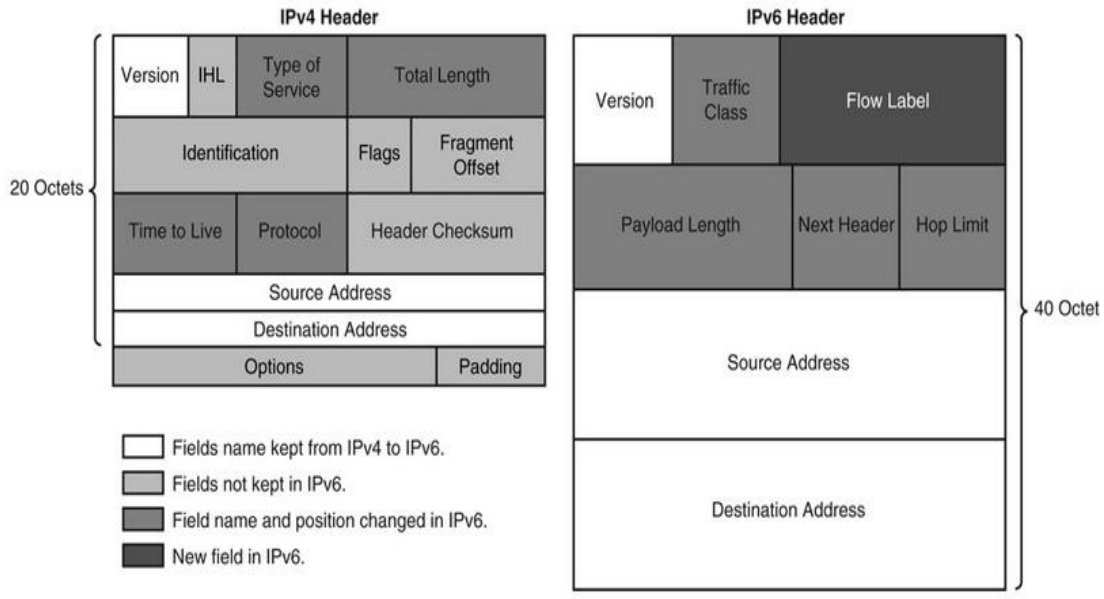


Figure 2.4: IPv6 Header Compared With IPv4 [11]

Figure 2.4 shows details of an IPv6 header. It uses an IPv4 header for comparison. The IPv6 header is a streamlined version of an IPv4 header. IPv6 headers have six fields and two addresses as compared to ten fixed header fields, two addresses and some options in IPv4. The **Version** field in a header is a 4-bit field that indicates the version of the IP header. Version is set to 0x6 for IPv6. The **Traffic Class** field is used to indicate the priority of the stream of packets. It is an 8-bit field and used to notify the routers to grant more attention for specific handling for that stream of traffic. The next field compared with IPv4 is a 20-bit field called **Flow Label** that indicates that a packet belongs to a stream or sequence of packets between the communicating hosts. The combination of this field and the previous field together provide Quality of Service (QoS) to specify the type of special handling needed by the intermediate routers. An example

would be to request prioritized delivery of a particular traffic flow [10]. The flow label is set to 0x0 for default handling. For a non-default quality of service a non-zero value is used for the packets involved in real time data like voice and video. Different flows between the same source and destination are distinguished by separate non-zero flow labels. The **Payload length** field indicates the length of data carried after the header. It is a 16 bit field and includes the length of the Extension headers and the upper layer Protocol Data unit (PDU) in bytes. The IPv6 header size is fixed and is always 40 bytes. Since it is fixed, there is no need for a field to specify the header length as compared to IPv4 header field **Total length**. The **Next Header** field is an 8-bit field and specifies the next extension header, if present or the transport protocol. The values used to indicate the upper layer protocol value remain the same as in IPv4. The next field is the **Hop Limit** field that indicates the number of hops after which the IPv6 packet is discarded. It is 8 bits in size and decremented by one at each node. It is similar to the Time to Live (TTL) field in IPv4 header except that there is no relation to the amount of time the packet is queued at a router [5] [10]. An ICMPv6 error message is sent to the sending host when the hop limit is decremented to zero. The next two fields are called the **Source Address** and the **Destination Address** and are each 128-bit in size. These fields uniquely identify the source and the destination addresses respectively. A comparison of IPv6 and IPv4 header fields is shown below in Table 2.1 [10].

IPv4 Header Field	IPv6 Header Field
Version	Same field but with different version numbers.
Internet Header Length	Removed in IPv6. IPv6 does not include a Header Length field because the IPv6 header is always a fixed size of 40 bytes. Each extension header is either a fixed size or indicates its own size.
Type of Service	Replaced by the IPv6 Traffic Class field.
Total Length	Replaced by the IPv6 Payload Length field, which only indicates the size of the payload.
Identification Fragmentation Flags Fragment Offset	Removed in IPv6. Fragmentation information is not included in the IPv6 header. It is contained in a Fragment extension header.
Time to Live	Replaced by the IPv6 Hop Limit field.
Protocol	Replaced by the IPv6 Next Header field.
Header Checksum	Removed in IPv6. In IPv6, bit-level error detection for the entire IPv6 packet is performed by the link layer.
Source Address	The field is the same except that IPv6 addresses are 128 bits in length.
Destination Address	The field is the same except that IPv6 addresses are 128 bits in length.
Options	Removed in IPv6. IPv4 options are replaced by IPv6 extension headers.

Table 2.1: IPv4-IPv6 Comparison [10]

The Flow label is a new field present in the IPv6 header, but not in the IPv4 header. Details about Flow field and its significance is described in Section 2.7.1

2.4 IPv6 Addressing

An IPv6 address is composed of 128 bits and uses hexadecimal notation. The address is divided into eight 16-bit words where each word is a combination of 4 digits. The words are separated by ':'. This is different than 32-bit IPv4 address notation that uses dotted decimal notation.

An example of an IPv6 address is 2610:e0:a010:302:218:8bff:fe3e:38b/64.

The IPv6 address notation is simplified for special types of IPv6 addresses. One of the types is the one with leading zeros with at least one numeric value in each field of the address. Another type is an address with a large number of contagious 0s that are written more compactly by omitting the entire 0 fields and placing "::" instead as is shown in the examples below.

Example: Lewis.rnet.missouri.edu has three addresses as follows:

inet6 addr: 2610:e0:a010:302::11/64 Scope:Global

inet6 addr: 2610:e0:a010:302:218:8bff:fe3e:38b/64 Scope:Global

inet6 addr: fe80::218:8bff:fe3e:38b/64 Scope:Link

The above addresses show the different forms with zeros suppressed, for both leading and consecutive 0s. Note that the "::0" can be used only once, otherwise it becomes ambiguous.

2.5 IPv6 Extension Header

The IPv6 header is a streamlined version of the IPv4 header. The options in IPv4 header require extra processing at the intermediate routers, which degrades the transmission performance. In order to improve the performance, the IPv6 packet options are moved to separate headers called the extension headers. These options are processed at the destination except for the Hop-by-Hop extension header. In IPv4, the payload follows the header. However in the case of IPv6, it is possible to add an arbitrary number of extension headers after the main header, followed by the actual data payload. This is called “daisy chaining” of extension headers. The identification of each extension header is by its header type and carries the next header type or the payload if it is the last extension header. Figure 2.5 shows a daisy chained extension header. Table 2.2 shows the IPv6 header extension types. A detailed description of each header can be found in [5].

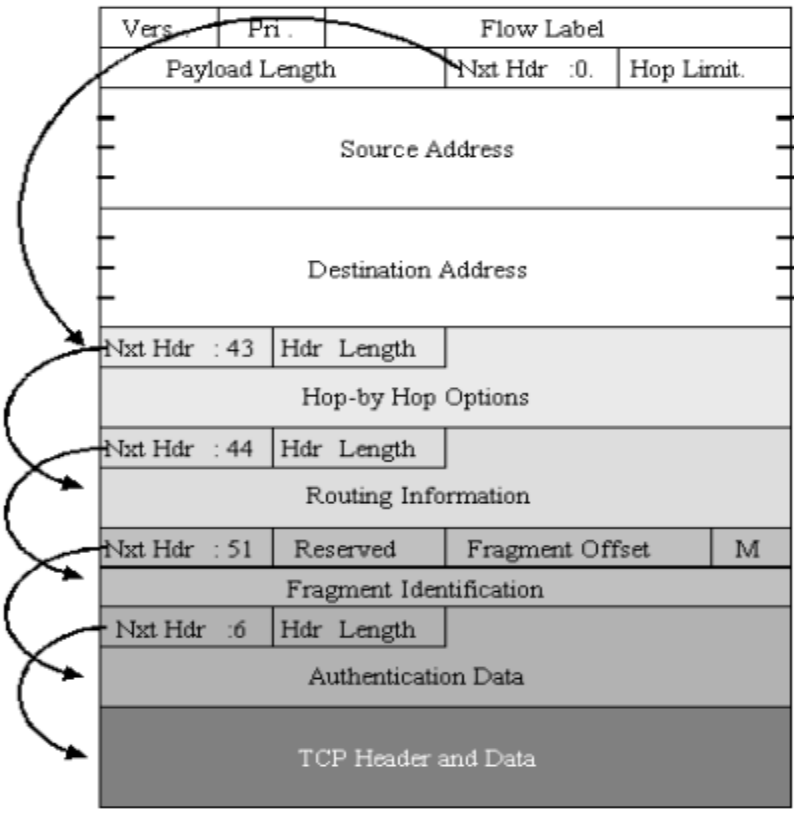


Figure 2.5: IPv6 Extension Headers [5]

Decimal value	Keyword	Header Type
0	HBH	Hop-by-Hop options
2	ICMP	Internet Control Message Protocol
3	GGP	Gateway-to-Gateway
4	IP	Ip-in-IP (IPv4 encapsulation)
5	ST	Stream
6	TCP	Transmission Control Protocol
17	UDP	User Datagram Protocol
29	ISO-TP4	ISO Transport Protocol Class
43	RH	Routing Header
44	FH	Fragmentation Header
45	IDRP	Inter Domain Routing

		Protocol
51	AH	Authentication Header
52	ESP	Encrypted Security Payload
59	Null	No next Header
80	ISO-IP	ISO Internet Protocol (CLNP)
88	IGRP	Interior Gateway Routing Protocol (proprietary to Cisco Systems)
89	OSPF	Open Shortest Path First (interior routing protocol)

Table 2.2: IPv6 Extension Header Types [5]

2.6 Enhancement Over IPv4

This section describes the enhancement of IPv6 for better understanding before moving to the details of its design. As mentioned, IPv6 not only overcomes the limitations of IPv4 mentioned in Section 2.2, but also provides several useful enhancements. According to the IPv6 specifications [5], the changes from IPv4 to IPv6 fall primarily into seven categories. The first change is expanded addressing capabilities. The size of an IP address is increased from 32-bits in IPv4 to 128-bits in IPv6. This increases the address space from 4.3 billion to 3.4×10^{38} unique addresses. This allows multiple levels of subnetting and address allocation from the Internet backbone to the individual subnets within an organization. The need to conserve addresses avoids Network Address Translation (NAT), thus making the network more transparent in terms of end users using these addresses.

The second change is in Anycast Addressing. The Anycast address is simply a Unicast address associated with more than one node or interface. It is used to send a packet to any one of a group of nodes that is the closest routable address.

This is useful for providing redundancy and load sharing to specific types of network services by assigning a common IP address to multiple instances of the same service. An example of such a network service would be DNS (Domain Name System) where root namesservers (DNS servers) of the Internet are implemented in a cluster of hosts using Anycast addressing. Each of the nameservers in DNS is configured with the same Anycast IP address to listen on. A client DNS resolver is configured within the DNS that has the same Anycast address as that of nameservers. It is responsible to resolve and choose a name server that is closest to it topologically for incoming requests to resolve a host name to an IP address. If the chosen nameserver fails for any reason, the DNS client packets are then routed to next nearest DNS server.

The third change is the IP header format change. The IP header in IPv6 is greatly simplified by dropping a few of the fields while making a few others part of the optional IPv6 Extension Header. This helps reduce the unnecessary amount of processing, fragmentation at the intermediate nodes and also to limit the bandwidth cost of IPv6 headers. IPv6 header and its fields are briefly described in Section 2.3.

The fourth change is tied to the changes in header format to provide improved support for Extensions and Options. The options in an IPv6 header are part of an Extension header that removes the stringent limit of 40 bytes of options as in IPv4, greater flexibility to add additional options and reduce unnecessary processing at intermediate nodes. This change also supports built in security by

providing extensions to support security options, such as authentication, data integrity, and data confidentiality that are built into IPv6 headers.

The fifth change is support for better QoS. IPv6 provides packet labeling for the sender to provide special handling of packets belonging to a particular stream or flow. This provides special treatment for some real time applications, which requires special handling by the IPv6 routers for Better QoS. This feature is still experimental and subject to change. Section 2.7 provides more details about this for packet labeling and concept of flow.

The sixth change is the way stateful and stateless addresses are configured. Configuration is automatic and simplified in IPv6 by means of Stateful or Stateless address configuration. In the presence of a Dynamic Host Configuration Protocol (DHCP) server, the host configures its address as a stateful address while in its absence the host configures itself as a stateless address. The stateless address is configured as a link local address (an address formed by concatenating the well known link local prefix and the 48-bit Ethernet address unique to that host interface) of a link and the address derived from prefixes advertised by local routers on the link. The Internet Control Message Protocol 6 (ICMPv6) discovery message is used to discover the local router [5]. If a local router is not present, hosts use the link local address to communicate. For example, the IPv6 addresses for one of the interfaces of Lewis.rnet.missouri.edu are as shown below.

inet6 addr: 2610:e0:a010:302::11/64 Scope:Global

inet6 addr: 2610:e0:a010:302:218:8bff:fe3e:38b/64 Scope:Global

inet6 addr: fe80::218:8bff:fe3e:38b/64 Scope:Link

Ethernet MAC HWaddr 00:18:8B:3E:03:8B

The first two addresses are global stateful addresses. The third address is a link local address and an example of the stateless address while the fourth address is the interface's Ethernet address. Figure 2.6 below shows how an IPv6 link local address is formed. As seen in the Figure 2.6, the first 64 bits fe80::/64 is the link local prefix advertised when a host is plugged into the network. The remaining address is obtained by converting the 48 bit Ethernet MAC address to 64 bit address by inserting the 16-bits "ff:fe" in the middle. Also the 7 bit most significant bit starting from 1 is inverted as per the EUI-64 standard. A more detailed description can be found in reference [5].

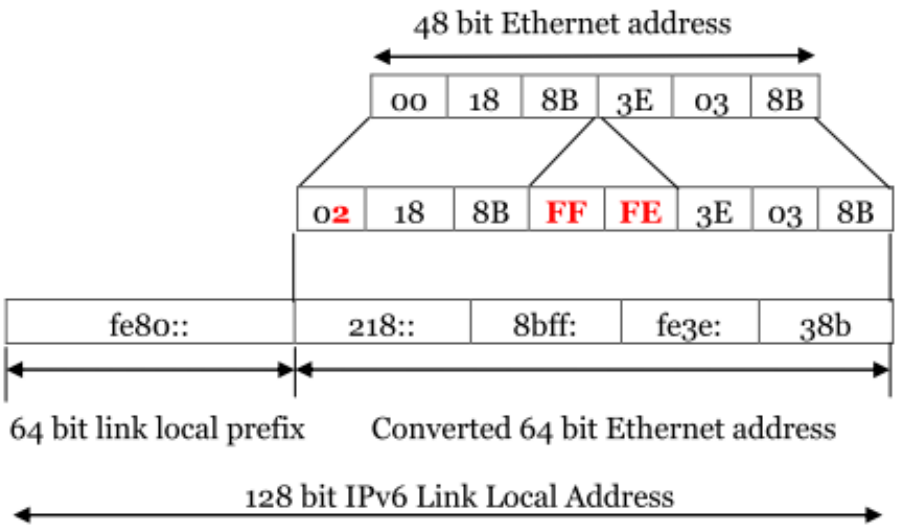


Figure 2.6: Lewis IPv6 Link Local Address

The seventh change is the option to send Jumbograms. The performance over high–MTU links, also called high speed or super computer networks, which allow transfer of large packets without fragmentation. Packets are just limited by the IPv4 packet size of 65,535 bytes (2^{16}-1). IPv6 increases this size by using the Jumbo Payload Option header and allows sending packets larger than the 64KB limit. The size of a Jumbogram can be as large as 4,294,967,295 (2^{32}-1) bytes [5]. The details of the effect of using Jumbograms as an enhancement over an IPv6 network on transport layer protocols can be found in [12].

2.7 IPv6 Quality of Service

The IPv6 protocol provides two new fields namely the '***Flow Label***' and the '***Traffic Class***' field to provide a non-default service to real time applications. As per IPv6 specification [9], the 20-bit Flow Label field is used by the source to label the sequence of packets for which it requests special handling by the intermediate routers. Its value ranges from `0x1 to 0xFFFFF` and is a pseudo-random number that is unique when combined with the source address. This capability provides quality of service in terms of consistent throughput, delay or jitter. Throughput is the amount of data that can be successfully transferred from a source to a destination within a given amount of time and measured in bits/second. Delay is the time required for a packet to reach its destination, while jitter is the delay variation.

2.7.1 The Concept of Flow

In order to indicate a stream of packets, traffic belonging to a particular flow, the flow field label is used. This field may be set to indicate that a particular flow requires additional resources and special handling by the routers along the path. As per the IPv6 specification [9] a flow is a sequence of packets that are sent from a particular source to a particular (unicast or multicast) destination for which the source desires special handling by the intervening routers. There can be multiple flows from a source to a destination and can contain traffic that is not associated with any flow (i.e., with flow value of zero). The combination of a source address and a nonzero flow label uniquely identifies a flow. Packets that do not belong to a flow carry a flow label of zero.

New flow labels are chosen randomly in a pseudo manner and ranges from 0x1 to 0xFFFFF. The source host assigns the flow label to a flow. The intervening routers use this randomly allocated set of bits within the flow label field as a hash key. The routers use this hash key to look up the state that is associated with the flow. All the packets that belong to the same flow must have the same source, destination address and the same nonzero flow label. If a packet has an extension header like hop-by-hop or the routing header, then the packets must all originate with the contents of that options header. The next header field in case of the hop-by-hop or the routing header is excluded. The routers or the destination though permitted may or may not verify whether the above conditions are satisfied. Any violation if detected, should be reported to the source by an ICMP problem

message with Code 0. The intermediate routers can also set, or modify the flow handling state associated with a flow, without any flow establishment information from a control protocol or any option header. For example, when a router receives a packet from a particular source with an unknown, non-zero flow label, the router can process the IPv6 header along with its extension headers. The extension header processing is done in the same way with the flow label field set to zero. The routers also determine the next-hop interface.

The routers might also update a hop-by-hop option, advance the pointer and addresses in a routing header, or decide how to queue the packet. The decision to queue the packet is based on the Traffic Class field of the packet. The routers can then choose to remember the results of those processing steps. Then the routers can cache the information. The routers use the source address and the flow label as the cache key. Subsequent packets, with the same source address and flow label can then be handled by referring to the cached information. The routers do not need to examine all those fields. The routers can assume that the fields are unchanged from the first packet that is checked in the flow.

2.7.2 Traffic Class

The source uses the traffic class field to mark a particular flow of packets to a particular class. Different classes indicate different priorities for a flow. The intermediate nodes use this field in an IPv6 header to make the identification and handle the flow according to the class also called the priority.

The following are the general requirements that apply to the Traffic Class field according to IPv6 specifications [9]

- The service interface to the IPv6 service within a node must supply the value of the Traffic Class bits for an upper-layer protocol. The Traffic Class bits must be in packets that are originated by that upper-layer protocol. The default value must be zero for all 8 bits.
- Nodes that support some or all of the Traffic Class bits can change the value of the bits. The nodes can change only the values in packets that the nodes originate, forward, or receive, as required for that specific use. Nodes should ignore and leave unchanged any bits of the Traffic Class field for which the nodes do not support a specific use.
- The Traffic Class bits in a received packet might not be the same value that is sent by the packet's source. Therefore, the upper-layer protocol must not assume that the values are the same.

Table 2.3 provides different Traffic Class field values.

Priority	Meaning
0	Uncharacterized traffic
1	"Filler" traffic (for example, netnews)
2	Unattended data transfer (for example, email)
3	(Reserved)
4	Attended bulk transfer (for example, FTP, HTTP)
5	(Reserved)
6	Interactive traffic (for example, telnet, X)
7	Internet control traffic (for example, routing protocols, SNMP)

Table 2.3: Traffic Class Field Values

2.8 Transition Mechanisms to IPv6

The transition to IPv6 has gained momentum in the past few years. However the IPv6 protocol is not backward compatible. The current hosts and routers based on the IPv4 protocol cannot directly handle the IPv6 traffic and vice versa. In order to achieve smooth and stepwise transition, IETF recommends different kinds of transition mechanisms. These mechanisms are broadly divided into three categories as described below. A detailed description can be found in [13]. The IPv4 and IPv6 protocols will coexist for an indefinite period until the transition completes. Thus, given the nature of an application, the available network paths for both the protocols between two hosts and the operating system involved, it is interesting as well as beneficial to know which IP protocol offers a better performance.

The first category is the Dual IP stack. The dual stack mechanism includes two protocols to operate either independently or in a hybrid form and allow network nodes to communicate either via IPv4 or IPv6. It is implemented in the operating system and is the foundation of other IPv6 transition mechanisms.

The second category involves the use of tunneling. The core of tunneling allows an IPv6 protocol packet to be encapsulated in an IPv4 protocol packet which then allows the IPv6 packets to be transmitted across the IPv4 network. For example, an IPv6 over IPv4 tunnel (IPv6 packet encapsulated in IPv4 packet) provides a method of using the existing IPv4 routing infrastructure to transfer IPv6 packets and reach other IPv6 enabled hosts. An IPv4 over IPv6 tunnel is just

the opposite of IPv6 over IPv4. The direct encapsulation of IPv6 datagrams within IPv4 packets is indicated by IP protocol number 41. IPv6 can also be encapsulated within a UDP packet. The format of IP packets in an IPv6 over IPv4 tunnel is shown in Figure 2.7

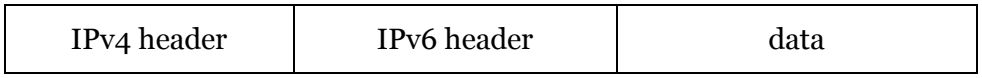


Figure 2.7: IPv6 Encapsulation [13]

The third category is known as IPv4 mapped IPv6 addresses. Translation technology can be used to implement address and protocol translation between IPv4 and IPv6. The hybrid dual-stack supports a special class of address called the IPv4-mapped IPv6 addresses where the first 80 bits of IPv6 addresses are set to zero and the next 16 set to one. The last 32 bits however is the actual IPv4 address in decimal dot notation format. An example is as follows '::ffff:192.0.5.127'.

This chapter briefly discussed the shortcomings of the IPv4 protocol and the design of its successor, the IPv6 protocol. As mentioned, IPv6 not only addresses the IP address space exhaustion issue, but also includes new features and enhancements. An application can be designed to use IPv6 protocol to leverage these new features to enhance its performance, especially over high speed networks, provided the underlying network supports it. Since both IP protocols are expected to coexist for a long period of time until the transition is

complete, it can be beneficial to measure the network performance between the two hosts over both IP protocols. This can help to make a decision as to whether it is beneficial to switch to the IPv6 protocol or use the IPv4 protocol over a network path. The next chapter briefly outlines various performance parameters and measurement techniques that can be used to measure network performance.

Chapter 3 – Network Performance Metrics and Measuring Techniques

Over the past several years, researchers have developed various techniques and tools to measure network performance given the hybrid nature of the underlying network. The traffic from various kinds of digital devices and the dynamic nature of changing traffic patterns adds to the complexity in performance measurement. Network performance can be measured at different layers of the Internet Protocol, but this thesis focuses on End-to-End network performance as seen and offered to an application running in a user's or application space. This chapter explains a few of the important network performance parameters and corresponding techniques to measure these parameters. These techniques are used and implemented within a single measurement tool, called ***netaware***, as part this thesis to measure the corresponding performance parameters for both the IPv4 and IPv6 protocols. The name *netaware* is chosen to signify the tool's importance to help applications be network aware based on measured performance. The design and implementation details of this tool are discussed in Chapter 5.

3.1 What Are Different Network Performance Metrics?

The performance of a network path can be broadly divided into four performance metrics. The first and the most commonly measured performance metric is the

bandwidth. Bandwidth is defined as the amount of data that a link or network path can deliver per unit time [34]. The bandwidth is measured in Bytes/second, bits/second or packets/second, based on the measurement technique adopted and the network layer where the measurement occurs. The term bandwidth can be further classified as capacity, available bandwidth and Bulk-Transfer-Capacity (BTC) [15] [34] as explained in the next section. BTC is also referred to as throughput.

The second important performance metric is latency. Latency is the delay of traffic through a network link and can be measured as one-way latency or the Round Trip Time (RTT). Round Trip is the difference between the time the packet is sent and the time it is received back. This metric plays a major role in network behavior. It can help reveal the network topology and the congestion along a network path especially for a TCP connection [16]. Previous research has shown that TCP throughput is inversely proportional to path RTT [16].

A third important performance parameter is the Packet Loss Ratio (PLO). PLO is defined as the number of packets lost during a transmission during a specified time interval. It is the ratio of the number of packets received versus the total number of packets transmitted. A high packet loss ratio indicates network issues like an inter link router problem, receiver machine being overwhelmed by the sender, high network congestion and various other network issues. Further analysis can also help reveal whether a network path supports a specific protocol, for example IPv4 or IPv6 or both.

The fourth performance metric that can be measured is the host CPU utilization. It is defined as the percentage of CPU utilization required for data transmission. This metric plays an important role for applications involved in large data transfers.

Of the above four metrics, this thesis studies various Bandwidth and Latency performance metrics. The next sections discuss the various types of Bandwidths, their measurement techniques. Implementation of one of the measurement techniques is integrated with latency measurement in the measurement tool *netaware*.

3.2 What Are Different Types of Bandwidths?

The term Bandwidth is broadly used to describe capacity, available bandwidth and Bulk-Transfer-Capacity in general as the rate at which data can be transmitted through the network. However there are subtle differences among these terms. It has its own relevance as to whether it is the bandwidth of a link, a sequence of successive links or end-to-end path or whether it is a layer-2 (data link), layer-3 (network layer) or end to end bandwidth at the application layer [34]. The methodology or the algorithm used decides which bandwidth is measured. Below is a brief description of each of the Bandwidth metrics. The term link or segment refers to a layer -2 (data link layer), while the term hop refers to layer-3 (network layer).

3.2.1 Capacity

At layer-2, a link, also called a segment [15] [34] has the transmission rate in a bit rate that is constant. For example, the data rate of a 10BaseT Ethernet segment is 10Mbps while for a T1 segment it is 1.544Mbps. The underlying propagation medium as well as its electronic or optical transmitter/receiver hardware limits this transmission rate [15]. At layer-3, the capacity of the hop is less than its transmission rate. This is due to layer-2 encapsulation and framing. The following equation gives the transmission capacity of a hop for a packet of size L_{L3} bytes.

$$\Delta L3 = (L_{L3} + H_{L2}) / C_{L2} \text{ [15]}$$

Where

C_{L2} – Nominal capacity of underlying segment

H_{L2} – Layer 2 overhead to encapsulate the IP packet.

So the capacity of the link CL3 for IP packet of size L_{L3} is

$$CL3 = LL3/\Delta L3 \text{ [15]}$$

For example, for 10BaseT Ethernet , C_{L2} is 10 Mbps and H_{L2} is 38 bytes (Ethernet header is 18 bytes, frame preamble is 8 bytes and 12 bytes for the inter-frame

gap), the capacity offered at layer-3 for packets of size 100-bytes is 7.24Mbps whereas for 1500 byte packets, the capacity is 9.5Mbps [15].

Since the Maximum Transmission Unit (MTU) is the largest packet that can be sent without fragmentation, the capacity of a hop, measured at the network layer, is the bit rate at which that hop can transfer packets of MTU size [15] [34]. Extending it further for a network path, the capacity of a path is the maximum capacity that a path can offer at layer-3 from source to sink machine. The source here represents the entity that sends the data while the sink represents the entity receiving the data. The equation describing this capacity is shown below.

$$C = \min_{i=1,\ldots,H} C_i \text{ [15]}$$

The C_i is the capacity of the *i*-th hop, and H is the number of hops in the path. The link with minimum capacity is called the narrow link and the overall capacity offered by a network path. Furthermore, traffic shapers or rate limiters on the network path further limit the maximum achievable capacity. Rate limiting is out of scope for this project and is not considered further. In short, the network capacity is the maximum capacity of a link or a network path to transmit data from one location in the network to another [34].

3.2.2 Available Bandwidth

The capacity of a path is the maximum available link bandwidth. As mentioned, it is based on the propagation medium and the transmission technology. However

the bandwidth offered by a path additionally depends on the traffic load, which varies from time to time. It is the unused or the spare capacity of link during a certain period of time [15]. The link is either transmitting at the full a link speed or it is idle, so the instantaneous utilization of a link can only be 0 or 1. The available bandwidth is the instantaneous utilization over the time interval of interest and thus varies from time to time [15] [34]. The average available bandwidth A_i of hop i for capacity C_i with average utilization u_i is the unutilized fraction of capacity. It is described in the equation below.

$$A_i = (1-u_i)C_i \text{ [15]}$$

Extending this further to a network path, the available bandwidth of a network path for H hops is the minimum of all the available bandwidth as described in the following equation.

$$A = \min_{i=1,\ldots,H} A_i \text{ [15]}$$

The hop having the minimum available bandwidth is the *tight link* of the end-to-end path [15] [34]. Figure 3.1 below shows the difference between the network path capacity and the available bandwidth. Each link in this figure is represented as a pipe. The width is the capacity of the pipe; the unshaded area determines the spare capacity while the shaded area is the utilized part of the link capacity for each link.

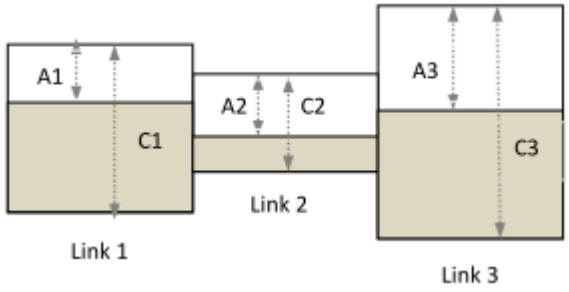


Figure 3.1: Pipe Model for Available Bandwidth [15]

As shown in Figure 3.1, the minimum link capacity is C2 (narrow link) and A1 (tight link) is the minimum available bandwidth at a given point of time. Thus C2 determines the available end-to-end network path capacity while A1 determines the available bandwidth of the end-to-end network path. As noted, the tight link of the path may not be the same as the narrow link of a path.

Thus it is important to measure both, the average available bandwidth and the path capacity as performance metrics. The available bandwidth is time variant and depends on the nature of the traffic (burstiness or long-range) whereas the capacity of the path usually stays constant unless the underlying link or the propagation medium changes. Two network paths over different IP protocols (IPv4 and IPv6) with same capacity may have different available bandwidths at a given point of time.

3.2.3 TCP Throughput and Bulk Transfer Capacity (BTC)

Capacity and Available bandwidth are defined independent of protocols used to transfer data. However, protocols like TCP/UDP/ICMP/IGMP and others

are used at the transport layer. They play an important role to determine the bandwidth offered to an application. Of particular interest is the TCP throughput since TCP is a major transport layer protocol carrying almost 90% of the Internet traffic [17]. The term throughput, in simple terms, is defined as the number of bytes transferred over a network path during a fixed amount of time [20].

There are several factors that affect the throughput of a TCP connection over an end-to-end network path. These factors include transfer size, which may be further influenced by the underlying IP protocol used (IPv4 or IPv6), number of TCP connections, cross traffic (TCP or UDP), socket buffer size at both the sender and the receiver, congestion, router buffers and other factors [15]. TCP throughput is further influenced by initial congestion window size, slow start and the Round Trip Time (RTT). Hence TCP throughput may vary significantly even if the underlying available bandwidth remains constant.

The *Bulk-Transfer-Capacity (*BTC) is defined as the maximum achievable throughput for a single TCP connection. It takes into consideration the TCP congestion algorithm as defined in reference [34]. BTC should not be related to the available bandwidth as it is a protocol dependent and also depends on how the network is configured to share TCP bandwidth with other TCP connections/Flows.

This thesis focuses on measuring both, the end-to-end capacity and also the Bulk-Transfer-Capacity for a TCP connection as part of a single measurement tool. The user is provided with the option to measure either metric for either IP

protocol as per the need. The next section discusses the measurement techniques for capacity and BTC.

3.3 What Are Different Bandwidth Measurement Techniques?

Previous sections provided brief descriptions of different types of bandwidths. This section explains in brief two of the established bandwidth measurement techniques to measure the capacity and BTC. Before going into the details of each of these techniques, let us first understand the different factors on which these measurement techniques can be classified.

The first classification is based upon the measurement technique or the algorithm used. Various measurement techniques are used by different tools. There has been lot of research to improve the current techniques and also develop new ones to accurately measure the network performance. Each technique measures a particular bandwidth metric. This section outlines two such techniques for bandwidth measurement used in the measurement tool.

The second classification is based upon the type of environment, either cooperative or uncooperative. The cooperative environment allows running the measurement tool on both the sides of network path, which may or may not require super user privileges. Most of the measurement tools require a cooperative environment for accurate measurement. Tools like Pathload, Pathrate, Iperf are a few of the examples that need the cooperative environment. The uncooperative environment is one in which the measurement tool is only deployed locally on the measurement host [20]; also called the source. An

example of such a tool is Sprobe that sends a few packet pairs like TCP SYN packets to an uncooperative remote host to reply with TCP RST packets. The bandwidth is measured by packet pair dispersion in the forward path.

The third classification is based upon whether the tool is active or passive in probing. An Active probing measurement tool generates its own traffic to measure the bandwidth or latency where as Passive probing relies on the application data to be used as the packet traces. The passive probing tools are non-intrusive to the network as these tools generate very light traffic to estimate the bandwidth. The active probing tools may or may not be intrusive depending on the measurement technique used. Most of the measurement tools generate their own traffic to accurately measure the bandwidth.

The fourth classification is based on the type of Bandwidth metric measured. The tool can measure the capacity, available bandwidth, TCP throughput (BTC) or a combination of these Bandwidth metrics.

The fifth classification is based upon the different layer 4 protocols used to measure the bandwidth or latency. UDP or ICMP are commonly used for latency measurement whereas both TCP and UDP are used to measure the bandwidth based on the type of Bandwidth metric measured.

Researchers over the past several years have sought to develop techniques to accurately measure the bandwidth. Each such technique measures a particular type of bandwidth. The need to measure network performance has become increasing important especially with high use of peer-to-peer networks for file transfer, overlay networks along with the advent of technologies such as Big Data,

which are involved in large dataset transfer, and of course social media like Youtube, Facebook and many others. Social media users are consuming more content online (Internet) than ever. To add to this, data consumption from various new gadgets like tablets and smart phones has added to the complexity of network traffic. The need for tools to provide better network administration and utilization to be network aware (netaware) in order to offer the best possible service to the end users has received more attention than ever. Thus understanding various bandwidth measurement techniques is crucial in building a measurement tool. Below are listed some different established measurement techniques [15].

1. Packet Pair/Train Dispersion (PPTD) probing
2. Throughput Measurement
3. Self-Loading Periodic Streams (SLoPS)
4. Variable Packet Size (VPS) probing

However, there are many variations of these techniques depending on various factors. These factors are based upon the mechanism used for data filtering, different statistical methods used, size and frequency at which packets are sent and various other combinations. The paper on Bandwidth estimation and Measurement techniques by authors R S Prasad, M Murry, C. Dovrolis K. Claffy [15] and the online repository of various measurement tools by the Cooperative Association for Internet Data Analysis (CAIDA) provides more details on different bandwidth measurement techniques and the taxonomy of many open

source tools respectively. We focus on the Packet Pair/Train Dispersion (PPTD) and the Bulk Transfer Capacity (BTC) measurement techniques as these are well established and proven techniques to measure end-to-end capacity and TCP throughput respectively [15] [22]. These two techniques are used and implemented in the measurement tool *netaware* built as part of this thesis.

1. *Packet Pair/Train Dispersion (PPTD) probing:*

The *Packet Pair/Train Dispersion* technique measures the capacity of an end-to-end network path. The idea is to send multiple packet pairs of the same size back-to-back from sender to receiver (also termed as source to sink). As the packet pair traverses from source to sink, the packet pair encounters dispersion delay at each link. The dispersion of a packet pair at each link is the time distance between the last bit of each packet. The Figure 3.2 below shows the packet pair dispersion technique as the packets go through a link of Capacity C*i* [15].

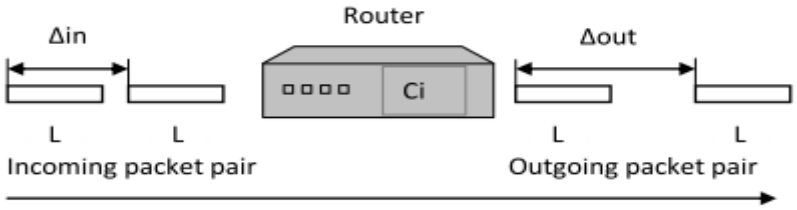


Figure 3.2: Packet Pair Dispersion [15]

The assumption here is that a link does not carry any cross traffic. The dispersion of packet pair of size L through a link of capacity C*o* is given as Δo =

L/C*o* [15]. For a network path with multiple links, the total dispersion ΔR from multiple links that a receiver will measure is as below [15]

$$\Delta R = \max_{i = 0,\ldots,H} (L/C_i) = L/\min_{i=0,\ldots H}(C_i) = L/C \text{ [15]}$$

Thus a receiver receiving the packet pairs can estimate the capacity C of an end-to-end network path as C= L/ΔR with the assumption that there is no cross traffic [15]. However in reality, the presence of cross traffic at different links can either cause erroneous estimation that ends up either overestimating or underestimating the capacity. For example, overestimation can occur if the cross traffic delays the first probe packet of a packet pair more than the second packet at a link that follows the paths narrow link. Various techniques have been used to remove the effect of cross traffic. For example, statistical methods are used to filter out erroneous measurements (median or mode), Packet Bunched Method uses local modes in the packet pair bandwidth distribution, variable sized packets are used to reduce the intensity of sub-capacity of local modes. Details of these techniques and more on *PPTD* can be found in reference the papers by R S Prasad, M Murry, C. Dovrolis K. Claffy and K. Lai, M. Baker [15] [21].

Packet train probing is an extension of the packet pair technique where multiple packets are sent back to back. The dispersion in this case is the time difference between the last bit of first and the last packets. The dispersion rate measured is equal to the path capacity with no cross traffic. *PPTD* is active probing technique that requires Cooperative environment for the tool to be run at

both ends (source and sink). It can also work in uncooperative environment by forcing receiver to send some form of error message like ICMP port-unreachable or TCP-RST packets in return for any packet sent [15]. The measurement tool *netaware* built as part of this thesis uses an open source algorithm *Pathrate* to measure the end-to-end capacity. It combines the use of statistical methods and variable sized packets to reduce the effect of cross traffic for better accuracy. Chapter 6 provides more details the *Pathrate* algorithm and the use of *PPTD* to measure the capacity.

2. Throughput Measurement*:*

The Throughput Measurement technique is used to measure the throughput of a path, or more specifically of a TCP connection. This technique is used to measure the Bulk Transfer Capacity (BTC) of a TCP connection. Throughput is measured in Mbps by counting the number of bytes received within a specified time interval. The formula is defined below where ΔT is the time interval.

Received Throughput = Number of bytes received * size of (byte)/ΔT

However, as mentioned in Section 3.2, BTC is affected by various factors including size of the data transfer, sender and receiver buffer size, TCP algorithm used and various other factors. Researchers over the years have developed many techniques to minimize the effect of these factors to accurately measure throughput of a TCP connection. Many commercially available tools like

Speedtest.net measure TCP throughput using parallel HTTP-TCP connections over a web browser. They use their own filtering mechanism to increase the accuracy [23]. As part of this thesis, a simple TCP throughput (BTC) measurement technique is developed which provides an average Bulk Transfer Capacity of a single TCP connection.

The next chapter provides a detailed design and implementation of the middleware application built as part of the thesis. Chapter 6 provides a detailed design and implementation of the measurement techniques built as part of the measurement tool to measure bandwidth and latency. The measurement tool utilizes the middleware application tool to transfer and receive the data for performance measurement.

Chapter 4 – Design and Implementation of A Middleware Application

This chapter provides a detailed design and documents the implementation of a middleware application built as part of this thesis. The first section provides design considerations of the middleware application followed by an overview of using sockets to be able to use in the third section. The third section discusses the middleware application design and its implementation along with the client server communication to switch between IP protocols.

4.1 Middleware Application Design Considerations

The middleware application developed as part of this thesis is intended to provide an easily integrated set of tools and functions for developing network-based applications that are capable of utilizing either IPv4 or IPv6 protocol. Every network-based application apart from its intended functionality needs the functionality or function calls to communicate with another process or program on the same host or a different host across a network. Most of these network applications are traditional client and server applications, where an application running on a client host communicates with another application on a server host [24]. A simple example of such an application would be a web browser (client) communicating with a web server to access a page or a File Transfer application where the client either downloads a file from or uploads a file to a server. Figure 4.1 gives an overview of the client and server communication using the TCP

protocol. As seen in the figure, the client application relies on the underlying communication protocols like TCP/IP, available within an operating system (OS) kernel to communicate with a server application running somewhere on a network.

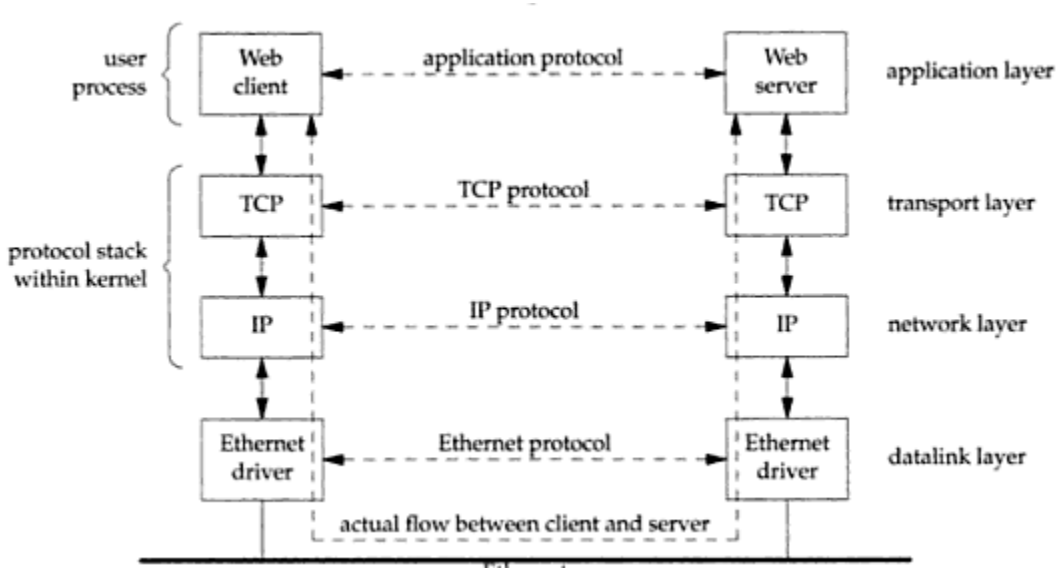


Figure 4.1: Client and Server Communication Using TCP/IP [24]

The middleware application developed facilitates the construction of such network applications, also called network programs [24], by having an application to deal with minimal code to use either IPv4 or IPv6 protocol. The middleware application includes the middleware application functions at the core of its design. These middleware application functions internally utilize the socket Application Programming Interface (API) provided as part of the operating system (OS) stack in the form of system calls. These system calls interact with the communication protocol stack provided within the OS kernel. During the design phase of this project, consideration was given to making these middleware

application functions modular enough to be able to easily integrate within an existing application and also to build new applications. Section 4.3.1 provides more details on the various middleware application functions developed.

The middleware application, through these middleware application functions, also provides the capability of switching between the IPv4 and IPv6 protocols interchangeably. This is done, on the fly, and without impacting application's functionality. The motivation behind building this capability is to leverage the existence of both IP protocols stacks (dual stacks) for better network utilization and to gain better performance at different points in time for an application. The focus here is on the TCP/IP protocol suite. The switching capability is built for this suite as it is one of the most widely used protocols for applications. TCP/IP has a rich set of features like congestion control, reliability, and in order transmission that helps build robust communicating applications [4]. Since TCP is a stream based, connection-oriented protocol, once a TCP connection is established after initial handshaking, data is transmitted in segments over the same established connection [4]. The application using TCP needs to have additional logic to delineate the data. This mechanism is different as compared to the UDP protocol, where in there is no initial hand shaking to establish a logical connection before hand, and packets are not acknowledged. Each UDP packet is forwarded independently and unrelated to the previous or following UDP packets [4]. It should be noted that for TCP, switching between the IPv4 and IPv6 protocol connections on the fly requires extra client-server communication. The middleware application internally takes care of this

communication while switching between the IP protocols, without impacting the application's functionality. The next section provides a brief overview of sockets and the basics of the system calls needed to implement the middleware application design while Section 4.3 provides more details on the middleware application design.

4.2 *What are Sockets?*

Sockets are one of the means of Interprocess Communication (IPC) that supports communication between independent processes on the same host or on a different host across a network [25]. Sockets are represented by descriptors and are end points of an IPC, providing bidirectional communication between hosts. These are also called *Internet sockets* as most of the communication between host computers happen on the Internet, which includes both public and private networks [26]. A socket is bound to an end point. The combination of an IP address and a port number is used to identify an end point. The combination of the socket and port number identifies a process/program running on a host. The socket API is provided by the underlying OS, which allows application programs, or network programs, to control and use these network sockets. There are many socket APIs, namely Linux/POSIX, AT&T and Windows, based on the underlying OS, but all of them are variants of the Berkeley Software Distribution (BSD) sockets that is a de facto standard [24]. The middleware application developed as part of this thesis is currently based on the Linux/POSIX standards, but can be easily converted for other socket APIs/OSes. The socket API provides various

system calls that interact with the OS kernel to perform I/O operations for reading and writing the data. Table 4.1 lists the basic system calls. Appendix B provides a more detailed description of these individual system calls and the corresponding middleware application functions that use these system calls.

SYSTEM CALLS	DESCRIPTION
socket()	Creates a socket and return socket descriptor
close()	Closes and destroys a socket
bind()	Labels a server socket with an address (IP address : port no)
connect()	Creates a connection between sockets
listen()	Configures a socket to accept connections
accept()	Accepts connections and creates a new socket for the connections
read()	Read data from the input stream available on the file/socket descriptor
recv()	Receive data from the input stream available on the socket (TCP)
recvfrom()	Receive data from the input stream available on the socket (UDP)
write()	Write data to the output stream to the file/socket descriptor
send()	Send data from the input stream available on the socket (TCP)
sendto()	Send data from the input stream available on the socket (UDP)
setsockopt()	Set the options associated with a socket
getsockopt()	Get the value of the options associated with a socket

Table 4.1: Socket API System Calls [25]

4.3 Middleware Application Design:

The middleware application comprises of set of middleware application functions and constants. A shared library named *libnetaware.so*, is built to hold the compiled version of the middleware software functions. A shared library is a complied version of functions into a single archive file that can be dynamically linked to by an executing program that refers to these functions during execution. These functions can be shared among processing applications and do not require multiple copies of the code to be loaded into main memory. An application using

the middleware application needs to include the header file named "commonhdr.h" in its code. A header file is an interface that contains the declarations (functions prototypes, symbolic constants, conditional compilation directives and other non-prototype definitions) to be shared and included among several source files. As the name suggests, "commonhdr.h" is a common header file, that provides the function prototypes, symbolic constants and conditional compilation directives of the middleware application needed to link to the shared library routines during execution. Once an application that utilizes the middleware application is built and ready, it can then access the shared library *libnetaware.so* at run time. The design of the developed middleware application can be divided into three sections. The first section describes the various middleware application functions developed; their function calls and return types. The middleware application functions provide a simple building block for an application and are modular to be used in any network application as explained later. The second section describes the design of the client-server communication to allow switching from IPv4 to IPv6 or vice versa in a multi-threaded environment. The use of threads helps the client-server communication to take place as a separate activity in the background without impacting the application's functionality. For example, for a file transfer application built using the middleware application, the use of threads on each side, helps to perform the client-server communication in parallel without impacting the functionality of transferring a file in the main thread. The use of threads in client-server communication plays a key role in the switching technique as explained further in

Section 4.3.2. The third section explains two different ways to communicate to an application using the middleware application to switch between the IP protocols.

4.3.1 Middleware Application Functions:

The socket API provides various system calls for inter-host, client-server communication across a network. The important and frequently used ones are listed in Table 4.1. Table 4.2 provides a list of middleware application functions, their purpose, input parameters and corresponding return types. Based on the parameters passed to a middleware application function, the middleware application function then internally make use of the corresponding socket API system call to operate on either IPv4 or IPv6 socket ids. For example, the middleware application function *fn_send()* internally calls the socket API system call *send()* to achieve the same functionality of sending data over sockets but with added logic that determines which IP protocol (IPv4 or IPv6) is to be used based on its input parameter values. Similarly, the middleware application function *fn_recv()* makes use of the system call *recv()* to receive data over either an IPv4 or IPv6 socket id. The middleware application is currently built to work with both TCP and UDP. A detailed description of each of the middleware application functions can be found in Appendix B.

Functions	Purpose	Input Parameters	Return type
fn_socket()	Creates sockets and then binds a socket to an address:port and also connects to the server. If the application is a server, it listens for an incoming connection	st_mySocket socket structure, int socket type, int IP version	int: Error type. 0 on success
fn_accept()	Accepts incoming connection for TCP and creates a new socket descriptor	st_mySocket *socket structure	int: New socket descriptor on success. -1 on error
fn_bind()	Binds socket descriptor to the address:port	st_mySocket *socket structure, int IP version	int: -1 on Error,errno is set accordingly. 0 on success
fn_connect()	Connects to the server	st_mySocket *socket structure, int IP version	int: -1 on Error,errno is set accordingly. 0 on success
fn_listen()	Listens on incoming connection for a TCP connection	st_mySocket *socket structure, int IP version	int: -1 on Error,errno is set accordingly. 0 on success
fn_close()	Closes the connection on a socket	st_mySocket socket structure	int: -1 on Error,errno is set accordingly. 0 on success
fn_write()	Writes data to a connection	st_mySocket *socket structure, void *buffer, size_t length, int flag	size_t: number of bytes sent. -1 on error
fn_send()	Sends data (TCP: connection based)	st_mySocket *socket structure, void *buffer, size_t length, int flag	size_t: number of bytes sent. -1 on error
fn_sendto()	Sends data (UDP: connectionless)	st_mySocket socket structure, void *buffer, size_t length, struct sockaddr *toaddr, int flag	size_t: number of bytes sent. -1 on error

fn_read()	Reads data from a connection	st_mySocket *socket structure, void *buffer, size_t length, int flag	size_t: number of bytes sent. -1 on error
fn_recv()	Receives data (TCP: connection based)	st_mySocket *socket structure, void *buffer, size_t length, int flag	size_t: number of bytes received. -1 on error
fn_recvfrom()	Receives data (UDP: connectionless)	st_mySocket socket structure, void buffer, size_t length, struct sockaddr *frmaddr,int flag	size_t: number of bytes received. -1 on error
fn_setsockopt()	Set socket options	st_mySocket *socket structure, int IP version, int opt level, int opt name, const void *opt val, socklen_t size	int: -1 on Error, errno is set accordingly. 0 on success
fn_getsockopt()	Get socket options set	st_mySocket *socket structure, int IP version, int opt level, int opt name	int: -1 on Error, errno is set accordingly. 0 on success

Table 4.2: Middleware Application functions

The structure named *st_mySocket* is the most important structure used by all middleware application functions used to build applications with the developed middleware application services. This structure is required as an input parameter for all middleware application functions. It is a structure that holds the socket descriptors, socket type, IP version, port numbers, host addresses and other information for both IP protocols. This structure can be used for creating socket connections for either TCP or UDP. It encapsulates the connection properties of both IPv4 and IPv6 connections and can be treated as a socket object for both of the IP protocols. The application using the middleware

application has to initialize a few of the structure members once and then use it wherever the middleware application functions are called in an application. The structure members that need to be initialized with values are the members holding the port numbers, host name, the preferred IP protocol to be used and a member variable that indicates whether an application is a client or a server. A set of constants having meaningful names are included as part of the middleware application to ease the socket structure initialization and also for the internal use of the middleware application. A complete list of middleware application constants can be found in Appendix B. Initializing the socket structure with appropriate values and using it across all the middleware application functions helps an application to deal with minimal code while using sockets. Creating and using socket connections requires an application to use socket API calls with specific parameters for each IP protocol. If the application desires to use both IP protocols, it would need to have separate socket API system calls within its code with individual parameters for both IP protocols. The middleware application provided here helps ease the use of sockets by initializing socket structure once and using it across all the middleware application functions. Based on the values assigned to the socket structure, the middleware application internally takes care of creating and using socket connections for the IPv4 or IPv6 protocol. This allows developers to focus on the core functionality of an application. For example, the application just needs to call the middleware application functions *fn_read()/fn_write()* with the socket structure as one of the input parameters to read and write data respectively using either IP protocol. This would otherwise

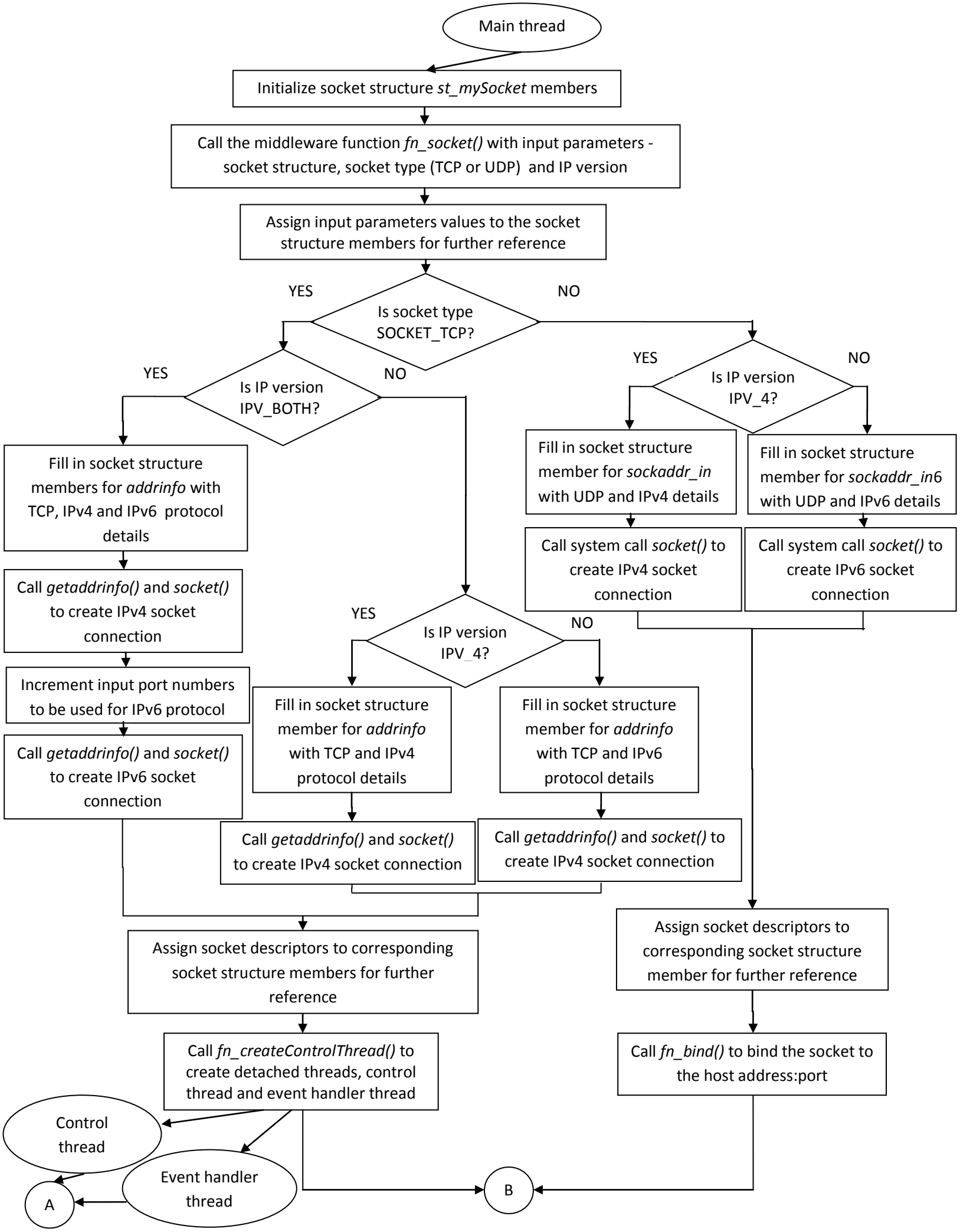
require two separate system calls *read()* and *write()* for IPv4 and IPv6 respectively. The same applies through the use of the other middleware application functions listed in Table 4.2. Apart from the member variables that need to be initialized by an application, the structure also holds other member variables used for the internal working of the middleware application. The middleware application takes care of assigning and changing the values of these remaining structure members once the socket structure is passed to it. The socket structure st_*mySocket* and all its members are outlined in Appendix B.

The starting point of using the middleware application is the middleware application function *fn_socket()*. As with any network application that calls the socket API function *socket()* as the first function call, *fn_socket()* also needs to be the first middleware application function to be called to use the middleware application version of the API. This applies to an application running as either a server or a client. This middleware application function is responsible for internally assigning values to the remaining members of the socket structure st_*mySocket* passed to it. Based on the parameters passed to this function, it creates a socket connection for either IPv4 or IPv6 or both IP protocols. Once the socket connection(s) is/are created by *fn_socket()*, the socket properties like the socket ids, and remote host addresses are stored in the socket structure st_*mySocket*. Once the call to *fn_socket()* is successful and the socket connections are created, the socket structure can then be used by other middleware application functions to operate using either IP protocol. In order to use the middleware application capability to switch between the IP protocols, the

function *fn_socket()* needs to create socket connections over both IP protocols. This occurs by initializing socket structures with appropriate values before calling *fn_socket()*. These values then mark the socket object, to be available for switching the IP protocol. The application can choose to either use or not to use the middleware application switching capability by controlling the values assigned to socket structure members. By default, the middleware application starts by using the IPv4 protocol until the middleware application is told to switch to the IPv6 protocol. The application developer also has an option to specify the preferred IP protocol to be used. The middleware application internally keeps track of which protocol is to be used. Both IPv4 and IPv6 socket connections are maintained throughout the application, as making and breaking of connections causes extra overhead while switching IP protocols. Moreover the application can also specify to strictly create and use either an IPv4 or an IPv6 network connection. In case *fn_socket()* is not able to create a socket connection for the specified protocol, it returns an appropriate error message. Figure 4.2 provides a high level flow diagram of the internal working of *fn_socket()*. The middleware application function *fn_socket()* further helps an application to have minimum code by internally calling the middleware application functions *fn_bind()*, *fn_listen()* and *fn_connect()* for TCP. For a server application, the functions *fn_bind()* and *fn_listen()* are called to bind a socket descriptor to an address:port pair and listens for incoming connection calls respectively. For client applications, the function *fn_connect()* is called internally to connect the client socket to the one on the server. Appendix B provides a detailed example on

how to initialize the socket structure and use of middleware application functions to use the desired IP protocol. It also provides more details on the use and significance of each of the socket structure member variables.

The middleware application provides code modularity in two ways. The middleware application functions have the function name, input parameters and the return type similar to that of the base socket API system calls. This is to make it easy to integrate the middleware application in existing applications or while building a new application. The second way code modularity is achieved is through the use of a shared library. In case the internal working of any of the middleware application functions is changed, the application using the middleware application does not need to be changed or recompiled. The shared library just needs to be recompiled to be available for use. The next section explains the client server communication involved to switch between the IP protocols for TCP connection in the main application thread.


Main thread
Initialize socket structure st_mySocket members
Call the middleware function fn_socket() with input parameters - socket structure, socket type (TCP or UDP) and IP version
Assign input parameters values to the socket structure members for further reference
Is socket type SOCKET_TCP?
YES
NO
Is IP version IPV_BOTH?
YES
NO
Is IP version IPV_4?
YES
NO
Fill in socket structure members for addrinfo with TCP, IPv4 and IPv6 protocol details
Call getaddrinfo() and socket() to create IPv4 socket connection
Increment input port numbers to be used for IPv6 protocol
Call getaddrinfo() and socket() to create IPv6 socket connection
Fill in socket structure member for sockaddr_in with UDP and IPv4 details
Fill in socket structure member for sockaddr_in6 with UDP and IPv6 details
Call system call socket() to create IPv4 socket connection
Call system call socket() to create IPv6 socket connection
Is IP version IPV_4?
YES
NO
Fill in socket structure member for addrinfo with TCP and IPv4 protocol details
Fill in socket structure member for addrinfo with TCP and IPv6 protocol details
Call getaddrinfo() and socket() to create IPv4 socket connection
Call getaddrinfo() and socket() to create IPv4 socket connection
Assign socket descriptors to corresponding socket structure members for further reference
Assign socket descriptors to corresponding socket structure member for further reference
Call fn_createControlThread() to create detached threads, control thread and event handler thread
Call fn_bind() to bind the socket to the host address:port
Control thread
Event handler thread
A
B

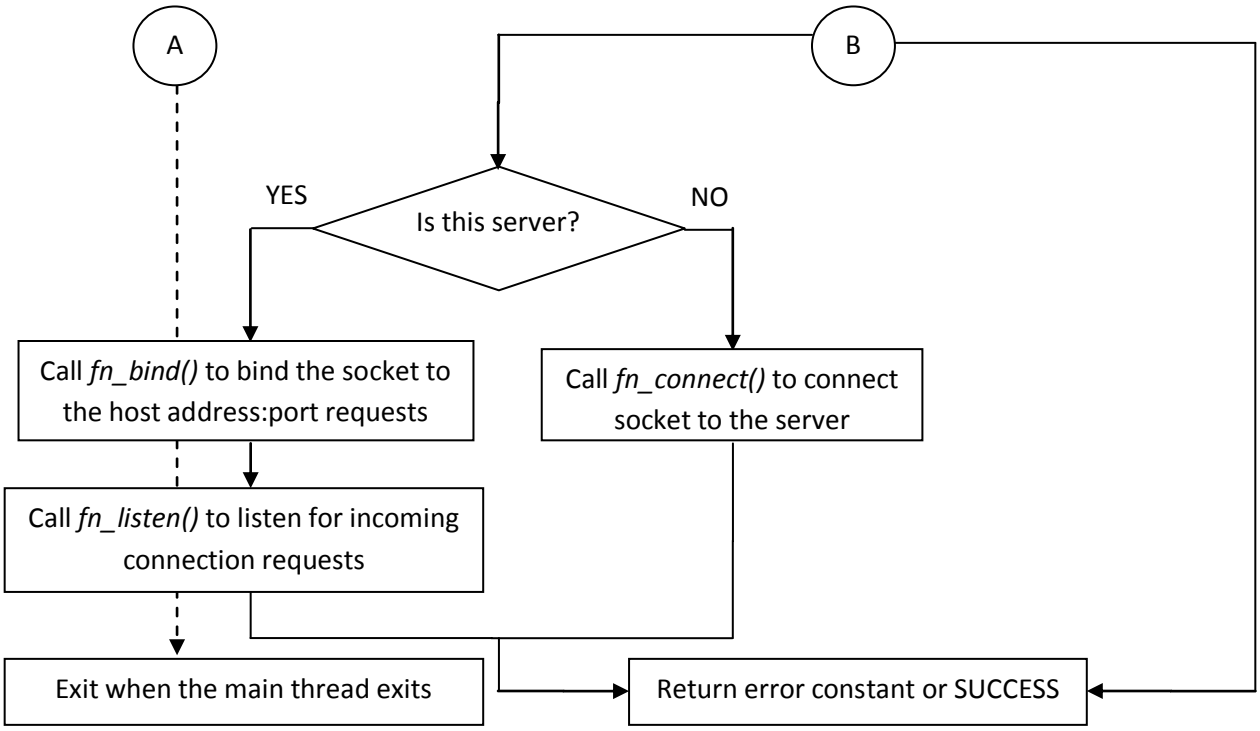


Figure 4.2: *fn_socket()* Flow Diagram

4.3.2 Client Server Communication

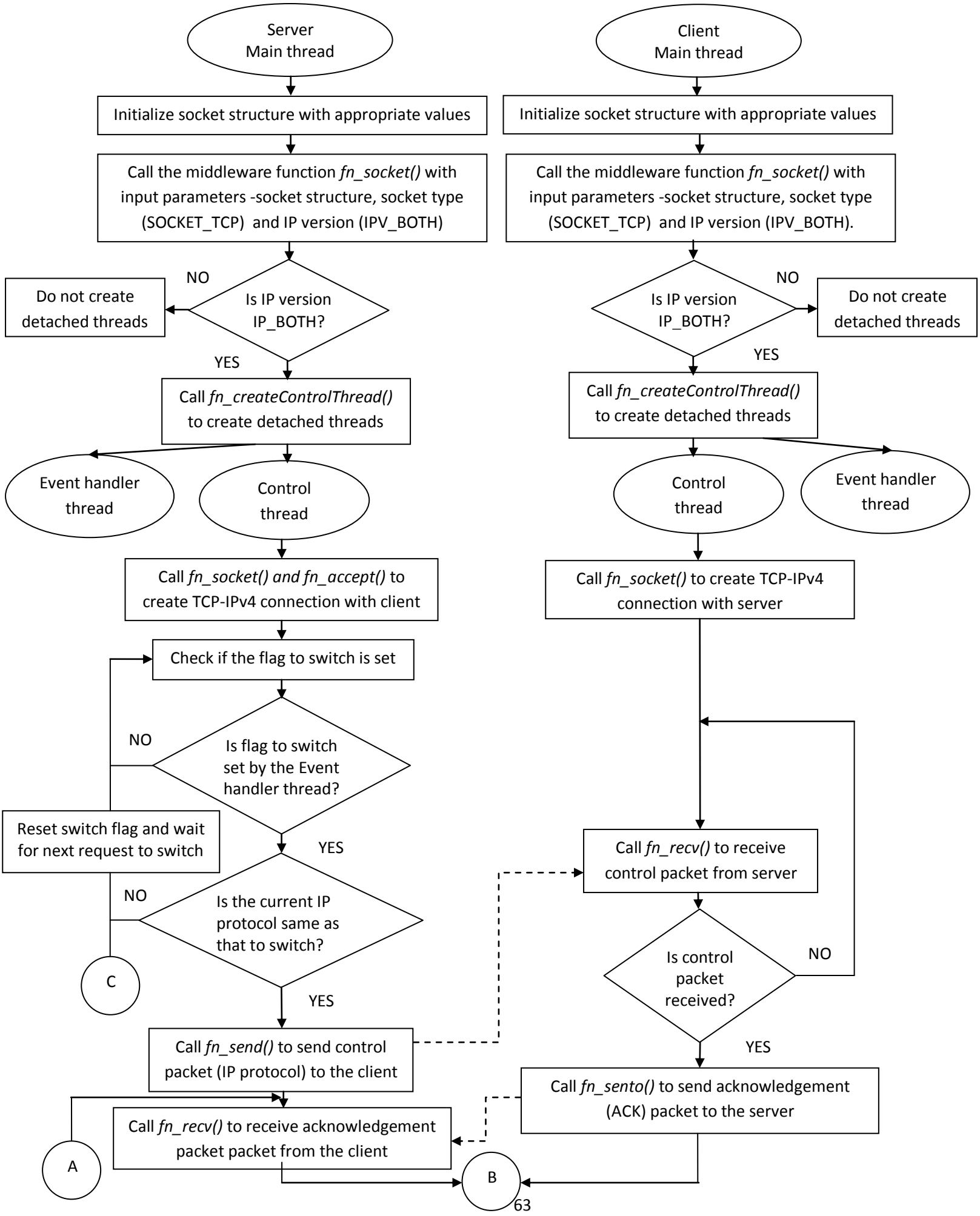
As discussed in previous sections, for a TCP connection, data is read and transmitted in segments, which is a byte stream as opposed to sending packets that have defined boundaries [4]. A single read operation at a receiver socket may or may not yield an entire message as it was sent. In terms of network programming, both IPv4 and IPv6 protocols need separate connections and corresponding socket descriptors, be it TCP or UDP or any other application layer protocol. Each s*end()/recv()* and *write()/read()* call is an Input/Output (I/O) operation that requires the use of socket descriptors to send and read data over a socket connection. Thus switching between IP protocols means to switch from one socket connection to the other through the use of corresponding socket descriptors. Hence there is a need for the client server communication to request

the change in socket descriptor to operate on in order to switch the IP protocol. Moreover since TCP transmits data in segments, there needs to be a clear demarcation between end of data stream on one socket connection and start of data stream on the other socket connection. This demarcation is needed in order to gracefully switch between established connections to avoid loss of data and also to receive data in the order it was sent. All this is achieved through the combined use of the middleware application functions *fn_socket(), fn_send()/fn_write()* and *fn_recv()/fn_read()* and the protocol to be followed while switching.

Figure 4.3 below shows a high level flow diagram of the client server communication when the server receives a request to switch. Either the server or client can initiate a request to switch the IP protocol. The function *fn_socket()* spawns two detached threads, namely the event handler thread and the control thread at the end of its function call. These threads are POSIX threads also known as pthreads and run until the main application thread exits. Both these threads are created only if the socket structure *st_mySocket* is marked to create both IP connections. The main purpose of the control thread on both the sides is to exchange control information between the client and the server while switching the IP protocol when a request to switch is received. This communication happens in the background without interrupting the functionality of the application in the main thread. The control thread on the server maintains its own TCP socket connection with the control thread on the client side to exchange control information. The control information is a control

packet. This packet contains either the IP protocol to switch to based on the request received or the acknowledgement of this control packet received from the control thread on the other side. It is a small packet of the size of an integer. The control thread keeps a check on the flag *vg_ctrlpacket_sent* that indicates that a request to switch was received. This is a global flag accessible to all the threads. The event handler thread on both the sides is responsible to receive the request to switch the IP protocol in form of an event. Once the event is received, the event handler thread sets the global flag *vg_ctrlpacket_sent*. More on the event handler thread, how the flag is set and different ways on how to request an application to switch to either IPv4 or IPv6 protocol is described in the next section. Once the flag *vg_ctrlpacket_sent* is set, the control thread sends a control packet over the TCP connection to the control thread on the other side to initiate a request to switch the socket descriptor. This control packet contains the IP protocol version to switch to based on the request received. After receiving the control packet, the control thread on the other side then sends back an acknowledgement packet over the same TCP connection. The control thread on each side keeps track of the arrival of these control information. Depending on whether the server or the client is sending the data in the main thread, this control information is exchanged to initiate the switch. For example, when the event handler thread on the server sending data to the client, receives a request to switch the IP protocol, it simply switches the socket descriptor to stop sending the data on one socket connection (IP protocol) and starts sending the data on the other. In case the server is receiving the data from the client, the server side

control thread sends a control packet to the client side control thread to initiate the change in the socket descriptor sending the data. When received, the client side control thread sends back an acknowledgement packet, ACK, to the control thread on the server side. The control thread on server then correspondingly sets another global flag *vg_ctrlpacket_recv*. This flag indicates a switch in IP protocol and is to be checked by the middleware application functions in the main thread as explained further. Once the server side control thread receives the ACK, it then resets the global flag *vg_ctrlpacket_sent* previously set by the event handler thread when the request was received. This flag is reset to indicate that the control thread on the client side has received and honored the request to switch the IP protocol and to further avoid sending the control packets. This flag is again set by the event handler thread when the next request to switch is received. The same mechanism to exchange the control information and correspondingly change the socket descriptor is followed when the client receives a request to switch the IP protocol.


Server
Main thread
Initialize socket structure with appropriate values
Call the middleware function fn_socket() with input parameters -socket structure, socket type (SOCKET_TCP) and IP version (IPV_BOTH)
Is IP version IP_BOTH?
NO
Do not create detached threads
YES
Call fn_createControlThread() to create detached threads
Event handler thread
Control thread
Call fn_socket() and fn_accept() to create TCP-IPv4 connection with client
Check if the flag to switch is set
Is flag to switch set by the Event handler thread?
NO
Reset switch flag and wait for next request to switch
YES
Is the current IP protocol same as that to switch?
NO
C
YES
Call fn_send() to send control packet (IP protocol) to the client
Call fn_recv() to receive acknowledgement packet packet from the client
A
B
Client
Main thread
Initialize socket structure with appropriate values
Call the middleware function fn_socket() with input parameters -socket structure, socket type (SOCKET_TCP) and IP version (IPV_BOTH).
Is IP version IP_BOTH?
NO
Do not create detached threads
YES
Call fn_createControlThread() to create detached threads
Control thread
Event handler thread
Call fn_socket() to create TCP-IPv4 connection with server
Call fn_recv() to receive control packet from server
Is control packet received?
NO
YES
Call fn_sento() to send acknowledgement (ACK) packet to the server

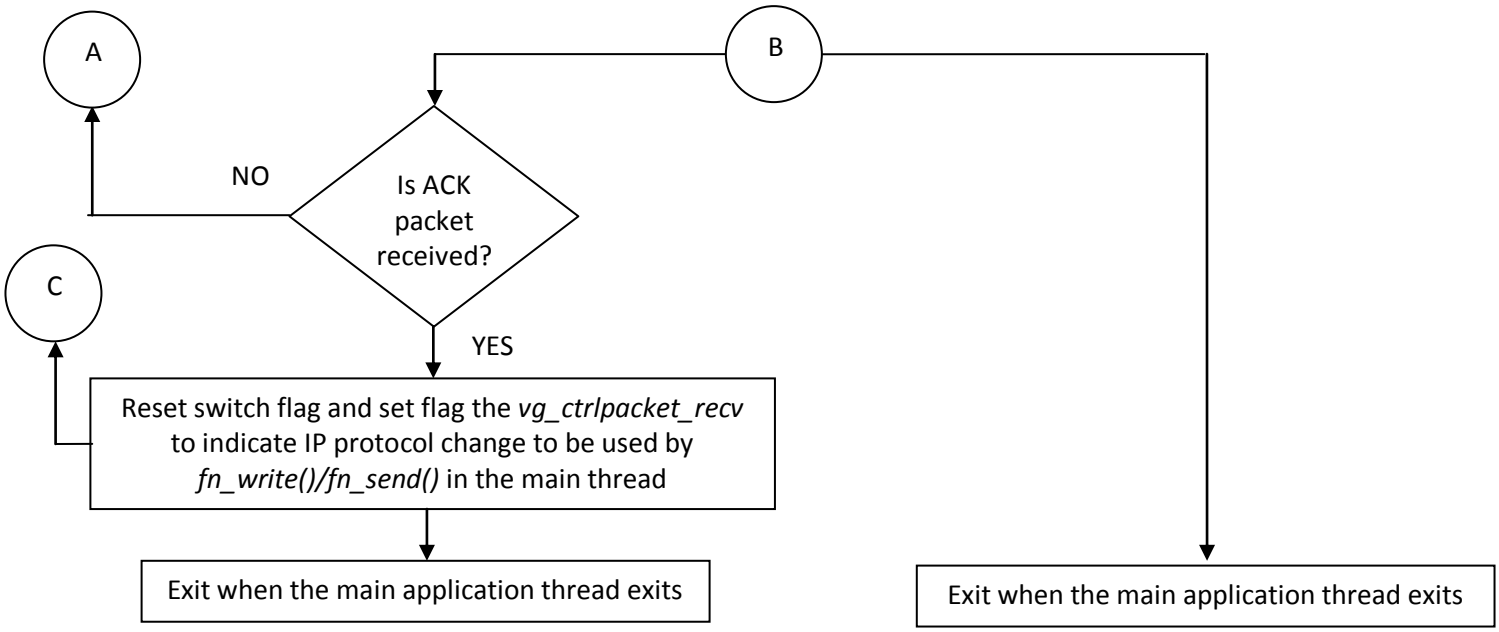


Figure 4.3: Client Server Communication

The global flag *vg_ctrlpacket_recv* set by the control thread is then used by the middleware application functions *fn_send()/fn_write()* in the main thread to switch the socket descriptor to use the desired IP protocol. As mentioned earlier, TCP transmits data in segments and hence there needs to be a demarcation to mark the end of data stream on one connection and start of data stream on the other in order to gracefully switch between IP protocols. This is achieved by constructing an application layer packet, each time the middleware application functions *fn_send()/fn_write()* is called to send data. The packet contains two fields, followed by the actual application data to be sent. The first field contains the total packet size including the application data. This field indicates *fn_recv()/fn_read()* the amount of data to be read in order to receive the entire packet. This means, *fn_recv()/fn_read()* internally calls *recv()/read()* multiple times until the entire packet is received. The value in the second field acts as a sentinel to demarcate the data while switching the IP protocol. Figure 4.4 shows the packet structure and size of each field on a 64-bit OS. The packet

size can however vary from OS to OS. *fn_send()/fn_write()* checks for the flag *vg_ctrlpacket_recv* set by the control thread each time it is called to send the data. Once set, it sets the value in the second field with a constant 'X' as a sentinel to mark the end of the data stream on current connection before switching the socket descriptor (IP protocol). The function *fn_recv()/fn_read()* called by the application thread across the network, checks for the value in this second field once the entire packet is received. The value X in this field indicates that no more data will be sent on the current socket connection and the socket descriptor needs to be switched to continue receiving the data on the other socket connection. *fn_recv()/fn_read()* then reads the last packet on current connection before switching the socket descriptor. The value 'X' as a sentinel is simply a design consideration, but can be any other constant value. Since the data is sent over structured packets that contains the total packet size and amount of data to be received in each read operation, *fn_recv()/fn_read()* is aware where the sentinel ends and actual data begins. This avoids the confusion while reading the data in case the data itself includes the string '_X_'. After the socket descriptor is switched to receive data on the other IP protocol, *fn_recv()/fn_read()* resets the flag *vg_ctrlpacket_recv*. Appendix B provides pseudo code of the middleware application functions *fn_send()/fn_write()* and *fn_recv()/fn_read()*. As seen from Figure 4.4, each *fn_send()/fn_write()* and corresponding *fn_recv()/fn_read()* call entails an overhead of 5 bytes for the first two fields of the packet. However, the percentage overhead may vary based on the application design and the amount of data sent per each *fn_send()/fn_write()* call. For

example, the percentage overhead for an application data of the size 1500 bytes is 0.33% while that for 500 bytes is 1%. Thus the higher the size of the application data per each *fn_send()/fn_write()* call, the lower the overhead. However, an application can choose not to use the switching capability and the corresponding overhead by passing the flag value NO_DATA_TX to the function *fn_send()/fn_write()*. Appendix B provides a working example using the middleware application functions.

In order to maintain synchronization while setting and resetting the flags in a multithreaded environment, the middleware application makes use of MUTEX locks. The client server communication, and packet creation to send and receive the data is transparent to the main application. The next section provides more details on the event handler that receives and handles the request to switch the IP protocol.

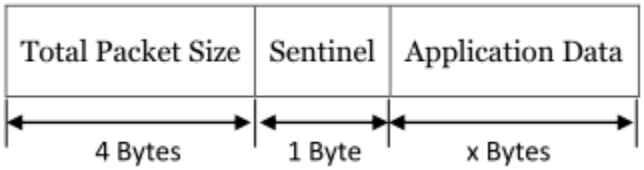


Figure 4.4: *fn_send()/fn_write()* Packet Structure

4.3.3 Event Handling to Switch IP Protocol

The previous two sections provided brief descriptions of the middleware application functions and their capability to switch between the IP protocols on the fly when a request is received. But the question is how to notify an application

to switch the IP protocol during its execution? This is achieved through the use of signals and a corresponding signal handler. The middleware application function *fn_socket()* creates a detached thread called event handler thread as shown in Figure 4.2. The event handler thread is a simple signal handler that checks on incoming requests in form of signals to set flags. These flags are set to indicate to switch the IP protocol and also to specify which IP protocol is desired based on the signal received. These flags are then used by the control thread and middleware application functions *fn_send()/fn_write()* and *fn_recv()/fn_read()* to switch to the socket descriptor of the desired IP protocol as described earlier. LINUX supports a wide range of POSIX and non-POSIX signals. Each signal has a default action that determines how a process behaves when that signal is delivered. The signal handlers are set to identify and handle received signals. In this application, the signal handlers are set in the event handing thread to identify two kinds of signals namely *SIGUSR1* and *SIGUSR2*. Apart from signal handlers set in the event handling thread, the middleware application also has a few other signal handlers to perform the tasks of cleaning up threads and resources when those signals are delivered. Table 4.3 below shows the signal numbers, default action and corresponding IP protocol to switch to when these signals are delivered. These signal values are obtained from the host ***web.rnet.missouri.edu*** and can change from OS to OS.

Signal	Value	Action	Comment	IP Protocol to switch to
SIGUSR1	10	Term	User-defined signal 1	IPv4
SIGUSR2	12	Term	User-defined signal 2	IPv6

Table 4.3: Signal Values and Action

SIGUSR1 and SIGUSR2 are user-defined signals and are not triggered from within the OS kernel. These are left for the developers to use programmatically as needed. The action value term in Table 4.3 means the default action is to terminate the process it is delivered to, unless it is caught and handled by a signal handler. When SIGUSR1 is received, the signal handlers set the flags to indicate to switch to IPv4 protocol while for SIGUSR2 it is set to use IPv6 protocol. These signals can be sent to an application while in execution by simply using command line utility ***kill*** as shown below. SIGUSR1 is the name of signal to be sent while pid is process id of the executing application. Instead of using SIGUSR1, its corresponding number can also be used. For example on ***web.rnet.missouri.edu***, the number 10 can be used instead of SIGUSR1.

```
kill -SIGUSR1 pid
```

Another way of sending signals is through the use of the system call named ***kill()*** from within the same or another application. An example below shows on how to use this option.

```
vlError = kill(pid, SIGUSR1);
```

pid is again the process id of the application the signal is to be sent to and *SIGUSR1* is signal name itself. This option is useful in the case where the IP protocol of an application like a file transfer application that transfers very large files across the network needs to be changed on the fly to enhance the transfer performance. A measurement tool running in parallel can measure the performance of the network path between the same hosts as that of the file transfer application. The measurement tool can send either SIGUSR1 or SIGUSR2 signals to the file transfer application based on measured performance, to switch to the desired IP protocol. A sample prototype is built to test the middleware application switching capability on receiving SIGUSR1 or SIGUSR2 from either a user through a command line terminal or from another program running on the same machine as that of the file transfer application. Thus with the combined use of the event handler thread, the control thread and the protocol followed by the middleware application functions while switching the IP protocol, an application using the middleware application can switch between IP protocols on the fly.

With the middleware application built as part of this thesis, the overall aim is to help applications leverage the existence of both IP protocols and gain performance by using an IP protocol offering better performance. The switching capability built as part of the middleware helps an application to gain this performance benefit by switching between the IP protocols on the fly, without impacting the application's functionality. Additionally, the use of middleware

application also helps an application deal with minimum code for interprocess communication across network using sockets. The application just needs to initialize few variables and pass it to middleware application functions. The middleware application then takes care of creating and using socket connections to use the preferred IP protocol. The measurement tool built helps to measure performance for both IP protocols as part of single tool. Based on measured performance, the application using the middleware application can be requested to switch the IP protocol if desired. The next chapter provides details on the design and implementation of this measurement tool, the algorithm used to measure each performance parameters and the corresponding client server communication involved. The tool uses the middleware application functions and constants whereever sockets are involved to transfer the data over both IP protocols as a proof of concept of the middleware application built.

Chapter 5 – Design and Implementation of Measurement Tool

This chapter provides a detailed design and implementation of the measurement tool built as part of this thesis. It is divided into four main sections and builds upon Chapter 3 that talked about different types of bandwidth and corresponding measurement techniques. The measurement tool provides tool specific implementations of these techniques. The first sections deals with the design considerations of the measurement tool. The next two sections provide detailed implementations to measure two different bandwidth metrics; namely, the capacity and the Bulk Transfer Capacity (BTC) respectively. The last section provides details on the latency measurement technique. All techniques are built and integrated as part of a single measurement tool.

5.1 Design Considerations

There are three main reasons to build the prototype measurement tool as part of this thesis. The first reason is to measure end-to-end performance between two hosts over both IPv4 and IPv6 protocols for comparative analysis. Both of the network stacks may or may not follow the same network paths based on the underlying network infrastructure. The motive is to provide a single tool to decide on a better IP protocol between the two to leverage the existence of a Dual IP stack in order to gain performance and better network utilization. Based on measured performance metrics, an application built using the middleware

application can then be run using either IPv4 or IPv6 protocol. For client-server applications involved in large data transfers across the network using sockets (TCP/IP) and taking considerable amount of time to complete transfer, the measurement tool can be especially be useful. This tool can be run as a separate client-server application between the same hosts involved in large data transfers to determine a better protocol between IPv4 and IPv6 at that point of time. The application can then use the switching capability of the middleware application as explained in Chapter 4 to switch to a better IP protocol on the fly if needed. The measurement tool can be run either before running the main application to choose an IP protocol to start with or in parallel as a separate application while the main application is in execution to switch the IP protocol on the fly through the use of the middleware application. The second reason and one of the design considerations is to integrate and measure various end-to-end network performance parameters explained in Chapter 3 combined together in a single measurement tool. Currently the tool is built to measure latency and two types of bandwidths namely path capacity and Bulk Transfer Capacity (BTC) for TCP. Measuring the capacity helps to gain information on the underlying network infrastructure from host A to host B as it is directly linked to layer-2 links [34]. Capacity remains constant until the underlying network infrastructure changes as explained in Section 3.1.1. Measuring BTC provides real time available bandwidth or TCP throughput offered to an application at a given point of time. BTC is a changing metric as is it influenced by many parameters as explained in Section 3.1.3. Two network paths having the same path capacity may offer different real

time bandwidth. The measurement tool can be run to measure BTC for both IPv4 and IPv6 protocols before the main application runs or while in execution to gain information on current traffic to help decide between the two IP protocols. Thus measuring capacity and BTC helps the application to be aware in terms of both; its constant maximum bandwidth and real time achievable bandwidth for both IP protocols. The third reason to build the tool is to demonstrate the use of middleware application in creating and operating on socket connections with minimal code for both IPv4 and IPv6 protocols as a proof of concept.

This tool can be classified as an active probing tool to be run in a cooperative environment. It is active probing because it generates its own traffic for performance measurement. It needs a cooperative environment because the tool requires access on both the end points of the networks; client-server communication to measure performance. End-to-end measurement may not be as accurate as compared to its counterpart like using counters maintained by routers, but it is the only feasible approach to measure performance of network paths that cross several networks that do not provide access to underlying hardware for measurement [29]. The design and implementation of the measurement tool can be divided in two sections namely latency measurement and bandwidth measurement. Both measurement techniques make use of UDP and TCP protocols in ways described in their corresponding sections. This tool is run through a command line terminal and does not require any special privileges. Either latency or bandwidth can be measured at one time and is one of the design considerations so as to not flood the network with probing packets. The user can

specify which parameter to measure as an option while running the tool. This measurement tool makes use of the middleware application functions explained in Chapter 4 to create and operate on sockets for both the IPv4 and IPv6 protocols. A simple example is accomplished by creating sockets using *fn_socket()* and performing I/O operations on those sockets using middleware application functions like *fn_send()/fn_recv()*. The socket structure *st_mySocket* created by *fn_socket()* is then further utilized for measurement technique. The measurement tool includes the middleware application header file "commonhdr.h" and link to the shared library *libnetaware.so* at run time to be able to use the middleware application functions and symbolic constants. It is linked to the middleware application shared library *libnetaware.so* at runtime. The next two sections provide more details on the measurement techniques for each of the performance parameters.

5.2 Bandwidth Measurement

As mentioned previously, the measurement tool measures two types of bandwidth; namely, capacity and Bulk Transfer Capacity (BTC) for a TCP connection. An open source measurement technique *Pathrate* that measures path capacity is modified and integrated with Bulk Transfer Capacity measurement technique in a single tool. This measurement technique is primarily based on the well-known *packet pair/Train Dispersion* (PPTD) algorithm to estimate capacity as explained in section 3.2.1. *Pathrate* is chosen to

estimate capacity for the following salient features and design considerations [29] [30].

1. Measures end-to-end capacity of a network path at the application level
2. Requires access to both ends of a network path, but does not require super user privileges
3. Robust to cross traffic interference
4. Based on a combination of packet pair and packet train technique for accurate estimation of capacity without being non-intrusive as compared to the other capacity estimation tool.
5. Follows POSIX standards and in line to use the middleware application for creating IPv4 and IPv6 connections.
6. Easy to configure and run the tool.

The next two sections provide more details on *Pathrate* and BTC measurement techniques.

5.2.1 Capacity Measurement (Pathrate)

Pathrate primarily uses the *packet pair/Train Dispersion* (*PPTD*) technique along with statistical techniques to discover the narrow link with minimum transmission rate. The transmission rate of the narrow link is the maximum rate seen by the network path and hence the end-to-end capacity [29]. The last equation in Section 3.2.1 shows the relationship between the narrow link and the capacity. *Pathrate* uses UDP packets for packet pairs and packet-trains while TCP

is used to exchange control information [30]. It requires access to both the ends of a network path to be able to run in a client-server mode in user space without the need to have super user privileges. *Pathrate* makes use of both the packet pairs and the packet trains in different phases of the measurements as described in subsequent sections. Beforehand let us understand the concept of bandwidth distribution and the need to use statistical methods to estimate capacity while using PPTD.

The bandwidth measurement of the network path with multiple hops (links) using packet pair in the absence of cross traffic at each link is nothing but the capacity of the path [29]. However, in the real world, cross traffic prevails at different links at different points of time. The dispersion Δ of two packets at a link; each of size L as seen by receiver, varies with time and so does the corresponding bandwidth b (capacity) measured. Thus the measured bandwidth b is continuous random variable with respect to time. The continuous random nature of b is established in equations in Section 3.2.1 and 3.3. b follows the probability density function B referred to as the *packet pair bandwidth distribution* which is multimodal in nature [29]. Thus, we use the term "*estimating capacity*" interchangeably with the term *"measuring capacity"* while using the packet pair for capacity estimation. Each mode also called the local mode [29] discovered during measurements is the probable capacity of that network path. The multimodal nature of the bandwidth distribution B can be represented in the form of a histogram for numerous packet pair measurements. Some local modes discovered underestimate the capacity while a few

overestimate it based on the cross traffic interference at the time of the measurement. One of them is the capacity mode representing the path capacity. Statistical techniques are used to discover local modes and then to estimate the capacity from among the local modes to eliminate the effect of various kinds of cross traffic. Please refer to the reference paper [29] for more details on the analysis of the effect of cross traffic on estimating the capacity.

Pathrate working is divided into three execution phases, namely, the initial phase, phase 1 and phase 2. The initial measurement phase is to detect the maximum train-length N*max* that the path can carry without packet loss. This is required to avoid flooding the network with long packet trains. The maximum value N*max* can have is 50 [29]. Phase 2 uses this to send packet trains of length N*max*. *Pathrate* then generates 60 packet trains of increasing length from N=2 to N=min(10,N*max*). These initial measurements also check on whether the network path is heavily loaded. For a lightly loaded path (low variations in measurements), *Pathrate* returns with a quick capacity estimate, which is the average of the preliminary measurements after removing 10% of the smallest and largest values [29]. These values are removed as they are treated as outliers that don't fall within other values measured and to accurately measure the average capacity. The second part of initial phase is to choose bandwidth resolution or the range also known as the bin width ω [29]. Consider this as a histogram bin width where the final capacity will be the range of this width. ω represents width or range of a bar in a histogram to represent the number of measurements lying within that range. ω is set to about 10% of the interquartile range of preliminary

packet train measurements while measuring Nmax. ω is an important parameter in discovering local modes in the bandwidth distribution *B* in phase 1 [29] as explained later. The final capacity estimate is the range of this width.

Phase 1 generates numerous packet pairs (1000) to discover local modes in bandwidth distribution *B*. One of the modes in these local modes is capacity mode. Packets in these measurements are of variable size to make non-capacity modes weaker and wider in distribution. Making modes wider and weaker helps to identify and eliminate non-capacity modes in Phase 2 as explained further. The packet size range between L*min* and L*max,* where L*min* is 600 bytes and L*max* is Maximum Segment Size (MSS) discovered in the initial phase. Each local mode M*i* discovered in this phase is characterized by central bin R*i*, number of measurements S*i* in R*i* from among range of measurements W*i*. The average bandwidth in central bin R*i* for a mode is denoted by H*i*. Figure 5.2 shows characteristics of such local modes in distribution *B*. Each such mode can be represented by an equivalent histogram as shown in Figure 5.3 (first histogram). For detailed information about the heuristics and statistical methods used in choosing ω in the initial phase and discovering the local modes in phase 1, please refer to the Appendix of the paper Packet Dispersion techniques and a capacity estimation methodology [29]. After discovering these local modes, the next challenge is to discover the capacity mode from among these local modes as the final estimate of that path capacity.

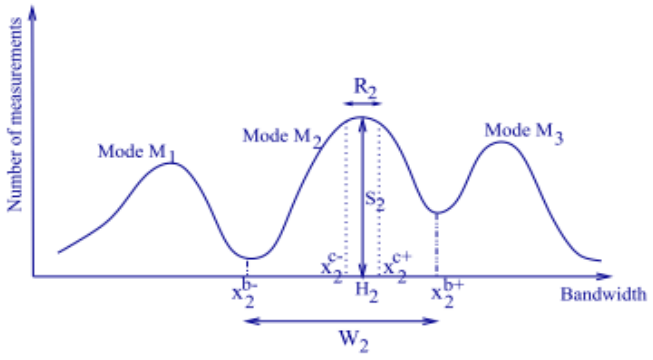


Figure 5.2: Characteristics of Local Mode [29]

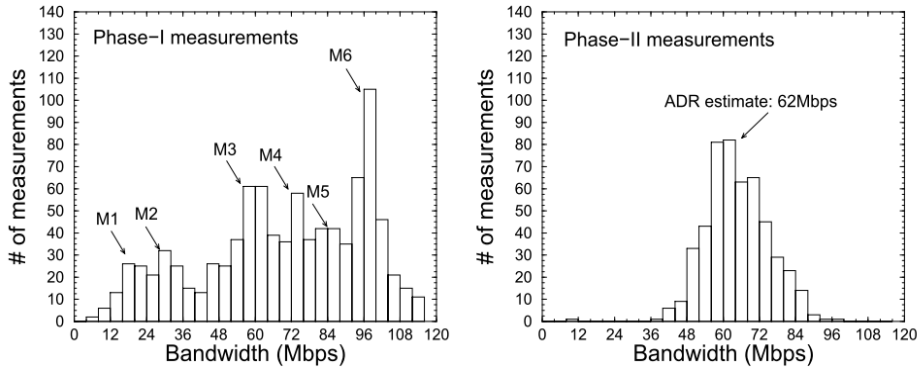


Figure 5.3: Histograms of Phase 1 and 2 Measurements [29]

Phase 2, the final phase does the job of identifying the capacity mode. This phase uses long packet trains to measure what is called the Average Dispersion Rate (ADR). ADR (R) is the mean packet train dispersion of all the packets in the packet train. Similar to packet pair distribution *B*, the packet train distribution is denoted by *B(N)*. Phase 2 uses 500 packet trains with N*max* and maximum sized packet L*max,* which were detected during initial phase. The authors in reference [29], through various experimental measurements and analysis with different values of train length N, have shown that the measured bandwidth tends towards

a certain value as N increases. This certain value is the mean packet train dispersion, the ADR. As the train length increases, the variability in corresponding packet train distribution *B(N)* becomes very low as the value tends to ADR and becomes unimodal in nature. In other words, the measured ADR is independent of packet train length N when N is sufficiently large. Furthermore as the train length increases, the probability of cross traffic interference is high resulting in additional dispersion causing corresponding measured bandwidth *b* to underestimate the path capacity. In short, the measured bandwidth is less than actual path capacity for large packet trains. Hence corresponding measured ADR is less than path capacity. The same statistical methods as used in phase 1 are used to calculate the ADR of the distribution B*(N)* and represents one of the modes in *B(N)*. This mode is also known as the global mode. The next step towards capacity estimation is to eliminate all Phase 1 modes whose average bandwidth value H*i* in central bin R*i* is below the ADR value as those modes underestimate the path capacity. If there are still many Phase 1 modes whose H*i* is greater than ADR value, the next step is to choose a local mode that is relatively narrow and strongest mode among remaining local modes. This is figured out using a simple value known as Figure of Merit F*i* belonging to each mode M*i*. F*i* is the number of measurements S*i* in central bin R*i* of a mode M*i*. A higher value of S*i* (number of measurements in central bin width of a histogram) represents a higher value of the Figure of Merit F*i*. This means, large number of measurements are concentrated around central bin R*i* of a histogram belonging to a local mode, making that mode narrower. As mentioned earlier, Phase 1 uses

variable sized packet pairs, both small (L*min*) and large (L*max*) to wider and weaker the non-capacity local modes. A wider mode means that the measurements are not concentrated around its central bin R*i* and instead spread across its entire range of measurements W*i*. Thus the average bandwidth value H*i* in central bin R*i* is low and so is the corresponding Figure of Merit F*i* for a wider mode, making it a weaker and less probable mode to be considered as a capacity mode. Hence the path capacity is a local mode M*i* whose average bandwidth value H*i* in central bin R*i* is greater than the ADR value (global Mode) with highest Figure of Merit F*i*. As shown in Figure 5.3, the Mode M6 (98 Mbps) is strongest local mode having value greater than ADR mode (62 Mbps). Hence the final estimated path capacity of the network path is 98Mbps. Figure 5.4 shows a high level overview on *Pathrate* algorithm. The connector L and B represent Latency and Bandwidth measurement flow respectively. Since capacity estimation is based on statistical methods, it is very critical to choose a correct initial bin width ω used in discovering local modes in phase 1. Please refer to the reference paper [29] for more details on the previous work and literature in capacity estimation using different techniques. It also provides in depth analysis of the packet pair/train distribution and its accuracy based on a large number of measurement datasets as part of their experimentation. Chapter 6 provides outputs and analysis of the various measurements performed and the corresponding local modes discovered between the hosts on the University of Missouri Research Network (Rnet).

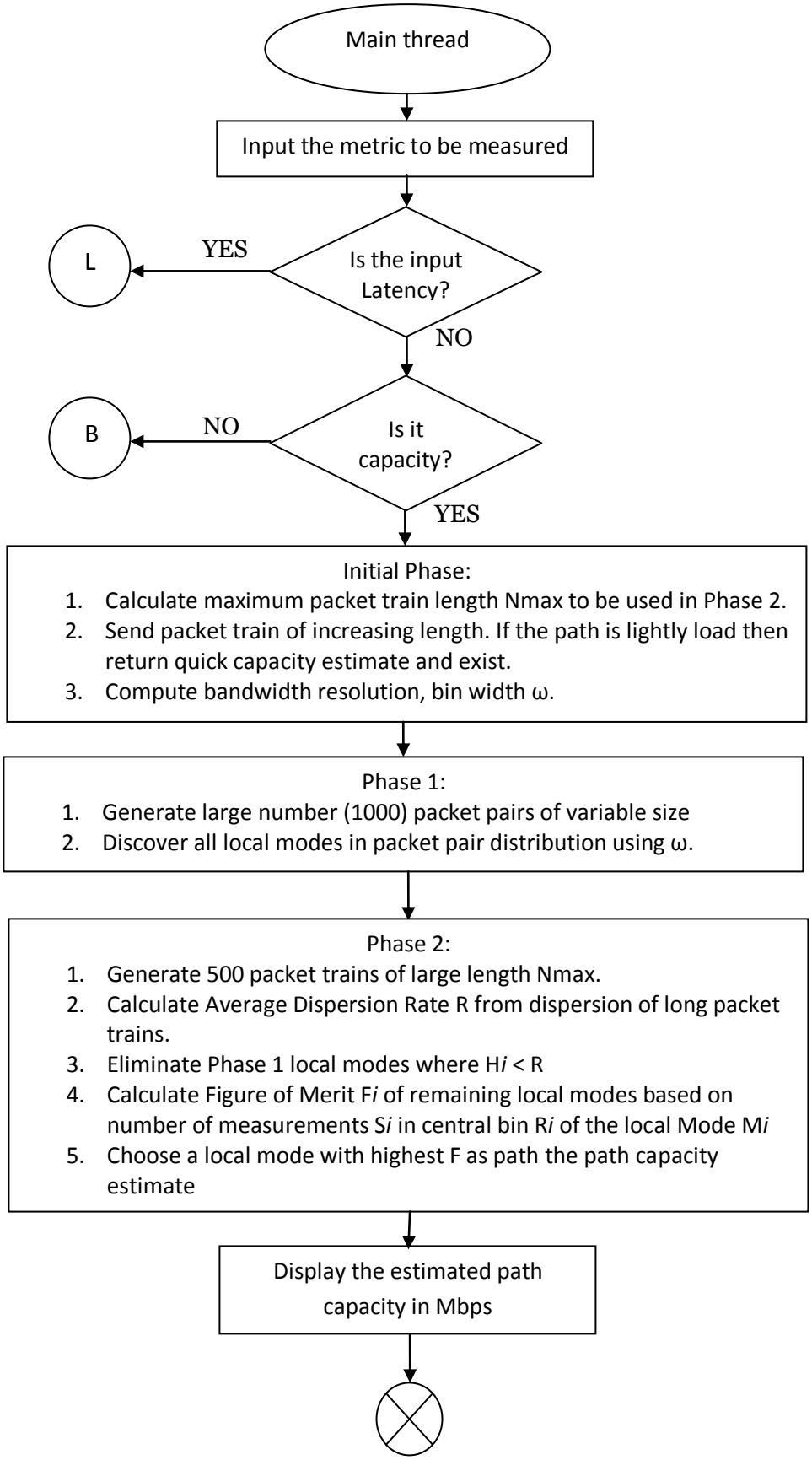


Figure 5.4: Capacity Estimate Flow Diagram [29]

5.2.2 Bulk Transfer Capacity Measurement

The previous section explained how PPTD technique is used to estimate the path capacity, which is a fairly constant Bandwidth metric until the underlying network infrastructure changes. This section explains one technique to measure Bulk Transfer Capacity for the TCP protocol that represents achievable throughput (TCP Throughput) for a single TCP connection on a network path between the two hosts. The measurement technique uses a simple formula of counting the number of bytes received within a specified amount of time as shown below. The size of a byte is represented by the constant CHAR_BIT whose value is dependent on the underlying OS, but typically, the value is 8 bits.

Current BTC = ((Number of Bytes Received/ Time) * CHAR_BIT) /1000000 Mbps

The measurement is performed for a single TCP connection over either IPv4 or IPv6 protocol based on user input. Each measurement period is called a measurement session. During each session, the server transmits a specified amount of data to the client. The client then calculates the current session's TCP throughput using the above formula based on the number of bytes received within a specified time. The unit of measurement is in Mbps and the precision of time captured is in nanoseconds of the real time clock. Each session also keeps track of the maximum BTCmax, minimum BTCmin and the average bandwidth

BTCavg achieved on all the measurement sessions including the current session. The formula to calculate the average throughput is as below

BTCavg = ((Average BTC *Total sessions) + Current BTC)/Total sessions + 1

Each packet contains the current session number, packet number and the time the packet was sent. This is in order for the client to keep track of the data received during each measurement session. The packet size is constant but the actual size may vary from OS to OS based on whether it is 32-bit or 64-bit architecture. Figure 5.5 shows the BTC measurement packet. During each measurement session, numerous BTC measurement packets of constant size are transmitted over established TCP connection. Thus, the amount of data sent during each measurement session is the number of packets * size of BTC measurement packet. The same amount of data is sent while measuring the BTC for both IP protocols for comparative analysis. After the client calculates the current throughput, BTCmax, BTCmin and BTCavg for each session, these values are transmitted to the server over a separate TCP connection in order to display it on the server side. This separate TCP connection is known as control connection and corresponding data sent as the control information. The server spawns a separate thread to create and listen for incoming control information using a separate TCP connection and then displays the BTC measurements on the console. A separate connection is maintained to send control data in order to keep the throughput measurement socket connection unaffected from any kind of

congestion or packet loss that may occur while sending the control data as a design consideration. After each session, the server waits for ten seconds before starting another measurement session. The server keeps sending data for throughput measurement at intervals of ten seconds until the client closes its socket connections or a signal *SIGINT (Ctrl + C)* is received. The delay of ten seconds between two consecutive measurement sessions means the variation in BTC over a network path is captured at intervals of ten seconds. The delay is maintained to take into account the packet loss and the corresponding retransmission of these lost packets by TCP if occurred. However, as a future enhancement, this value would be configurable based on user input. The measurement session does not end until the signal is received as one of the design considerations for the user to be able to measure BTC as long as desired. Since BTC is dependent on many factors changing real time as explained in Section 3.1.3, there is no defined threshold to compare this changing metric calculated and hence a good approach is to get as many measurements as possible until the users see the BTCavg value stabilized. Figure 5.6 provides a high level flow diagram for the steps followed in measuring BTC. For all the socket operations, the middleware application functions are used to minimize the code to handle both IPv4 and IPv6 protocols.

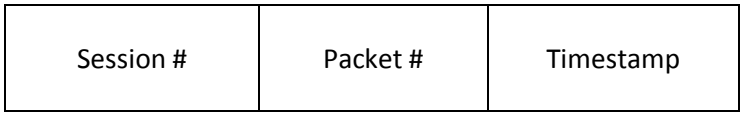


Figure 5.5: BTC Measurement Packet

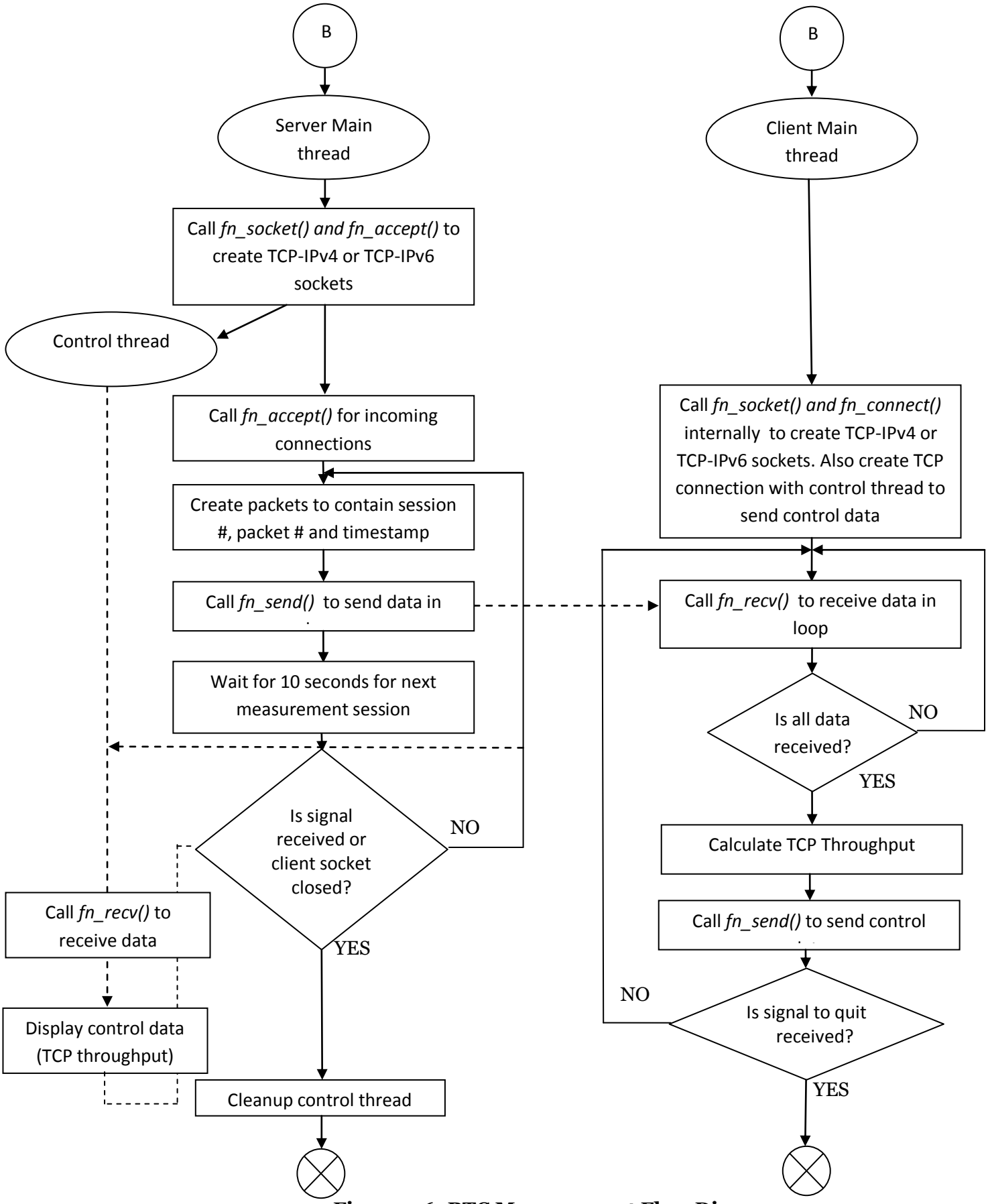


Figure 5.6: BTC Measurement Flow Diagram

5.2.3 Latency Measurement

Latency Measurement uses a simple measurement technique to calculate Round Trip Time (RTT) of a network path and then divide the calculated RTT by two to represent one-way latency. The server measures the RTT as the time difference between the time the packet was sent until the time it is received back from the client. However, the client first initiates a request by sending a control packet to the server in order to start the latency measurement. Once the server receives the request, it starts the latency measurement by sending the measurement packets. UDP packets are used over either IPv4 or IPv6 as per the user's input to measure latency for the corresponding IP protocol. The packet contains the session number, packet number and time it was sent. These three values are stored as part of the packet structure as shown in Figure 5.7. Each measurement session consists of transmitting one packet and receiving it back for latency measurement. The size of each packet sent is the size of this structure in addition to the TCP and IP header size. Currently, the amount of data sent per measurement session is set 56 bytes at the application layer. This includes the size of the packet structure followed by additional padding at the end. However, as a future enhancement, this value would be configurable based on user input.

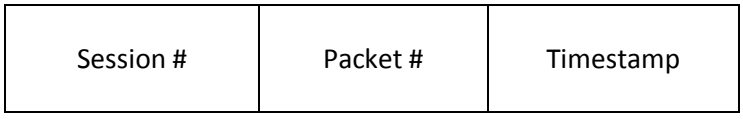


Figure 5.7: Latency Measurement Packet

The precision of the time captured is in nanoseconds of the real time clock. The corresponding time difference, while calculating the RTT, is also calculated in nanoseconds. Latency measured in each measurement session is then displayed in milliseconds (ms) on the server side with five digits scale (number of digits to the right of the decimal point). The conversion from nanoseconds to milliseconds to display the latency value is to keep in line with the other standard latency measurement tools like Ping, Speednet. However, the five digits scale allows to display the variation in the measured latency in nanoseconds (last two digits). The precision of nanoseconds is used taking into consideration the use of the measurement tool over high speed networks. Each measurement session also keeps track of the minimum, maximum and average latency over the previous measurement sessions. The formula to calculate average latency is defined below and similar to the formula used to measure BTCavg.

$$\text{Average Latency} = ((\text{Average Latency} * \text{Total packets received}) + \text{Current Latency}) / \text{Total packets received} + 1$$

The thing to note here is that the time captured represents application level time as opposed to the OS kernel timestamps. Thus, the corresponding measured latency is an end-to-end latency as seen by an application, which is a combination of the propagation, transmit, network queuing and processing time [4]. The latency measurement is performed in a multi-threaded environment on both the server and client side. The server uses two autonomous threads, one to transmit

the packet and the other to receive packets. These threads are called the transmit and receive threads respectively. These threads are spawned once the socket connection is established and are available to be used by both of the threads. Here a single socket connection is maintained for measurement whose port numbers are known to both the client and server in advance. The middleware application functions are used to create and use the sockets to send and receive latency measurement packets. The transmit thread is responsible to build packets for each session and send them to the client. After each session, the transmit thread waits for two seconds before starting the next session. The delay value of two seconds is again to be in line with the other standard latency measurement tools like Ping. The receive thread is responsible for receiving packets echoed back by a client and to calculate the time difference between the time the packet was received and the time it was sent. This time difference is the RTT. Once the RTT is calculated, it computes the corresponding one-way latency and also correspondingly calculates the maximum, minimum and average latency among all previous sessions including the current session. Once all the values are calculated, the results are displayed on the console. The thread on the client side, called the echo thread, simply echoes back the packet received from the server. The server keeps measuring the latency at the specified interval until the client closes its socket connection or a signal SIGINT (Ctrl + C) is received. Figure 5.8 provides a high level flow diagram for the client-server communication for latency measurement.

The tool provides inter-thread communication between the control thread that sends and receives control information as explained in the previous sections and the threads involved in performance measurement. This communication involves resource cleanup, signal masking and unmasking, and use of mutual Exclusion (MUTEX) for global variables. The next chapter provides measurement outputs and analysis for each of the three performance metrics between two hosts on the University of Missouri Research Network (Rnet) using this measurement tool to demonstrate its utility.

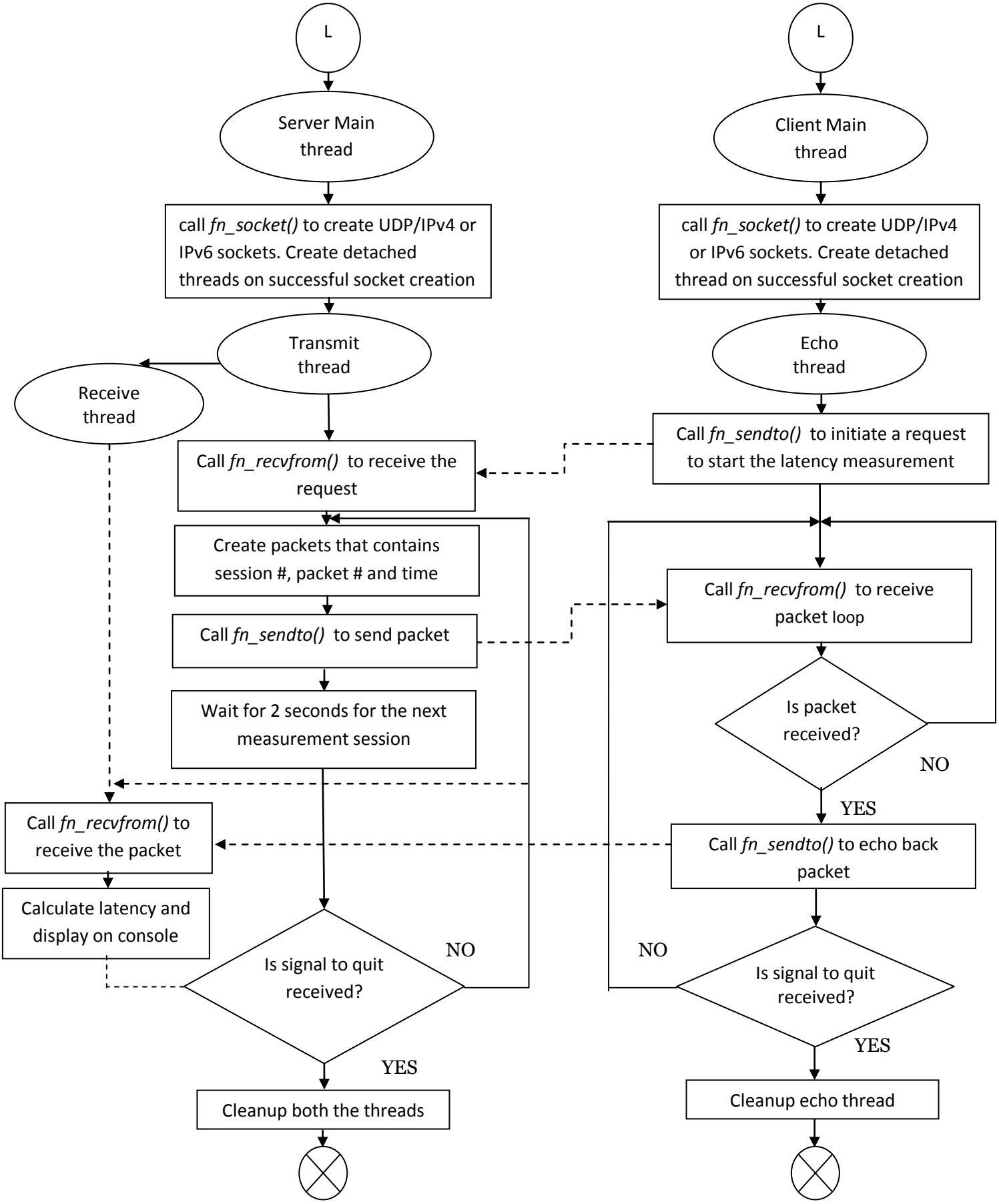


Figure 5.8: Latency Measurement Flow Diagram

Chapter 6 – Measurement Tool Output and Results

This chapter discusses the measurements performed for IPv4 and IPv6 protocols between two hosts on the University of Missouri Research Network (Rnet) using the measurement tool *netaware*. It demonstrates the tool's utility in measuring various performance metrics for both IPv4 and IPv6 protocols combined in a single tool. As discussed earlier, the measurement tool utilizes the middleware application functions and constants to operate on sockets. The middleware application functions are used to send and receive data to measure performance and also to exchange control information for client server communication. The tool is used to measure three performance parameters, namely, capacity, Bulk Transfer Capacity (BTC) and the latency for both IP protocols between the two hosts. The tool uses the measurement technique and the formulas discussed in Chapter 5 to measure each of the performance parameters. Each measurement section is devoted to each performance parameter to include sample screen shots of the tool output and data gathered over the measurement period. The data gathered and their corresponding graphs are used to make observations about the use of both IP protocols between the two hosts. The hosts chosen to measure performance are ***peregrine.rnet.missouri.edu*** (peregrine) and ***web.rnet.missouri.edu*** (web). The characteristics of the chosen hosts are listed below.

- Linux based servers (web-GNU/Linux 2.6.18 and peregrine-GNU/Linux 2.6.32).
- The hosts provide support for both IPv4 and IPv6 protocols and correspondingly have support as a dual stack operating system.
- The hosts support POSIX standards and threads.
- Both servers are on different parts of Rnet connected by a router and each has 1G network connection on 10G Rnet.

Figure 6.1 shows the network connections between ***web*** and ***peregrine*** on Rnet. As seen in this figure, ***web*** and ***peregrine*** are on different Virtual LANs (VLAN) on Rnet and have to pass through a router to determine the routing. For presentation purposes, two routers are shown to distinguish between IPv4 and IPv6 addressing at each interface but physically, there is only one router between the two servers for both IPv4 and IPv6 protocols. The IP addresses denote the addresses at each network interface for each IP protocols. For example, the IPv4 address for ***peregrine*** is 128.206.118.142 while IPv4 address for ***web*** is 128.206.116.112. In IPv6 address space, each server has two addresses. One assigned address and one computed address by the server when plugged into the network. For example, ***peregrine*** has two IPv6 addresses, 2610:e0:a010:301::9 and 2610:e0:a010:301:225:64ff:fec9:ad7a while ***web*** also has two IPv6 addresses, 2610:e0:a010:301::8 and 2610:e0:a010:302:219:b9ff:feba:7d3f. Thus ***web*** can be reached from ***peregrine*** and vice versa through either IPv6 addresses.

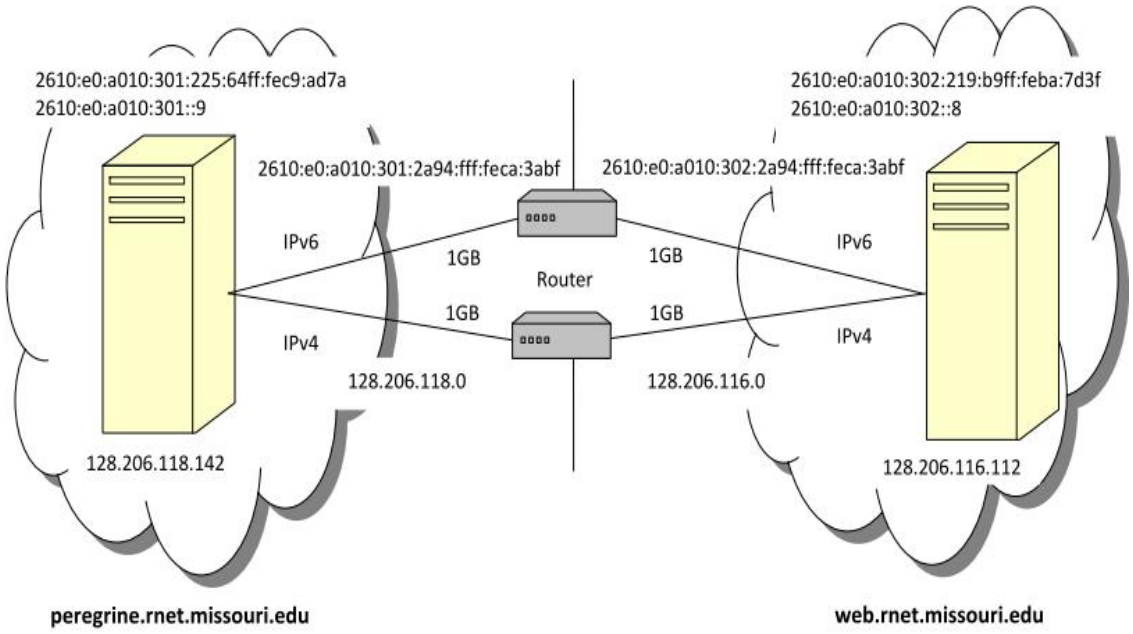


Figure 6.1: Web and Peregrine on Rnet

The measurement tool is executed from the command line. The corresponding client and server programs are named ***netaware_rcv*** and ***netaware_snd*** respectively. These command line programs have various options that can be supplied when starting the respective programs. The important ones are the options to choose among the performance parameters to measure and the IP protocol for which performance parameters needs to be measured. The option M specifies which performance parameter to measure and the option P specifies the IP protocol. Apart from these two options, there are a few more options that should be specified when the option –M has value C to measure the capacity. Tables 6.1 and 6.2 provides possible values for all the options and their corresponding meanings for n*etaware_snd* (server) and

n*etaware_rcv* (client) respectively. The command line arguments for each of the two programs are as below.

netaware_snd [-i] [-H|-h] [-q|-v] [-o <filename>] [-P <4|6>] [-M <L|C|B>]

Options	Meaning	Value
-i	It is a switch to run the sender in iterative mode	
-q	Quite mode	
-v	Verbose mode	
-o <file>	Print log in user specified file	Log file name
-P	IP protocol to be used for performance measurement	4/6
-M	Performance metric/parameter to measure	L/B/C
-H\-h	Print help and exit	

Table 6.1: *netaware_snd* Options

netaware_rcv -s <sender> [-H|-h] [-Q] [-q|-v] [-o|-O <filename>] [-P <4|6>] [-M <L|C|B>]

Options	Meaning	Value
-s	Hostname	Host name
-q	Quiet mode	(Use either –q or –v)
-v	Verbose mode	(Use either –q or –v)
-Q	Quick termination mode	
-o	Save log output in user specified file	Log file name
-O	Append log data in user specified file [default is pathrate.output]	Log file name
-P	IP protocol to be used for performance measurement	4/6
-M	Performance metric/parameter to measure	L/B/C
-H\-h	Print help and exit	

Table 6.2: *netaware_rcv* Options

A sample command line argument for the program *netaware_snd* (server) to measure the Latency for Ipv4 protocol is shown below. For the case where no client name or correct IP address is provided as an input parameter, the server retrieves the client details when the client establishes a connection with the server.

```
netaware_snd –P 4 –M L peregrine.rnet.missouri.edu
```

The next three sections focus on the measurement output for each of the performance metrics/parameters for both IP protocols. The sender and the receiver buffer values in the applications level are the same for both IP protocols.

6.1 Latency Measurement Output

Latency is measured between two hosts (***web*** and ***peregrine***) for both the IPv4 and IPv6 protocols. The sender/server (*netaware_snd)* is run on the ***Web.rnet.missouri.edu*** system while the client/receiver (*netaware_rcv)* is run on the ***Peregrine.rnet.missouri.edu*** system to measure one-way latency from ***Web*** to ***Peregrine***. The value of L is specified for the option –M to measure latency and the value of 4 and 6 for the option –P to measure latency over IPv4 and IPv6 protocols respectively. Figures 6.2 and 6.3 shows screenshots of the sample output latency measurements. Figure 6.2 shows the *netaware_snd* output on ***web*** that displays the Minimum, Maximum and the Average latency in Milliseconds (ms). Figure 6.3 shows the *netaware_rcv* output on ***peregrine***

that displays the number of packets received during each measurement session. Both the outputs also display the IP address and the port number of the peer machines. In this example, the client (***peregrine***) with IPv6 address 2610:e0:a010:301:225:64ff:fec9:ad7a at port number 45341 is connected to the server (***web***) with IPv6 address 2610:e0:a010:302:219:b9ff:feba:7d3f at port number 15002. The port numbers to be used are known to both the client and the server as part of the tool built. Currently the port numbers are not configurable, but as part of future enhancement, the tool will be able to read the port numbers from a user configurable file at run time. The measurement tool continues to measure latency until the tool is interrupted by a signal *SIGINT (Ctrl + C)*. The client and server both have signal handlers to receive and handle the signal *SIGINT*. On receiving *SIGINT*, the signal handler sets a global variable. All the threads keep a check on this global variable and once set, it calls a resource cleanup function to close socket connections to exit. The measurement tool program running on the host across the network returns an error value of zero on the function call *fn_recvfrom()* when its peer host closes the socket connections. Correspondingly, the hosts on the other side then also call its resource cleanup functions to close socket connections and exit all threads.

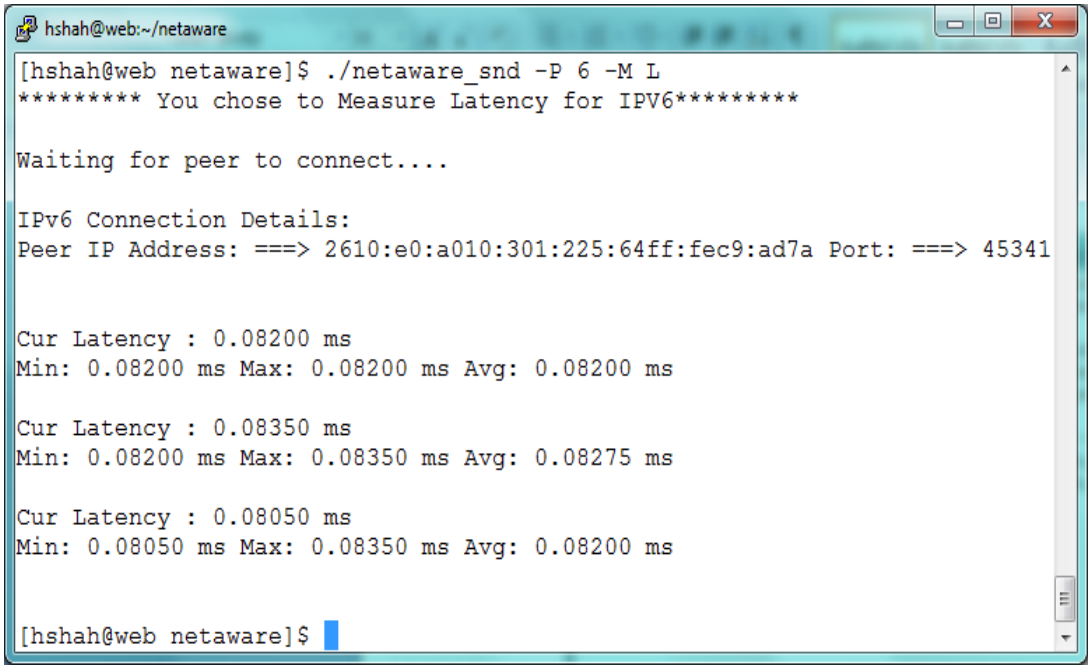


Figure 6.2: *netaware_snd* Output on Web for IPv6

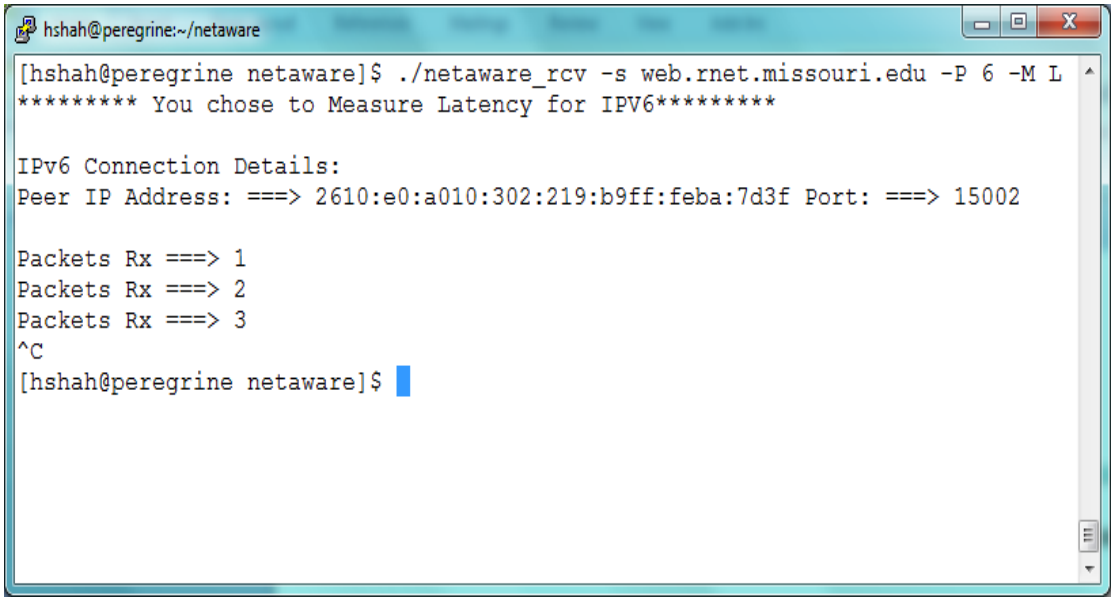


Figure 6.3: *netaware_rcv* Output on Peregrine for IPv6

The measurement tool was run between ***web*** and ***peregrine*** to gather latency measurements at different points of time during a day to check the variation in the measured latency. Figure 6.4 shows the graph of latency measurements over both IPv4 and IPv6 protocols from 9 am to 9 pm. A latency measurement packet of 56 bytes is sent and received back by the sender during each measurement session. The time elapsed for each measurement session is the Round Trip Time (RTT) from ***web*** to ***peregrine***. As discussed in Section 5.2.3, the latency measurement packet contains the session number, packet number and the time the packet was sent. A delay of 2 seconds is maintained between each measurement session. A total of 7,384 measurement sessions and corresponding measurement packets were sent from the sender to the receiver for a duration 12 hours. This amounts to a total of 428,272 bytes (around 418 KiloBytes) of measurement data sent from ***web*** to ***peregrine*** the over IPv4 protocol. For IPv6, a total of 7,356 measurement packets (around 412 KiloBytes) were sent from ***web*** to ***peregrine***. The minimum and maximum latency values for IPv6 are 0.0795 milliseconds (ms) and 0.0955 milliseconds (ms) respectively while those for IPv4 are 0.0695 milliseconds (ms) and 0.0858 milliseconds (ms) respectively. These values are end-to-end latency values as seen by an application. The latency for both of the IP protocols remains fairly constant during the measurement period which is an indication that the network path is fairly congestion free during latency measurement [32]. No packet loss was seen during measurement over both IP protocols. Table 6.3 gives the minimum, maximum, median, standard deviation and variance of the measurements for

both IP protocols and Figure 6.5 shows the corresponding graph. Each data point in the graph shows the standard deviation as vertical line around it.

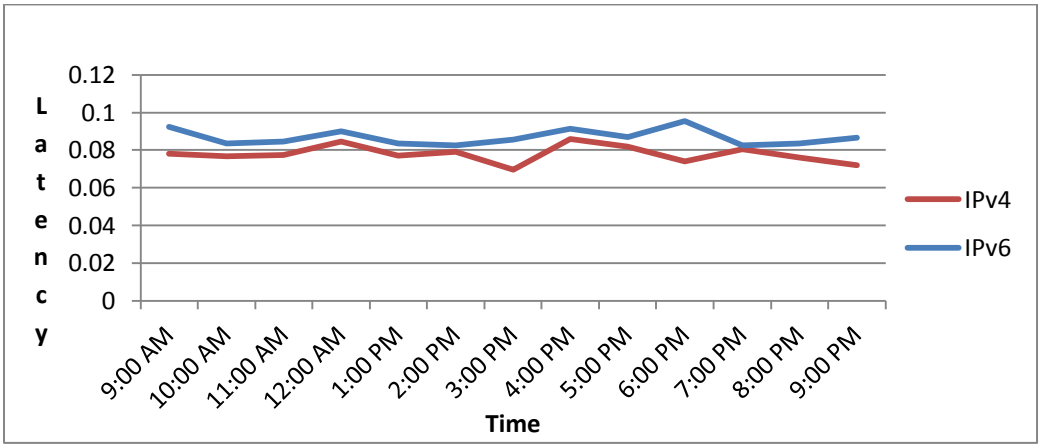


Figure 6.4: IPv4 vs IPv6 – Time vs Latency (ms)

	Minimum	Maximum	Average	Median	Standard Deviation	Variance
IPv4	0.0695	0.0858	0.0779	0.0774	0.0046	0.00002
IPv6	0.0825	0.0955	0.0868	0.0855	0.0042	0.00002

Table 6.3: IPv4 vs IPv6 Latency Measurements (ms)

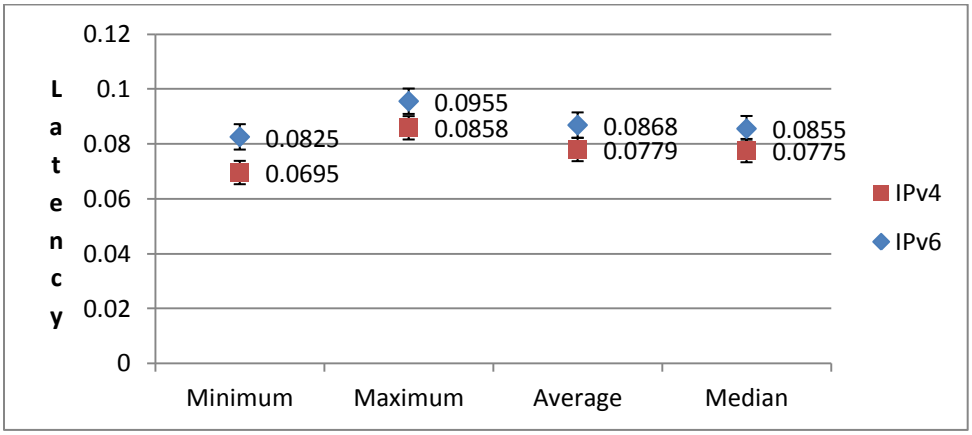


Figure 6.5: IPv4 vs IPv6 Latency Measurements (ms)

Furthermore, the tool was run for 5 days for a duration of 12 hours each day to gather more data on one-way latency from ***web*** to ***peregrine***. The size of

the latency measurement packet was increased from 56 bytes to 1500 bytes over this 5 days period. No packet loss was seen during entire measurement duration. Table 6.4 shows the minimum, maximum and the average latency calculated over the period of a duration of 12 hours each day. Figure 6.4 shows the graph of the maximum, minimum and the average latency calculated over period of 5 days. The maximum values reached up to 0.13 and 0.2 milliseconds (ms) for IPv4 and IPv6 protocols respectively. The minimum and maximum values for both the protocols are fairly comparable. Additionally it is observed that the measured latency increases as the packet size increases for both IP protocols. Comparing all the Figures and Tables values in this section, it is observed that latency over IPv6 protocol is slightly higher than the latency over IPv4 protocol. The difference between maximum, minimum and average values for both IP protocols decreases as the packet size increases. The increased latency over the IPv6 protocol can be attributed to various factors starting from CPU utilization, link traffic, delay at the router processing IPv6 packets and many other factors as mentioned in reference [2]. Identifying each these factors with help of appropriate tools and access to the underlying network can be useful as a part of further study.

	Minimum (ms)		Maximum (ms)		Average (ms)		Packet Size (Bytes)
	IPv4	IPv6	IPv4	IPv6	IPv4	IPv6	
Day1	0.0495	0.066	0.0739	0.0814	0.0803	0.094	56
Day2	0.052	0.0715	0.081	0.198	0.06837	0.08733	256
Day3	0.065	0.0775	0.0975	0.102	0.07684	0.08709	512
Day4	0.081	0.088	0.0945	0.1032	0.0856	0.0952	1024
Day5	0.1	0.9555	0.1295	0.142	0.1089	0.1164	1500
	0.0495	0.066	0.1295	0.198	0.084	0.096	

Table 6.4: IPv4 vs IPv6 Latency Measurements – 5 Days (ms)

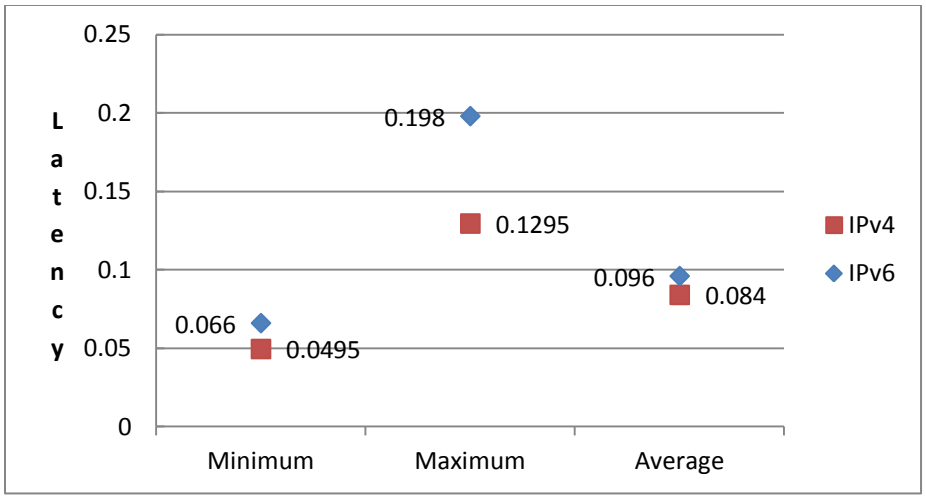


Figure 6.6: IPv4 vs IPv6 Latency Measurements – 5 Days (ms)

6.3 Bulk Transfer Capacity (BTC) Measurement Output

Bulk Transfer Capacity (BTC) is measured between ***web*** and ***peregrine*** by specifying the value B for the option –M and using the option –P with a value of 4 or 6 to choose between IPv4 and IPv6 protocols respectively. The tool is run to measure BTC from ***web*** to ***peregrine***, ***web*** being the server/sender and ***peregrine*** being the client/receiver. A sample output from the BTC measurement from ***web*** (*netaware_snd*) to ***peregrine*** (*netaware_rcv*) for the IPv4 protocol is shown in Figures 6.7 and 6.8 respectively. The sender and client both display the Minimum, Maximum and Average BTC measured. Both of the outputs display the IP address and the port number of the peer machine. Additionally, the client also shows the amount of data (in bytes) received during each measurement session. In this specific example, the client (***peregrine***) with IPv4 address 128.206.118.142 at port number 43765 is connected to the server

(***web***) with IPv4 address 128.206.116.112 at port number 10000. The port numbers to be used are known to both the client and the server as part of the BTC program within the measurement tool. Currently the port numbers are not configurable, but as part of future enhancement, the tool can read the port numbers from a user configurable file at run time. The measurement tool continues to measure the BTC until it is interrupted by a signal *SIGINT (Ctrl + C)*.

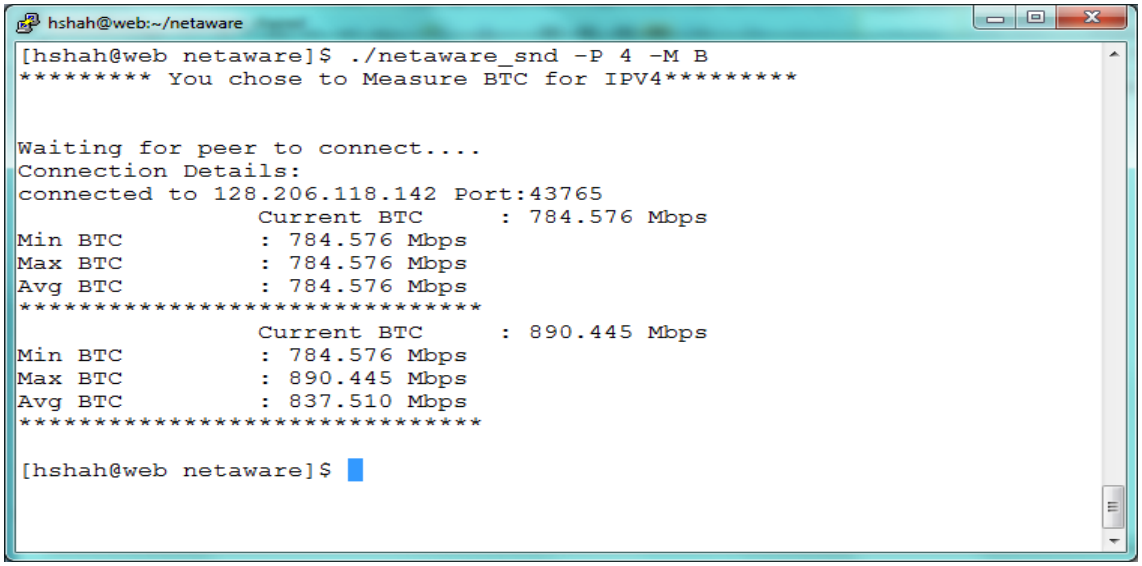


Figure 6.7: *netaware_snd* BTC Output for IPv4

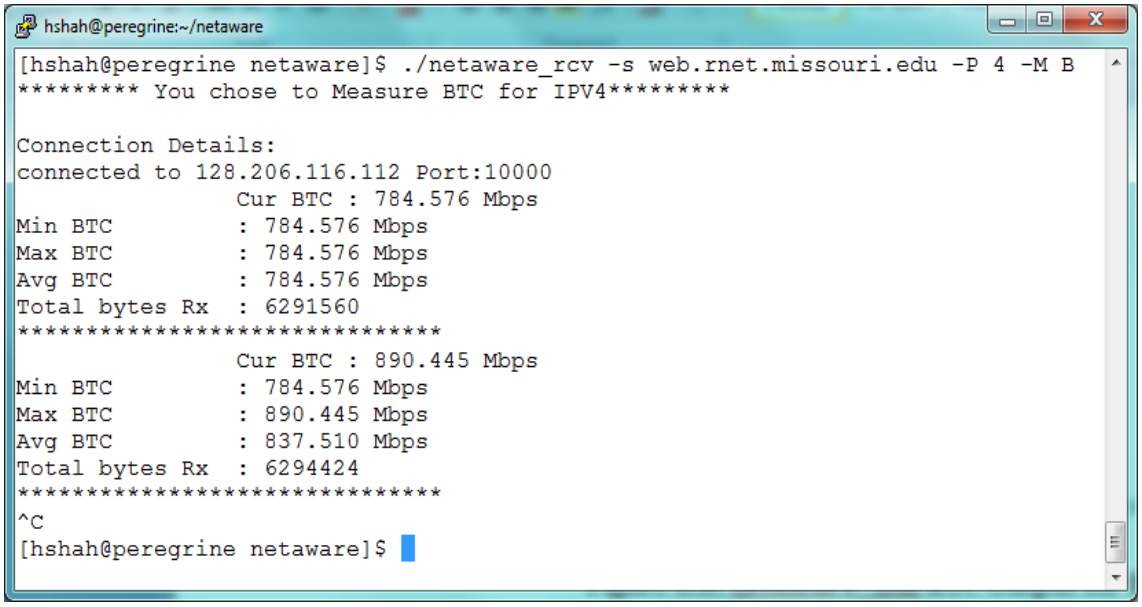


Figure 6.8: *netaware_rcv* BTC Output for IPv4

The tool was run for 12 hours to measure the BTC at different points of time during the day to observe the variation in measured BTC. Figure 6.9 shows the BTC measurement graph over both IP protocols from 9am to 9pm. Table 6.5 and the corresponding graph in Figure 6.10 provides the statistics of the measurements for both IP protocols. During each measurement session, 6Megabytes (MB) of measurement data was sent from the sender to the receiver for both IP protocols. As explained in Section 5.2.2, the measurement data consists of numerous BTC measurement packets sent during each measurement session. The packet consists of session number, packet number and the time the packet was sent. A delay of ten seconds is maintained between each measurement session. A total of 3660 and 3645 measurement sessions were performed for IPv4 and IPv6 protocols respectively. Correspondingly, the total amount of data sent from the server to the client was 21.5 Gigabytes (GB) and 21.3 Gigabytes (GB)

over IPv4 and IPv6 protocols respectively. The maximum, minimum and the average BTC values measured denote the values calculated over all the measurement sessions including the current session using the formula as explained in Section 5.2.2.

As shown in Figure 6.9, the BTC values for both IP protocols vary at different points of time. As seen from Table 6.5 and Figure 6.10, it is observed that the variation in BTC for both IP protocols is quite close. The minimum, maximum and the average values for both the protocols are quite comparable. These measured values represent the real time bandwidth available for a TCP connection. Moreover it is observed that the variation in BTC value is highest during the day time having the lowest values for both IP protocols. The dip in BTC values as seen by an application from noon to 3 pm can be attributed to various factors combined together as listed in Section 3.1.3. Capturing all these factors and corresponding analysis in the variations of the measured BTC require extra tools and privileges on the underlying network and were not considered in this thesis.

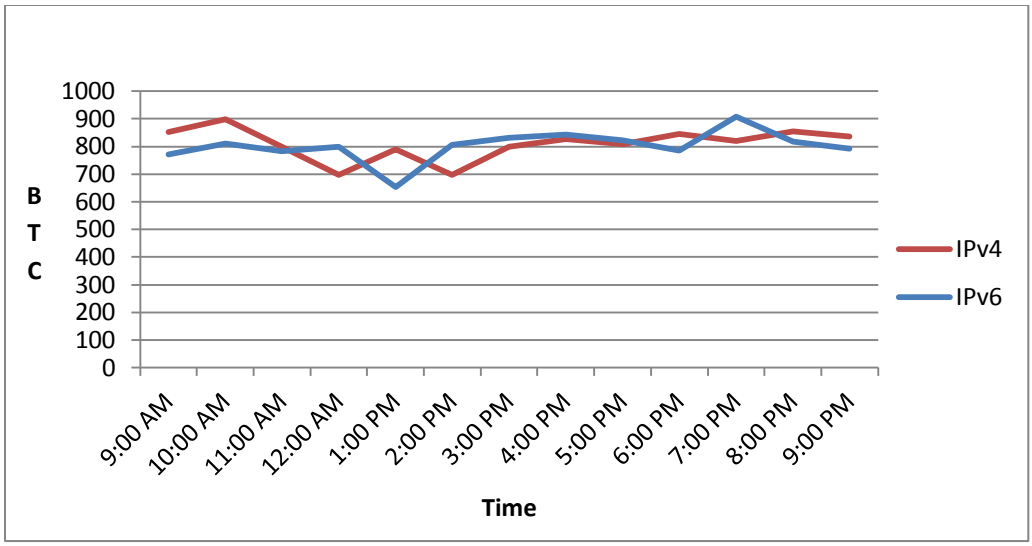


Figure 6.9: IPv4 vs IPv6 - Time vs BTC (Mbps)

	Minimum	Maximum	Average	Median	Stdev	Variance
IPv4	696.437	898.549	809.538	820.353	57.65921	3324.584
IPv6	652.824	907.782	801.7501	806.416	56.64739	3208.926

Table 6.5: IPv4 vs IPv6 BTC measurements (Mbps)

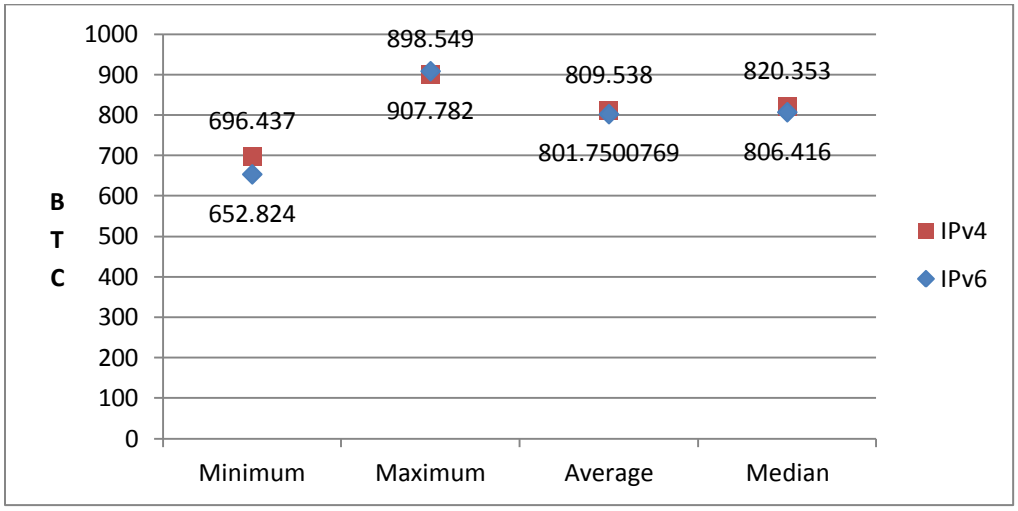


Figure 6.10: IPv4 vs IPv6 BTC Measurements (Mbps)

The tool was run for 5 days for a duration of around 12 hours each day to gather more data for BTC measurements. The amount of data sent per session was increased from day 1 to day 5. On day 1, the server sent 6 Megabytes (MB) of measurement data per session to the client while the amount was increased to 5 Gigabytes (GB) per session on day 5. The amount of data sent per session was increased in order to gather the measurement results for small data transfers as well large data transfers over a period of time over both IP protocols. Table 6.6 shows the measurement results obtained per day for both IP protocol. Figure 6.11 shows the minimum, maximum and the average BTC calculated over measurements gathered over the period of 5 days for comparative analysis. As seen from Figure 6.11, all the values are fairly comparable for both IP protocols with average value of measured BTC in the range of 820 to 860 Megabytes (MB). Comparing all the Figures and Table values in this section, it can be concluded that the BTC offered between ***web*** to ***peregrine*** over both IP protocols are quite comparable. The BTC offered over a network path varies with traffic at a given point of time.

	Minimum (ms)		Maximum (ms)		Average (ms)		Size/session
	IPv4	IPv6	IPv4	IPv6	IPv4	IPv6	
Day1	779.748	771.093	887.267	811.146	828.117	791.12	6 MB
Day2	779.748	749.468	904.372	889.138	891.284	846.919	60 MB
Day3	679.13	676.07	885.241	858.839	832.761	824.343	500 MB
Day4	675.962	831.803	933.368	949.491	869.172	828.343	1 GB
Day5	756.042	711.803	903.766	908.979	836.516	828.343	5 GB
	675.962	676.07	933.368	949.491	851.57	823.813	

Table 6.6: IPv4 vs IPv6 BTC measurements - 5 Days (Mbps)

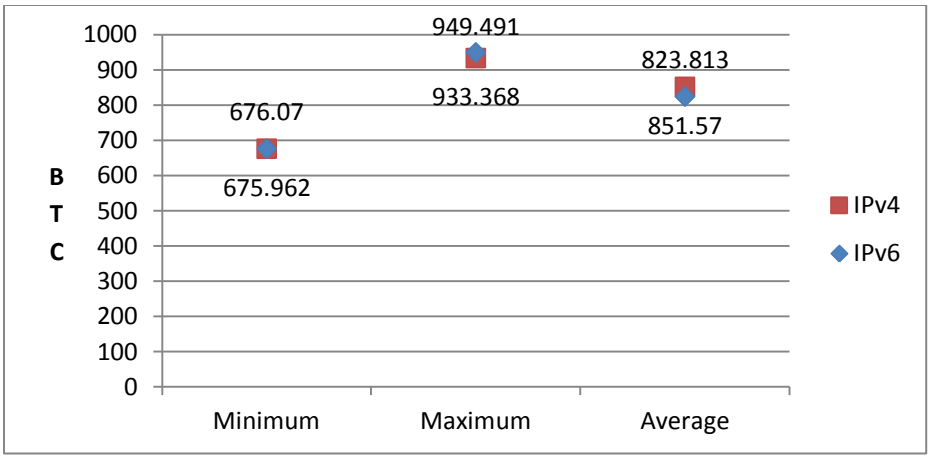


Figure 6.11: IPv4 vs IPv6 BTC Measurements - 5 Days (Mbps)

6.4 Capacity Measurement Output

As explained in Section 3.2.1, the capacity of a network path is the maximum capacity offered by a link over a series of link as opposed to BTC, which is the measure of real time available bandwidth. Capacity is a constant metric and doesn't change until the underlying network infrastructure changes [15] [34]. It is measured by specifying the value C for the option –M and the value of 4 or 6 for the option –P to measure capacity over IPv4 and IPv6 paths respectively. Apart from these two options, there a few more options as mentioned in Tables 6.1 and 6.2 that are used while estimating the capacity. For example, specifying the option –v makes the tool run in verbose mode; displaying more details of each measurement phase, while the value for the option -o specifies the name of a log file to record the output for each measurement phase for that run. These added options are provided in order to capture more details

on each of the three measurement phases of the *Pathrate* algorithm as briefly explained in Section 5.2.1. Figures 6.12 and 6.13 provides a sample output of the capacity measurement between ***web*** (*netaware_snd*) and ***peregrine*** (*netaware_rcv)* respectively. The outputs at both the ends display the final capacity measured between ***web*** and ***peregrine***. Additionally the output also displays details about the measurements carried during each phase. The details of the measurements captured during each phase of IPv6 capacity measurement is discussed below. A complete log file for IPv6 capacity measurement used in this study can be found in Appendix C.

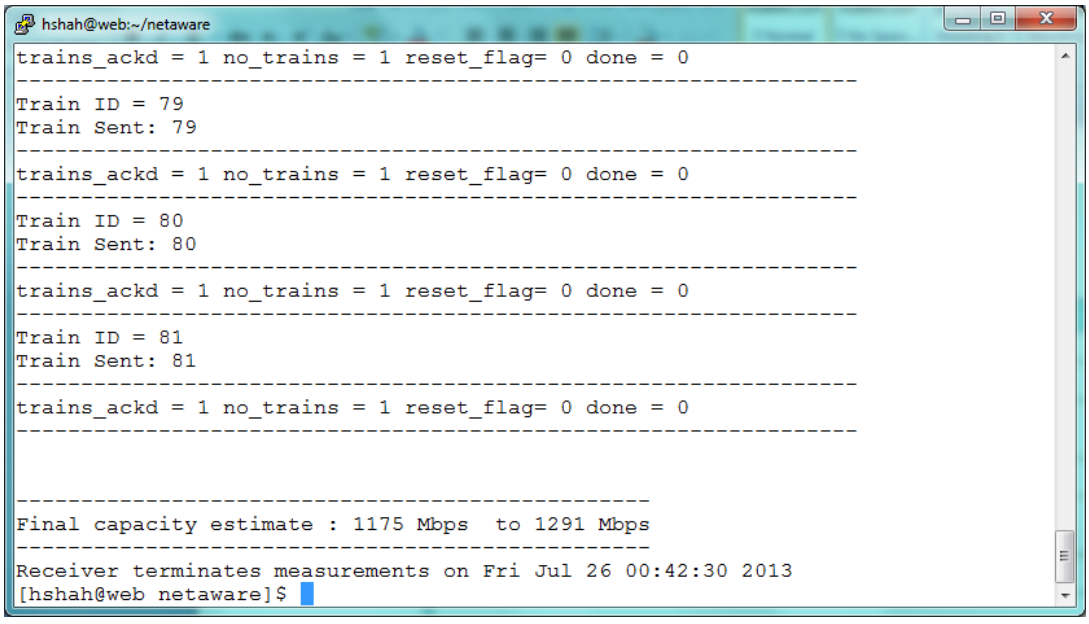


Figure 6.12: *netaware_snd* Capacity Output for IPv6 (Mbps)

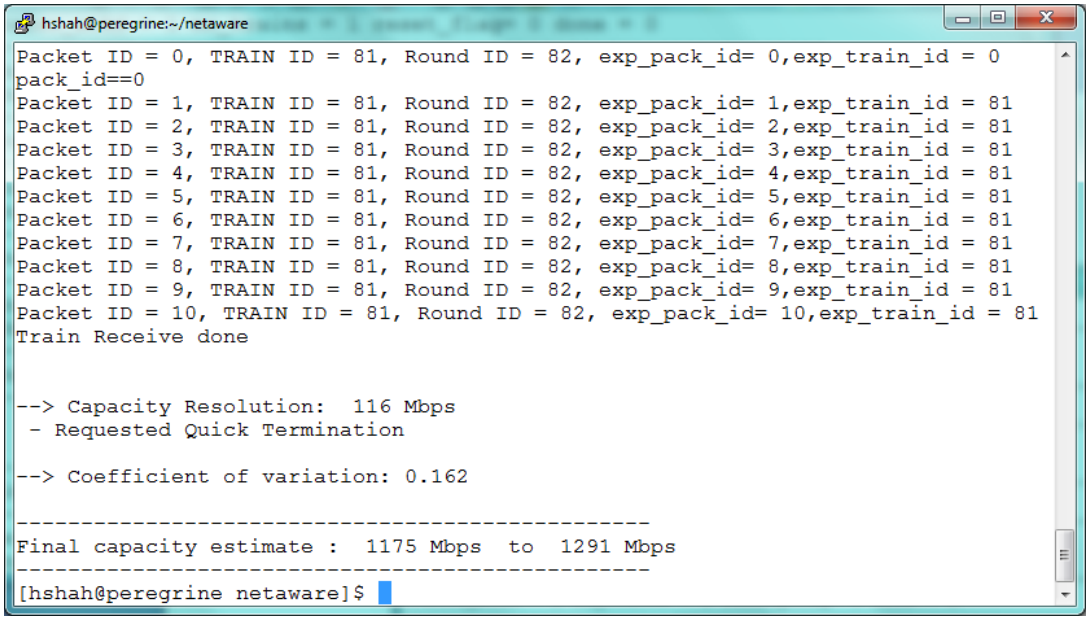


Figure 6.13: *netaware_rcv* Capacity Output for IPv6 (Mbps)

The first phase, of the *netaware_snd* program for capacity, also known as the initial measurement phase in *Pathrate* algorithm detected the maximum train-length N*max* to be 48 packets that the IPv6 protocol can carry without packet loss. N*max* is the maximum train length that Phase 2 uses when calculating the Average Dispersion Rate (ADR). This phase also calculated the capacity resolution, known as the bin width ω to be 56 Mbps. During this phase, packet trains with increasing train lengths from 2 to 10 were sent between ***web*** and ***peregrine*** to calculate ω. A total of 60 packets were sent during this phase. Phase 1 then detected three local capacity modes M1, M2 and M3 for IPv6 protocol. Figure 6.14 shows these local modes in the form of a histogram with corresponding measurements within each bin. For presentation purposes, this Figure does not plot measurements captured within bin widths for other non-local modes. A total of 1000 packet pairs with increasing train lengths from 2 to 8

were sent between ***web*** and ***peregrine*** to detect the local modes. Correspondingly, the packet size also increased from 600 bytes to upto 1488 bytes as the train length increased. As discussed in Section 5.2.1, one of the local modes discovered in Phase 1 represents the final capacity estimate. Phase 2 then calculated the ADR to be 954 Mbps. During this phase, 500 packet trains of length N*max* (48) were sent from ***web*** to ***peregrine***. The size of each measurement packet is 1488 bytes. Since ADR represents the mean packet train dispersion of all packets in the packet train, which is less than the actual path capacity, all the local modes whose value is greater than the ADR represents the probable IPv6 path capacity. In this study, modes M2 and M3 are the probable path capacity. Since there is more than one local mode as a probable path capacity, Phase 2 also calculated the figure of Merit M_i for each of these modes. The figure of Merit M_i represents the number of measurements in the central bin width of local modes. The figure of Merit calculated for mode 2 (M2) is 3.61 while that for mode 3 (M3) is 1725.43. Since mode 3 (1156 Mbps to 1212 Mbps) has the highest figure of Merit, it represents the final capacity for IPv6 protocol.

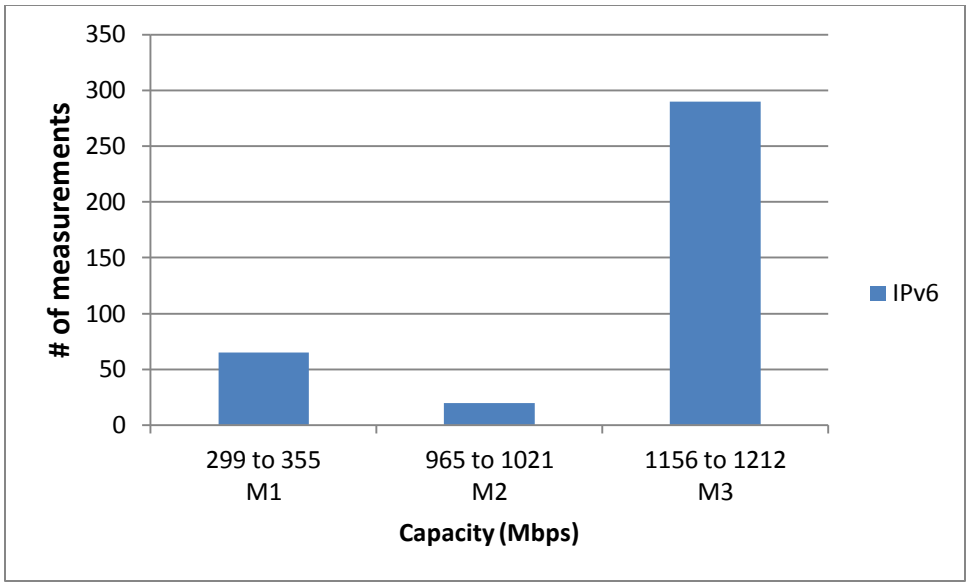


Figure 6.14: Phase 1 Local Capacity Modes -IPv6 (Mbps)

The tool was run to measure the path capacity over both IP protocols. Table 6.7 provides the capacity output measurement for both IP protocols. As observed, the capacity of both IP protocols is almost the same. Furthermore, the measured capacity at the application level is quite close to the underlying link capacity of 1GB between the two hosts as shown in Figure 6.1. Since statistical methods are used to estimate the initial bin width ω and also to discover the local capacity modes, the coefficient of variation is an important indicator of the accuracy. The higher the value, the higher the presence of cross traffic and lower the accuracy of the estimated capacity. A value greater than 1 represents heavy cross traffic and thus it is a good idea to rerun the tool to measure the capacity. In this specific example, the cross traffic encountered over IPv4 protocol is slightly higher than the IPv6 protocol [32]. Moreover, the average value of BTC measured

in the previous section is less than the estimated capacity for both IP protocols. This confirms that the available bandwidth (BTC) is less than capacity of underlying network path as explained in Section 3.2.1.

	Capacity (Mbps)	Resolution (Bin width ω)	Coefficient of variation	ADR (Mbps)	Nmax
IPv6	1156 to 1212	56 Mbps	0.112	954	48
IPv4	1113 to 1141	28 Mbps	0.061	965	48

Table 6.7: IPv4 vs IPv6 Capacity Estimation (Mbps)

Overall, based on the values measured for BTC and capacity over both IP protocols, it is observed that both IP protocols offer comparable performance in terms of bandwidth. However, IPv4 protocol provides a slightly better performance over IPv6 protocol in terms of latency. Based on the design and the nature of an application, an application can choose to use either IPv4 or IPv6 protocol between ***web*** and ***peregrine*** based on measured performance. The tool can be run multiple times over a longer time span (number of days) to gather more data and get better measurements of accuracy if needed. The tool implements state-of-the-art bandwidth and latency measurement techniques within a single tool for both IP protocols for a comparative analysis. The tool provides a way to easily measure a particular performance parameter for both IP protocols by simply specifying appropriate values for the options while running the tool. This measurement tool in combination with other tools can be helpful to not only measure the end-to-end performance, but to also identify existing

network problems over both IP protocols. The measurement tool helps get feedback in terms of the load or the capacity of a network path. The middleware application uses this feedback to help use a better network path in order to improve network utilization and in a larger picture improve the end-to-end network performance of an application.

Chapter 7 – Conclusion

Since the inception of the Internet, Internet Protocol version 4 (IPv4) has been one of its core working protocols and undoubtedly its backbone. With the tremendous success of the Internet and the exponential increase in the number of devices transmitting data over the Internet, IPv4 is running out of the address space (unique IP addresses). Its successor, IPv6 has been designed to not only resolve the issue of the exhaustion of the IPv4 address space, but also to provide improvements over its predecessor protocol. As IPv4 address space is nearing exhaustion, more and more networks have or are preparing for IPv6 deployment. However, the widespread use and success of the Network Address Translation (NAT), IPv6 tunnels over IPv4 only networks and private networks have served to delay the inevitable exhaustion of the IPv4 network addresses. Additionally, due to the difficulty in predicting the QoS of the newly deployed protocol, its connectivity, the time and the monetary cost involved, IPv6 is gradually replacing IPv4 as against its sudden conversion. Moreover, even though IPv4 addresses can be mapped to IPv6 addresses, IPv6 is not completely backward compatible with IPv4. Thus these two IP protocol stacks are expected to coexist (Dual IP stack) and supported by almost all the servers for a long period of time. A given host can be reached through either an IPv4 protocol or an IPv6 protocol.

This thesis has focused on leveraging the coexistence of both IP protocols (Dual stack). The combined use of the prototype middleware application

interface and the measurement tool *netaware* built as part of this thesis helps to leverage the coexistence of both IP protocols. Though wide scale deployment of IPv6 has occurred over the past few years, there have not been many applications switching to use the IPv6 protocol. The middleware application built as part of this thesis is to help in the creation of applications that use either IP protocol or both with minimal effort. The middleware application includes the middleware application functions and constants at the core of its design. The combined use these functions and constants provide an easy way to use either IP protocol over both TCP and UDP. A developer can focus on the core functionality of the application while letting the middleware application take care of creating and using sockets for Inter Process Communication (IPC) between hosts.

Chapter 4 provided more details on how this is achieved while Appendix B provided more details on each of the middleware application functions and a working example of the use of the middleware application. The utility of the middleware application was also demonstrated by using it in building the measurement tool *netaware* to operate on sockets as a proof of concept. The existence of both IP protocols provides an option to an application to choose an IP protocol in order to gain performance before it starts its execution or while in execution. The middleware application built also provides the capability for TCP to switch between IP protocols on the fly, without impacting the application built using the middleware application. As explained earlier, the aim is to better utilize the network by leveraging the existence of both IP protocols and hence provide an option to switch between them at some point of time based on the measured

performance. The switching capability can be beneficial for applications involved in large data transfers to switch the IP protocol for faster transfers.

The focus is on the TCP/IP protocol suite since it is the mostly widely used protocol among a wide range of applications and application layer protocols. Chapter 4 explained the complexity involved and correspondingly, the need for a special client-server communication and event handlers in order to gracefully transition from one IP protocol to the other for TCP. A sample file transfer application to transfer files of different sizes between hosts on Rnet was built to demonstrate the utility of this switching capability. Overall, the middleware application can be treated as an API to help migrate to use IPv6 protocol or to quickly build applications to use IPv4, IPv6 or both of the IP protocols. A shared library named *libnetaware.so* is built for the compiled version of the middleware application source code. An application wanting to use the middleware application needs to use middleware application functions and constants wherever the use of sockets is involved. An application needs to include a header file *commonhdr.h*, that acts as an interface to the middleware application and link its code to the shared library *libnetaware.so* at run time. Dynamically linking to the shared library and adding a header file provides code modularity for an application using the middleware application. This helps in accommodating any changes to the middleware application by simply using the latest header file and the shared library of the middleware application. The user may also need to modify the application code wherever the middleware application functions and

constants are used to incorporate any changes to the middleware application function calls and its return type.

Also as part of this thesis, a measurement tool named *netaware* is built to measure the end-to-end network performance for both IPv4 and IPv6 protocols to a single host combined into a single tool. The performance parameters measured are namely latency, Bulk Transfer Capacity (BTC) and capacity of a network path. Chapter 3 provided a detailed description of each of these performance parameters and also discussed the need to measure these parameters. Chapter 5 detailed the algorithms used to measure each of the performance parameters while Appendix C provided a complete log file for all the three phases of IPv6 capacity measurement. As explained earlier, this tool can be used to measure the performance for both IP protocols to a single host for comparative analysis. This is helpful to choose between a better IP protocols in order to leverage the coexistence of both IP protocols to enhance an application's performance. Furthermore, based on the measured performance, an application using the IPv4 protocol can change to switch to the IPv6 protocol and further optimize its performance over high speed networks by utilizing the new features provided by the IPv6 protocol. The tool was used to measure performance between two servers hosted on the University of Missouri Research Network (Rnet) for both IPv4 and IPv6 protocols as a comparative analysis. These two hosts are on different network segments and have to pass through a router to determine the routing. Chapter 6 provided the statistics of the measurement results and the corresponding analysis of the performance of both IP protocols to

demonstrate the utility of this tool. Based on these measurement results, it was observed that both IP protocols offered comparable performance in terms bandwidths. However it was observed that IPv6 protocol offered slightly higher latency compared to IPv4 protocol.

There are numerous improvements and enhancements that can be made to both the middleware application and the measurement tool *netaware* in order to make them better and provide more functionality as part of continuous improvement. The first enhancement is to provide graphical interface to display measured performance for both IP protocols for better visual display and comparison. This enhancement can range from displaying a simple progress bar for each of the measurements in progress to displaying charts of the measurement results over both IP protocols for better comparative analysis.

The second enhancement is to add other performance parameters namely packet loss, CPU utilization and jitter for both IPv4 and IPv6 protocols. Measuring these performance parameters at the application level can help to gain more information about the underlying network for applications of a specific type. For example, measuring packet loss may help an application involved in streaming video content to choose a better IP protocol among the two with comparable bandwidths or latencies. Similarly, measuring CPU utilization may help applications which are CPU intensive while transferring data over the network.

The third enhancement is to allow the user an option to configure few of the measurement parameters. This can be achieved by providing more options to

the tool and also through a user configurable file that the measurement tool can access at run time. For example, as mentioned earlier, the tool can provide additional options to configure the delay between two measurement sessions for BTC and also to configure the packet size used for the latency measurement. Similarly, a user configurable file can be used to configure the port numbers for latency, BTC or capacity measurement.

The fourth enhancement is to provide an option to use the tool in a non cooperative environment to measure the performance parameters. This may help analyze the network paths that do not provide access to the tool to be run at the receiver (sink) end. This involves the use of Negative Acknowledgements, TCP SYNC-RST packets, or ICMP packets for both latency and bandwidth measurements. Additionally, latency can also be measured passively by using TCP timestamps of the application packets received [16]. However, the use of these protocols may be limited since many network servers block the use of layer 3 protocols for diagnosis and performance measurement. Also the accuracy of the measured performance may vary based on the limited response received from the sink.

The fifth enhancement is to add a decision making capability within the middleware application to decide to use a better IP protocol on the fly. The middleware application will measure the performance in background while the application is in execution and correspondingly switches between the IP protocols, without impacting the application based on a user-defined policy. This

can be achieved by integrating the existing measurement tool with the middleware application to measure the performance parameters in background.

With the advent of new technologies, ideas and growth in the use of digital media, more and more data is transmitted over the Internet than ever before. With this growing traffic, the need to measure performance to get explicit feedback from the network has gained more importance. Given the complex nature of the growing traffic and also the decentralized nature of the Internet, there is a constant challenge as to how one can effectively and accurately measure network performance and correspondingly, make improvements. With the combined use of the middleware application and the measurement tool *netaware*, this thesis is one of the constant attempts and efforts to help enhance the network utilization, performance and in a larger picture make applications network aware.

As part of this thesis, I got an opportunity to learn and gain good insight on different network performance measurement techniques, various parameters on which theses techniques are classified and corresponding challenges involved. This thesis also helped me to gain detailed knowledge and importance on the use of various network protocols especially IPv4 and IPv6 protocols. Apart from gaining theoretical knowledge on various aspects and advanced concepts of computer networks, I also got good hands on experience in network programming, use of advanced OS concepts and other network related tools. This thesis and other related course work has given me a good exposure to wide range of areas in computer science. This is not only helping me in tracking and

troubleshooting few of the issues related to the networks, application servers and server load balancers in my current professional job but will also be helpful in my future professional career to understand and adopt new technologies.

Appendix A – Internet Protocol Version 4 (IPv4)

Internet Protocol Version 4 (IPv4)

The IP layer also sometimes called the network layer prefixes its own header to the datagram/packet passed down from the upper layer namely Transport layer. The format of the IPv4 header is as shown in Figure A.1.

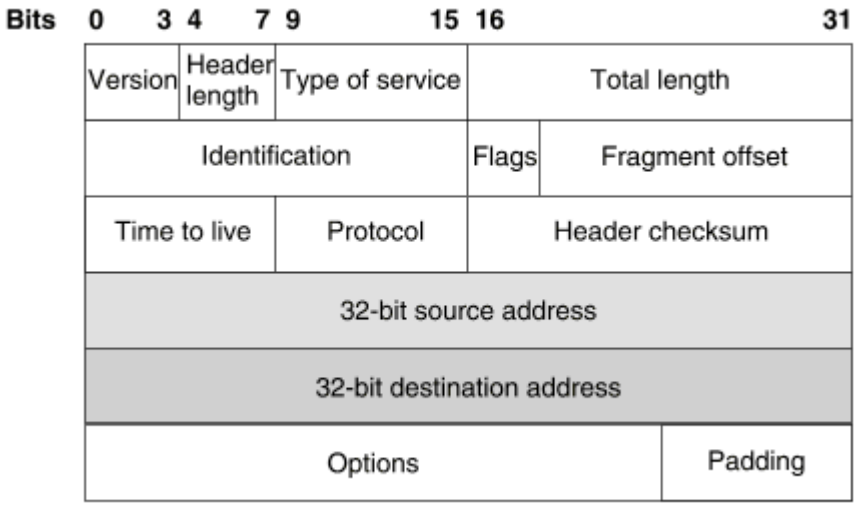


Figure A.1: IPv4 Header [4]

The IPv4 header consists of 14 fields, out of which 13 are required. The Option field as the name indicates is an optional field. It indicates the list of optional specifications for source routing, security restrictions, so on and so forth. The Hlen field is used to check the presence or absence of options. The options can be upto 40 bytes in addition to the minimum IP header of 20 bytes to a total of 60

bytes, the maximum length an IP header can have. Options are used fairly rarely. **Padding** in form of null bytes is added to the end of a packet. As required by the header length field, padding is added to make header an integral multiple of 32 bytes. A detailed description on significance of each of the header fields can be found in reference [4].

As part of IPv4 versus IPv6 comparison, there are few things that need to be pointed out about each of the IP headers. As compared to IPv4 header, IPv6 header contains fewer fields and is designed to be simpler to remove unnecessary functionality from the IP header. This helps to improve and also simplify the processing the IP packets by the routers. For example, if the Option field is present in IPv4 header, every router has to check whether any of the Option field is relevant to its processing. Moreover, the option field appears at the end of IPv4 header. Thus the router has to parse the entire header until it finds out whether the option fields are set. With IPv6, options are part of the extension headers that appear in specific order if present after the main header. This helps improve the efficiency, as it will quickly help routers to identify whether options are relevant to it. In most of the cases, option fields are not relevant to intermediate routers, except the final router or destination itself [4]. Furthermore, removing the optional fields in main header and correspondingly reducing number of fields in IPv6 header makes it a fixed length header. The IPv6 header is 40 bytes in length while IPv4 header ranges from 20 bytes minimum to maximum of 60 bytes. Fixed length helps router to process more packets as options are processed only when optional headers are present. The Minimum Transfer Unit (MTU) for IPv6

is now 1,280 bytes as opposed to IPv4's 64 bytes [11]. This helps reduce packet fragmentation at routers and further increase the efficiency by transferring more data per packet. Computing checksum is also improved with new IPv6 header format. The need to compute checksum for an IP header at each router is now removed increasing the efficiency further. Typically layer 2 (the link layer) that encapsulates the IP packet detects most of the errors using Cyclic Redundancy Check (CRC) technique, while end-to-end transport layer checksum detects most other errors. The working of CRC technique is currently out of scope for this thesis. Furthermore, with the option of sending jumbograms over high speed IPv6 networks that support high MTU, an application can send packets of size upto 4,294,967,295 (2^{32}-1) bytes compared to the maximum packet size of 65,535 bytes (2^{16}-1) over IPv4 networks. This drastically reduces the IP header overhead per packet and correspondingly increases the network utilization and efficiency. Over all, based on the underlying network infrastructure and support for IPv6, an application can reduce the IP overhead and fragmentation at intermediate routers to increase the efficiency by leveraging IPv6 features, especially for applications involved in large data transfers.

IPv4 Addressing

To identify a host on a network uniquely, each host is assigned an IP address. IPv4 addresses are 32-bits (4 byte) wide. So there are 2^{32}-1, approximately 4 billion, IPv4 addresses. At a given time no two hosts should have the same IP address. A "DOTTED DECIMAL" is followed for representation. Each of the byte

in this 4 bytes address is separated by 'dots" or "periods" and represented as a decimal number (base 10). The possible range of each byte is from 0 to 255. Every address is a combination of network and host identifications and thus hierarchical in nature. This is due to the fact that each physical network has its own unique network address and each host has its own unique address. As mentioned the source and destination address in an IP header uniquely identifies the sender and receiver.

The IP addresses are divided into classes namely A, B and C, which defines different sized network and host part. Class D specifies a multicast group and class E that are currently unused. In all the cases the addresses are 32-bits long. The five classes are as shown in Figure A.2 while Table A.1 shows the address range of all the classes.

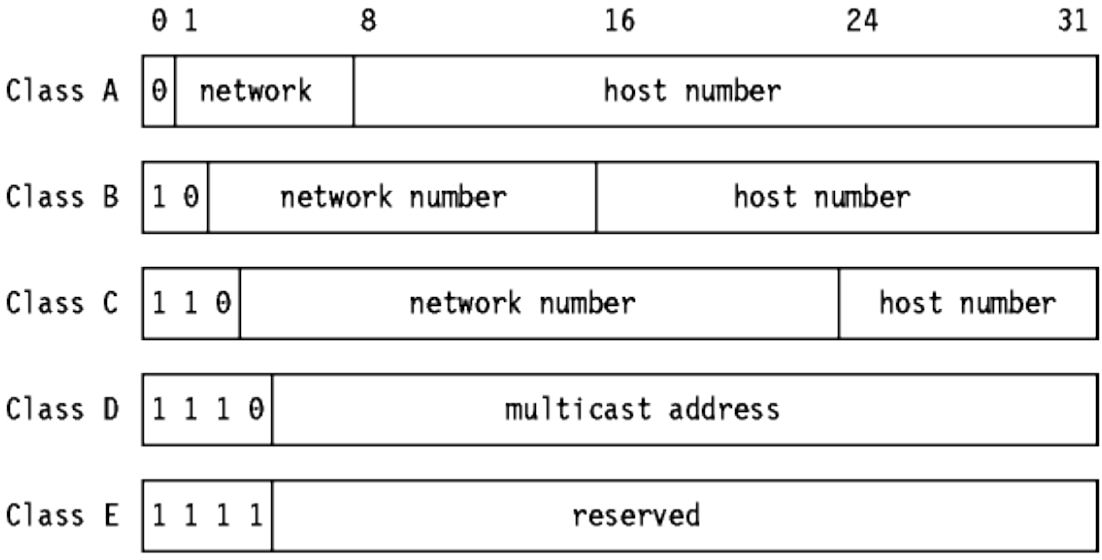


Figure A.2: IPv4 Address Classes [4]

Class	Leftmost 5-bits	Start Address	Finish Address	Size of Network number bits	Size of host number bits
A	0xxxx	0.0.0.0	127.255.255.255	8	24
B	10xxx	128.0.0.0	191.255.255.255	16	16
C	110xx	192.0.0.0.	223.255.255.255	24	8
D	1110x	224.0.0.0.	239.255.255.255	Not defined	Not defined
E	11110	240.0.0.0	255.255.255.255	Not defined	Not defined

Table A.1: IPv4 Class Range [4]

Appendix B – Middleware Application Functions and Constants

Middleware Application functions

The socket API provides various system calls to operate on sockets for inter process communication across the network. Chapter 4 provided details on the middleware application design and its capability to switch between IP protocols on the fly. This section provides an overview of the basic socket API system calls and the corresponding middleware application functions, which internally use these socket API system calls. For more details on each of the system call, their parameter values and list of error messages returned, please refer to Unix Programmer Manuals as further source of reference. All these system calls are included in header file **"sys/socket.h"** and their corresponding parameter values are in the header file **"sys/types.h"** except for *close(), read()* and *write()* which are included in the header file **"unistd.h"**. The middleware application functions are declared under the header file **"mysocket.h"**. This header file is further included under the middleware application header file **"commonhdr.h"**. This header file is an interface to the middleware application functions and its constants that need to be included in an application source code using the middleware application. A complied version of the middleware application source is made available through the shared library ***libnetaware.so***. This shared library needs to be specified when the code is

linked to the application using the middleware application so the system knows to look for the shared library when the program commences execution.

1. Socket():

   System call:

   ```
   int socket(int domain, int type, int protocol);
   ```

   The system call socket() creates an end point for inter process communication and returns a socket identifier/descriptor for that end point to operate on. The input parameters determine the nature of communication and protocol followed. This is the first system call that needs to be called to create a socket descriptor to be used by other socket system calls.

   The *domain* parameter specifies a communication domain. This selects the protocol family for example IPv4/IPv6, which will be used for communication. The *type* specifies communication semantics. In simple words it specifies the communication protocol over IP layer. For example if it is connection-oriented, use value SOCK_STREAM (TCP) while for connectionless, unreliable communication use SOCK_DGRAM (UDP) and so on. The *protocol* specifies a particular protocol to be used with socket and typically the value is 0 as there is only one protocol within a *domain*. On success, *socket()* returns a new socket descriptor. On error, the return code -1 is returned and system's *errno* is set appropriately. This system call is internally called by middleware application functions *fn_socket()*. The function call of *fn_socket*() is as described below.

```
int fn_socket (st_mySocket *p_mysocket,int st_sockettype,
int st_ipversion);
```

This function is the starting point of the middleware application. The first parameter *p_mysocket* is the address of the socket structure that holds the connection properties of the IP protocol and is used across all the middleware application functions. Based on the values assigned to its members, *fn_socket()* creates a socket connection for either IPv4 or IPv6 or both the protocols for either TCP or UDP. Socket structure members are further described in later section of this Appendix. The parameter *st_sockettype* specifies to create a socket connection for either TCP or UDP and its value is either the middleware application constant SOCKET_TCP or SOCKET_UDP respectively. The parameter *st_ipversion* specifies to create socket connection for IPv4, IPv6 or both IP protocols. The corresponding middleware application constants to use are IPV_4, IPV_6 and IP_BOTH respectively. Based on these input parameters, *fn_socket()* internally calls the system call *socket()* with appropriate values for parameters *domain, type* and *protocol* to create socket connections. The socket descriptor returned by the call to each *socket()* is then stored in the socket structure for further use. The default IP protocol to be used for communication is IPv4 when the value of *st_ipversion* is IPV_BOTH unless the application explicitly specifies to use IPv6 protocol. *fn_socket()* further calls the other middleware application functions namely *fn_bind(), fn_connect()* and *fn_listen()* based on whether an application is a

server or a client when the parameter *st_sockettype* has the value of SOCKET_TCP (TCP). For a server, *fn_socket()* calls *fn_bind()* and *fn_listen()* while for a client it calls *fn_connect()* for TCP. For UDP *fn_socket()* calls *fn_bind()* irrespective of whether it is a server or client. On success it returns 0 (Middleware application constant SUCCESS) or returns error messages for errors encountered at each execution step. A list of middleware application constants is used to denote each error number. The middleware application function *fn_getErrorString()* is used to get the corresponding error message error numbers returned by *fn_socket()*. An example below provides on how to use the middleware application function *fn_socket()*.

```
st_mySocket vl_dataSocket;

vl_dataSocket.vst_isserver = TRUE;
strcpy(vl_dataSocket.vst_serverName,
'web.rnet.missouri.edu');
vl_dataSocket.vst_myPort = 10001;
vl_dataSocket.vst_remotePort = 10000;
vl_dataSocket.vst_version_use = IPV_6;

int vl_Error =
fn_socket(&vl_dataSocket,SOCKET_TCP,IPV_BOTH);
if(vlError != SUCCESS){

fprintf(stderr,"Error:%s\n",fn_getErrorString(vlError));
}
```

2. Close():

System call:
int close(int fd);

The system call *close()* closes the file descriptor passed to it. Calling close on a socket descriptor closes the end point and terminates any further communication on it. On success, *close()* returns 0. On error, the return code -1 is returned and the system's *errno* is set appropriately. This system call is internally called by middleware application functions *fn_close()*. The function call of *fn_close()* is described below.

```
int fn_close(st_mySocket p_mysocket);
```

The parameter *p_mysocket* is the address of the socket structure that holds the connection properties of the IP protocol. Based on the values initialized for *p_mySocket, fn_close()* calls the system call *close()* to close socket connections for either IPv4 ,IPv6 or both the protocols. On error, the function returns the corresponding error message. A return code of 0 indicates success (Middleware application constant SUCCESS).

3. Bind():

   System call:

```
int bind(int sockfd, const struct sockaddr *my_addr,
socklen_t addrlen);
```

The system call *bind()* binds the socket descriptor *sockfd* with local address *my_addr* of length *addrlen*. The socket structure *sockaddr* holds the

port number, protocol family address (IPv4 or IPv6) to bind *sockfd* to. It is necessary to bind the socket descriptor of type SOCK_STREAM to an address (either IPv4 or IPv6) in order to accept incoming connections (system call *accept()*). On success, *bind()* returns 0. On error, the return code -1 is returned and system's *errno* is set appropriately. This system call is internally called by middleware application function *fn_bind()*. The function call of *fn_bind*() is as described below.

```
int fn_bind(st_mySocket *p_mysocket, int ipversion);
```

The parameter *p_mysocket* is the address to the socket structure *st_mySocket* while the parameter *ipversion* specifies the IP protocol version that needs to bind. *ipversion* value can be either IPV_4 or IPV_6. Based on the values of the member variable of the socket structure and IP version passed to it, *fn_bind()* calls *bind()* to bind the appropriate socket descriptor and the address stored in *p_mysocket*. On error, *fn_bind()* returns the corresponding error message. A return code of 0 indicates success (Middleware application constant SUCCESS). The function is internally called by middleware application function *fn_socket()* for server application. An application can also explicitly call *fn_bind()* for a client application if needed by passing the same socket structure as that for *fn_socket()* called previously with appropriate values.

4. Connect():

System call:

```
int connect(int sockfd, const struct sockaddr *serv_addr,
socklen_t addrlen);
```

The system call *connect()* connects the socket descriptor *sockfd* to the address referred by *serv_addr* whose length is specified by *addrlen*. The format of the address *serv_addr* is determined based on what *domain* was used while creating socket through *socket()* call. For *sockfd* of the type SOCK_STREAM (TCP), then it attempts to create connection bounded to address *serv_addr* and can be called only once. For socket of type SOCK_DGRAM (UDP), *serv_addr* specifies default address to which datagrams are sent. c*onnect()* is typically called by the client to connect to the socket end point on the server side. On success, *connect()* returns 0. On error, the return code -1 is returned and system's *errno* is set appropriately. This system call is internally called by middleware application function *fn_connect()*. The function call of *fn_connect*() is as described below.

```
int fn_connect(st_mySocket *p_mysocket, int ipversion);
```

The parameter *p_mysocket* is the address to the socket structure *st_mySocket* while the parameter *ipversion* specifies the IP protocol version whose socket descriptor and corresponding bounded address to needs to connect. *ipversion* value can be either IPV_4 or IPV_6. Based on the values of

the member variable of the socket structure and IP version passed to it, *fn_connect()* calls *connect()* to connect the appropriate socket descriptor and the address stored in *p_mysocket* to the server side socket and bounded to an address. On error, the function returns the corresponding error message. A return code of 0 indicates success (Middleware application constant SUCCESS). The function is internally called by middleware application function *fn_socket()* for a client application.

5. Listen():

System call:

```
int listen(int sockfd, int backlog);
```

The system call *listen()* is used to listen on incoming socket connections to be accepted. It represents willingness to accept connections and is usually called by the server. It is typically used for *sockfd* of type SOCK_STREAM or SOCK_SEQPACKET as these are connection-oriented protocols. The parameter *backlog* specifies the queue length for completely established sockets waiting to be accepted. If the queue is full, the client trying to connect may receive the error ECONNREFUSED or if the underlying protocol supports retransmission, the request may be ignored so that retries succeed. On success, *listen()* returns 0. On error, the return code -1 is returned and system's *errno* is set appropriately. This system call is internally called by

middleware application functions *fn_listen()*. The function call of *fn_listen()* is as described below.

```
int fn_listen(st_mySocket *p_mysocket, int ipversion, int
p_backlog);
```

The parameter *p_mysocket* is the address to the socket structure *st_mySocket*. The parameter *ipversion* specifies which IP protocol version to listen on for incoming connections. *ipversion* value can be either IPV_4 or IPV_6. The parameter *p_backlog* specifies the queue length for completely established sockets waiting to be accepted. Based on the values of the member variables of the socket structure and the IP version passed to it, *fn_listen()* calls *listen()* to listen on incoming connection for an IP protocol. On error, the function returns the corresponding error message. A return code of 0 indicates success (Middleware application constant SUCCESS). The function is internally called by middleware application function *fn_socket()* for a server application.

6. Accept():

   System call:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t
*addrlen);
```

The system call *accept()* accepts incoming connection on *sockfd* which is in listening state after calling *listen()* and bound to local address using *bind()*. The system call s*ocket(), bind()* and *listen()* needs to be called in order as specified before calling *accept()* and is to be used with connection-oriented socket types (SOCK_STREAM or SOCK_SEQPACKET). Calling *accept()* extracts the first connection in queue of pending connections, creating a new socket descriptor. The function returns the newly created socket descriptor on success. The *sockfd* state is not changed and is still listens on incoming connections. On accepting a connection, the parameter *addr* is filled with address of the peer socket (client). The parameter *addrlen* then contains the actual length of the address *addr* filled. *accept()* is a blocking I/O in nature and blocks until a connection is present. It can be marked as non-blocking in which case it returns immediately with error EAGAIN if there are no pending connections. Alternately system calls *select()* or *poll()* can be used to detect any incoming connections before calling *accept()*. On success, *accept()* returns non-negative socket descriptor of the accepted socket. On error, the return code -1 is returned and system's *errno* is set appropriately. This system call is internally called by middleware application functions *fn_accept()*. The function call of *fn_accept*() is as described below

```
int fn_accept(st_mySocket *p_mysocket);
```

The parameter *p_mysocket* is the address of the socket structure *st_mySocket*. Based on the values initialized for *st_mySocket, fn_accept()* calls the system call *accept()* to accept the incoming socket connections for either IPv4, IPv6 or both the protocols. Once an incoming connection request is accepted, the new socket descriptor is stored in the structure *st_mySocket* and returns the socket descriptor of the IP version specified to be used. The new socket descriptor returned can then be assigned to one of the members of socket structure for further use. On error, the function returns the corresponding error message. A return code of 0 indicates success (Middleware application constant SUCCESS). An example below shows how to use *fn_accept()* in an application. The socket structure member *socket_id* holds the socket descriptor of the IP protocol version specified to be used.

```
int vlError =
fn_socket(&vl_dataSocket,SOCKET_TCP,ip_version);
if(vlError != SUCCESS){

fprintf(stderr,"Error:%s\n",fn_getErrorString(vlError));
     exit(EXIT_FAILURE);
}
 fprintf(stdout,"Socket Initialization done\n");

vl_dataSocket.socket_id = fn_accept(&vl_dataSocket);
if(vl_dataSocket.socket_id <=0){
   fprintf(stderr,"Error in accepting the connection .
   err no:%s\n Hence Exiting\n",
   fn_strerror_r(errno,err_buff));
   fn_close(vl_dataSocket);
   exit(EXIT_FAILURE);
}
```

7. Write()/Send()/Sendto():

   System call:

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t send(int fd, const void *buf, size_t len, int
flags);
ssize_t sendto(int fd, const void *buf, size_t len, int
flags, const struct sockaddr *to, socklen_t tolen);
```

   The above three system calls are used to write data to socket descriptor *fd*. *write()* is more general in sense it works with any kind of descriptor while *send()* and *sendto()* is specific to operate on socket descriptors. *send()* is used with connection-oriented socket while *sendto()* is used on connectionless socket. All three system calls are blocking I/O unless marked non-blocking and waits on until the data is written to the output stream. Alternately system calls *select()* or *poll()* can be used to detect for any presence of data before calling these system calls. On success, these system calls returns number of bytes sent. On error -1 is returned and system's *errno* is set appropriately. These system calls are internally called by middleware application functions *fn_write()*, *fn_send()* and *fn_sendto()* respectively. The function calls for *fn_send()/fn_write()/fn_sendto()* are as described below.

```
ssize_t  fn_send(st_mySocket  *p_mysocket,void  *p_buffer,
size_t p_len,int p_flags);
ssize_t  fn_write(st_mySocket  *p_mysocket,void  *p_buffer,
size_t p_len,int p_flags);
ssize_t  fn_sendto(st_mySocket  p_mysocket,void  *p_buffer,
size_t p_len,struct sockaddr *p_toaddr,int p_flags);
```

The first parameter *p_mysocket* represents the socket structure *st_mySocket*. The parameter *p_buffer* is the address of the buffer holding the data to be sent, while the parameter *p_len* represents the size of data to be sent. The parameter *p_toaddr* for the middleware application function *fn_sendto()* represents the address of the remote host that needs to receive the data. The parameter *p_flags* represents the parameter *flags* of the base system calls *send()/write()/sendto()*. All these parameters are similar to the parameters of the corresponding socket API system calls. Based on the values assigned to the members variables of the socket structure *p_mysocket*, *fn_send()/fn_sendto()/fn_write()* sends the data over either IPv4 or IPv6 protocol. If the socket structure is marked to have socket connections for both IP protocols, the corresponding write operations can leverage the use of switching capability built within these middleware application functions *fn_write()* and *fn_send()* for TCP. Chapter 4 provides more details on the *fn_send()/fn_write()* algorithm followed while switching the IP protocol when the request is received. The value NO_DATA_TX for the parameter

p_flags can be used to indicate not to use the switching capability of the middleware application even if the socket structure *p_mysocket* is marked to have socket connection for both IP protocols. On success, all three middleware application functions return the amount of data sent in bytes. On error -1 is returned. The algorithm followed by *fn_send()/fn_write()* to switch the IP protocol on the fly is as below.

```
size_t fn_send(st_mysocket *p_mysocket,void *p_buffer,size_t p_len, int p_flag){

 Intialize local variables;
 if p_flag != NO_DATA_TX
 then
      /* Structure containing the total packet size and the sentinel */
      declare st_packetData vst_packet;
      calculate total length vl_packetSize to be sent
      vl_packetSize = sizeof(st_packetData) + (int)p_len ;

     Initialize the member variables of vst_packet to '\0';
    Copy the total length vl_packetSize to vst_packet;

     Mutex lock
       If vg_ctrlpacket_recv set
      then
         copy sentinel X to vst_packet;
        vl_change_sock = 1;
      end if;
    Mutex Unlock

    Copy the structure vst_packet and application data contained in p_buffer to
    buffer vl_bufData
    size_t vl_err  = send(p_mysocket->socket_id, vl_bufData, vl_packetSize, p_flag);
    if vl_err is not -1
    then
            vl_err = calculate the size of actual application data sent;

            if vl_change_sock set
            then
                    check the socket type to change to;
```

```
                    switch the socket descriptor to new protocol;
                     p_mysocket->socket_id = p_mysocket->sock_id[i];
                    Mutex lock
                       reset vg_ctrlpacket_recv;
                    Mutex Unlock
            end if;
    else
        vl_err = send(p_mysocket->socket_id, p_buffer, p_len, p_flag);
    end if;
    return vl_err;
}
```

8. Read()/Recv()/Recvfrom():

   System call:

```
ssize_t read(int fd, void *buf, size_t count);

ssize_t recv(int fd, void *buf, size_t len, int flags);

ssize_t recvfrom(int fd, void *buf, size_t len,int

flags, struct sockaddr *from, socklen_t *fromlen);
```

   The above system calls are used to read incoming data on a socket descriptor *fd*. *read()* is more general in sense it works with any kind of descriptor while *recv()* and *recvfrom()* is specific to operate on socket descriptors. *recv()* is used with connection-oriented sockets while *recvfrom()* is used on connectionless sockets though *recv()* is identical to *recvfrom()* when *from* parameter is NULL. All the system calls are blocking I/O as it waits on data until received. It can also be marked non-blocking. Alternately system calls *select()* or *poll()* can be used to detect for presence of any data before calling these system calls. On success, these system calls returns the number of bytes

read. On error -1 is returned and system's *errno* is set appropriately. The error code of 0 is returned when the peer socket has performed orderly shutdown. These system calls are internally called by middleware application functions *fn_read()*, *fn_recv()* and *fn_recvfrom()* respectively. The function calls of *fn_recv()/fn_read()/fn_recvfrom()* are as described below.

```
ssize_t fn_recv(st_mySocket *p_mysocket,void *p_buffer,
size_t p_len,int p_flags);
ssize_t fn_read(st_mySocket *p_mysocket,void *p_buffer,
size_t p_len,int p_flags);
ssize_t fn_recvfrom(st_mySocket p_mysocket,void
*p_buffer, size_t p_len,struct sockaddr *p_frmaddr,int
p_flags);
```

The first parameter *p_mysocket* represents the socket structure *st_mySocket*. The parameter *p_buffer* is the address of the buffer the data is read into, while the parameter *p_len* represents the size of data to be read. The parameter *p_frmaddr* for the middleware application function *fn_recvfrom()* represents the address of the remote host address sending data. The parameter *p_flags* represents the values to be assigned to the parameter *flags* for the system call *recvfrom()*. All these parameters are similar to the parameters of the corresponding socket API system calls. Based on the values assigned to the members variables of the socket structure

p_mysocket, *fn_recv()/fn_recvfrom()/fn_read()* reads the data for either IPv4 or IPv6 protocol. If the socket structure is marked to have socket connections for both IP protocols, the corresponding read operations can leverage the use of switching capability built within the middleware application functions *fn_read()* and *fn_recv()* for TCP. Chapter 4 provides more details on *fn_recv()/fn_read()* algorithm while switching the IP protocol when the request is received. The value NO_DATA_TX for the parameter *p_flags* can be used to indicate not to use the switching capability of the middleware application even if the socket structure *p_mysocket* is marked to have socket connection for both IP protocols. On success, all three middleware application functions return the amount of data read in bytes. On error, -1 is returned. The algorithm followed by *fn_recv()/fn_read()* to switch the IP protocol on the fly is as below.

```
size_t fn_recv(st_mysocket *p_mysocket,void *p_buffer,size_t p_len, int p_flag){

  Intialize local variables;
  if p_flag != NO_DATA_TX
  then
       /* Structure containing the total packet size and the sentinel*/
       declare st_packetData vst_packet;
       calculate total length to be sent
       vl_packetSize = sizeof(st_packetData) + (int)p_len ;

       Initialize the member variables of vst_packet to '\0';
      size_t vl_err = recv(p_mysocket->socket_id, vl_bufData, vl_packetSize, p_flag);

      if vl_err is -1
      then
              return vl_err;
      else
```

```
            vl_cntData= vl_err;
        end if;
        Assign vst_packet.vst_size with the size of packet contained in vl_bufData
        If vl_cntData is less than vst_packet.vst_size
        then
            Keep receiving the data by calling recv() to get the entire packet;
            vl_cntDataTot = recv(p_mysocket->socket_id, vl_bufDataTemp,
                                vst_packet.vst_size-vl_cnt_Data, p_flags);

        end if
        Copy sentinel contained in vl_bufDataTemp to vst_packet;
        Copy the data contained in vl_bufDataTemp toinput buffer p_buffer;
         if setinel value is X
        then
            p_mysocket->socket_id = p_mysocket->sock_id[i];
        end if;

        if vl_err is not -1
        then
            vl_err = calculate the size of actual application data received;
        end if;

     else
         vl_err = recv(p_mysocket->socket_id, p_buffer, p_len, p_flag);
    end if;
    return vl_err;
}
```

The section "*Middleware Application Example"* provides an example on how to make use of the middleware application functions and constants in an application.

Socket Structure st_mySocket

The socket structure *st_mySocket* is as shown below.

```
typedef struct
{
 /* Members to be initialized */
     int vst_socketType;
     int vst_iptype;
     int vst_myPort;
     int vst_remotePort;
     char vst_serverName[MAX_ADDRESS];
     int vst_isserver;
     int vst_version_use
     int vst_apptype;

 /* Members initialized by the middleware application */
     pthread_t vst_id_thread;
     char vst_nm_thread[MAX_NAME_SIZE];
     const char *vst_myPort_str;
     const char *vst_remotePort_str;
     char vst_hostName[MAX_ADDRESS];
     int socket_id;
     int sock_id[3];
     int vst_socketId;
     int vst_socketId6;
     int vst_ttl;
     struct sockaddr_in vst_myAddr;
     struct sockaddr_in vst_remoteAddr;
     struct sockaddr_in6 vst_myAddr6;
     struct sockaddr_in6 vst_remoteAddr6;
     struct addrinfo vst_myhints;
     struct addrinfo *vst_myres;
     struct addrinfo vst_myhints6;
     struct addrinfo *vst_myres6;
     socklen_t vst_addrlen;
     struct sockaddr_in vst_addr;
     struct sockaddr_in6 vst_addr6;
     int sock_alive[3];

} st_mySocket;
```

1. Members to be initialized by an application

Structure Members	Description	Values (middleware application constants)
st_socketType	Indicates whether to create TCP or UDP connections	SOCKET_TCP SOCKET_UDP
vst_iptype	Indicates whether the socket structure *st_mySocket* holds either IPv4, IPv6 or both connections	IPV_4 IPV_6 IPV_BOTH
vst_myPort	Port to which socket should be bound to, on server side	
vst_remotePort	Remote port to which socket should be connected to, on client side	
vst_serverName	Server Name. Maximum size 256 characters	
vst_isserver	Flag that indicates whether an application is a server or a client	TRUE FALSE
vst_version_use	Indicates the preferred IP protocol to be used	IPV_4 IPV_6

2. Members initialized by the middleware application function *fn_socket()*

Structure Members	Description
vst_id_thread	Current thread ID
vst_nm_thread	Current thread name
vst_myPort_str	String representation of Port to which socket should be bound to, on server side. Used by socket API *getaddrinfo()*
vst_remotePort_str	String representation of remote port to which socket should be connected to, on client side. Used by socket API getaddrinfo()
vst_hostName	Host Name. Filled in by function *gethostname().* Max length 128
socket_id	Current socket ID in use. This ID keeps track of the current IP protocol in use
sock_id	Array to hold the accepted socket IDs for each IP protocol. Use by the middleware application function *fn_accept()*. Maximum length 3
vst_socketId	IPv4 socket ID returned by socket API call *socket().* Internally called by *fn_socket()*

vst_socketId6	IPv6 socket ID returned by socket API call *socket()*. Internally called by *fn_socket()*
vst_ttl	Time to live for the packets transmitted by the socket (future use)
vst_myAddr	IPv4 Structure sockaddr_in that holds the host address information
vst_remoteAddr	IPv4 Structure sockaddr_in that holds the remote host address information
vst_myAddr6	IPv6 Structure sockaddr_in that holds the host address information
vst_remoteAddr6	IPv6 Structure sockaddr_in that holds the remote host address information
vst_myhints	IPv4 structure addrinfo that indicates the IP protocol family and type of socket, TCP or UDP. This structure is initialized with values based on the values passed to middleware application function *fn_socket().* This structure is used by socket API function *getaddrinfo()* called by *fn_socket()*
vst_myres	IPv4 structure addrinfo to hold actual host address
vst_myhints6	IPv6 structure addrinfo that indicates the IP protocol family and type of socket, TCP or UDP. This structure is initialized with values based on the values passed to the middleware application function *fn_socket().* This structure is used by socket API function *getaddrinfo()* called by *fn_socket()*
vst_myres6	IPv6 structure addrinfo to hold actual host address
vst_addrlen	socket address length for both IPv4 and IPv6 protocols
vst_addr	IPv4 structure sockaddr_in that holds generic address to be used by the middleware application function *fn_accept()*
vst_addr6	IPv6 structure sockaddr_in that holds generic address to be used by the middleware application function *fn_accept()*
sock_alive	Flag to check whether the socket is alive

Middleware Application Example

This section provides an example on how to use the middleware application functions and constants in an application. The code snippet below is taken from the measurement tool *netaware* that uses the middleware application functions and constants to send control information over TCP socket connection while measuring the network path capacity. The code is slightly modified to focus on

the middleware application utility to create TCP connections for both IP protocols and specify the default IP protocol to be used. The measurement tool itself doesn't switch between the IP protocols on the fly to send control information.

```
     /* Include the middleware application header file to
     access middleware application functions and constants
     */
1    #include "/home/hshah/middleware/commonhdr.h"

     /* Declare socket structure */
2    st_mySocket sndrTCPSock;
3    int err = 0;

     /* Control stream: TCP connection */
     /* Initialize socket structure to create socket
     connection */

4    bzero((char *)&sndrTCPSock,sizeof(sndrTCPSock));
5    sndrTCPSock.vst_isserver = TRUE;
6    sndrTCPSock.vst_myPort = 10001;
7    sndrTCPSock.vst_remotePort = 10000;
8    sndrTCPSock.vst_version_use = IPV_4;
9    strcpy(sndrTCPSock.vst_serverName, host_name);

10   if((err = fn_socket(&sndrTCPSock,SOCKET_TCP,IPV_BOTH))
     != SUCCESS){
11        fprintf(stderr,"Unable to create socket
          connection%s\n", fn_getErrorString(err));
12        exit(-1);
13   }

     /* Accept connection from client */
14   sndrTCPSock.socket_id = fn_accept(&sndrTCPSock);

15   if (sndrTCPSock.socket_id < 0) {
16        perror("Unable to accept connections\n");
17        exit(-1);
18   }
```

```
     /* Read data */
19   int err = fn_read(&sndrTCPSock,(char
     *)&bw_lo,sizeof(bw_lo),NO_DATA_TX);
20   if(err == -1) {
21        perror("Unable to read\n");
22        exit(-1);
23   }


     /* Close socket connection */
24   fn_close(sndrTCPSock);
```

The middleware application header file “commonhdr.h” that provides interface to the middleware application functions and constants is included at line number 1. This header file is necessary for compilation. The measurement tool is linked to middleware application shared library *libnetaware.so* at run time by specifying library path in its make file. Currently, this file is under the folder “middleware”. However, this header file can be under any of the specific locations where the compiler searches by default. The variable *sndrTCPSock* at line number 2 represents the middleware application socket structure *st_mySocket*. This structure members needs to be initialized before passing it to the middleware application function *fn_socket()*. Line numbers 4 to 9 shows how this structure’s members are initialized. The member *vst_isserver* at line number 5 specifies to the middleware application (*fn_socket()*) whether an application is a server or a client. This is needed in order to listen and accept incoming connections for a server or to connect to a socket on server as a client for TCP. The value can be either TRUE or FALSE. Both these values are middleware application constants. The member *vst_myPort* and *vst_remotePort* at line

number 6 and 7 specifies the server port and the remote client port numbers respectively. The socket IDs created are then bound to these port numbers. The member *vst_version_use* at line number 8 specifies the preferred IP protocol to be used once the TCP connection is established for both IP protocols. The value can be either IPV_4 or IPV_6. Both these values are middleware application constants. Since the value IPV_BOTH is passed to the function *fn_socket()* at line number 10, the socket structure *sndrTCPSock* is marked to have both IP connections. The application can specify to use either IPv4 or IPv6 as preferred protocol for communication until an event to switch the IP protocol is received. The default protocol to be used is IPv4 if the application does not specify the preferred IP protocol. Line number 9 copies the host name to the socket member *vst_serverName* for further use. Once these five structure members are initialized, the middleware application function *fn_socket()* is called at line number 10. As mentioned earlier, *fn_socket()* is the starting point of the middleware application. The first parameter passed to *fn_socket()* is the address of the socket structure *sndrTCPSock* initialized in earlier steps. The second parameter value SOCKET_TCP specifies *fn_socket()* to create TCP socket connections. This parameter value can be either SOCKET_TCP or SOCKET_UDP for TCP and UDP respectively. Both these values are also middleware application constants. The third parameter value, IPV_BOTH, specifies *fn_socket()* to create socket connections for both IP protocols. To strictly create and use only IPv4 connection, use the constant IPV_4 instead of IPV_BOTH as the input parameter. Similarly to use only IPv6 protocol, use the constant IPV_6. All these

values are again middleware application constants. The values passed to *fn_socket()* is then internally assigned to two other *sndrTCPSock* members for further use. The second input parameter value SOCKET_TCP is assigned to *sndrTCPSock* member *vst_socketType*. This marks *sndrTCPSock* to be used for TCP. The third parameter value, IPV_BOTH is assigned to the socket member *vst_type*. This marks *sndrTCPSock* to have both IP connections and is important to use the middleware application switching capability. In short, values assigned to *sndrTCPSock* members and the values passed to the function *fn_socket()* tells middleware application to create TCP connections for both IPv4 and IPv6 protocols with IPv4 being the preferred protocol to be used for communication. The socket structure members are then collectively used by the other middleware application functions to use IPv4 protocol until the request to switch is received. The function *fn_socket()* internally calls the socket API call *socket()* to create socket connections and correspondingly get socket descriptors for both IPv4 and IPv6 protocols. The socket descriptors are assigned to *sndrTCPSock* members *vst_socketId* and *vst_socketId6* as IPv4 and IPv6 socket descriptors respectively. Since *sndrTCPSock* is marked to be used for TCP connection for a server application, *fn_socket()* internally calls the middleware application functions *fn_bind()* to bind both the socket descriptors (*vst_socketId* and *vst_socketId6*) to the corresponding IP addresses and port numbers. *fn_socket()* internally increments both the input port numbers by 1 to be used for IPv6 protocol. In this specific example, the IPv4 socket descriptor is bound to the server port number 10001 while IPv6 socket descriptor to the server port number 10002. Once

fn_bind() is successful, *fn_socket()* then calls the middleware application function *fn_listen()* to listen on incoming connections for each of the IP protocol. On successfully completion, *fn_socket()* returns the middleware application constant SUCCESS (value 0). For any errors encountered during any of these steps, *fn_socket()* returns corresponding error numbers. The middleware application function *fn_getErrorString() at* line number 11 is to be used to get the corresponding error message.

Once call to *fn_socket()* is successful, the middleware application function *fn_accept()* is called at line number 14. The input to *fn_accept()* is the address to the socket structure *sndrTCPSock* previously passed to the function *fn_socket()*. Since *sndrTCPSock* is marked to have TCP connection for both IP protocols, *fn_accept()* internally calls the socket API system call *accept()* to accept incoming connections for both IP protocols. The important point to be noted is, the client application also needs to specify IPv4 as the preferred protocol to be used and mark its socket structure to have socket connections for both IP protocols. Once the system call *accept()* successfully accepts incoming connections for both IP protocols, the new socket descriptors returned by *accept()* for each IP protocol is then assigned to *sndrTCPSock* member *sock_id*. This member is an array to hold the socket descriptors returned by the system call *accept()*. The IPv4 socket descriptor is stored at the locations *sock_id[0]* while IPv6 socket descriptor at the location *sock_id[1]*. Since *sndrTCPSock* is marked to use IPv4 as the preferred protocol, *fn_accept()* returns the socket descriptor of the accepted IPv4 connection. The socket descriptor returned by

fn_accept() is to be assigned to the socket member *socket_id* for further use. This member holds the socket descriptor of the current IP protocol in use and further used by the middleware application functions *fn_read()/fn_recv()* and *fn_write()/fn_send()*. For any error encountered, *fn_accept()* returns -1 and correspondingly sets the system *errno*.

Once connections are successfully created, bound and accepted, the middleware application functions *fn_read()* is called at line number 19 to read the control data. The first parameter is the address to the socket structure *sndrTCPSock*, the second parameter is address of the buffer to hold the data while the third parameter is the size of the data to be read. *fn_read()* internally calls the system call *read()* on *sndrTCPSock* member *socket_id* to read the data over IPv4 connection. The fourth input parameter to *fn_read()* is the middleware application constant NO_DATA_TX. It specifies *fn_read()* not to switch the IP protocol on the fly even when a request to switch is received. This provides a flexibility as to which read or write operation should an application use the middleware application switching capability even if a socket structures is marked to have both IP protocols. In order to leverage the middleware application switching capability, the fourth input parameter can have any other flag value but the value NO_DATA_TX. Again, it is important that the corresponding middleware application function *fn_write()* at client side should also pass NO_DATA_TX as its fourth parameter. On successful completion, *fn_read()* returns the number of bytes read. For any error encountered, *fn_read()* returns -1 and correspondingly sets the system's error number *errno*. At line number 24,

the middleware application function *fn_close()* is called to close all the socket descriptors stored in the socket structure *sndrTCPSock*. Since *sndrTCPSock* is marked to have socket connections for both IP protocols, it closes all socket descriptors for both IP protocols.

Apart from the socket structure members mentioned so far, the structure members *vst_myAddr, st_myAddr6, vst_myhints* and *vst_myhints6* are also assigned with values and used by the system calls *getaddrinfo()/bind()/connect()*. As mentioned previously these system calls are internally called by the middleware application function *fn_socket()*. The socket structure members *vst_iptype, vst_socketType, vst_version_use* and *socket_id* are of particular importance as they are further used by other middleware application functions. The value IPV_BOTH is important to be used and assigned to *vst_iptype* in order to use the middleware application switching capability. Since every middleware application function has the socket structure as its input parameter, the middleware application functions makes combined use of these five members to decide on their internal working. Chapter 4 provided a high level flow diagram of the function *fn_socket()*. The algorithm used by *fn_send()* and *fn_recv()* on switching the IP protocol is described in their corresponding descriptions in this Appendix. As explained earlier, initializing the socket structure with right values and passing it to the middleware application functions helps an application to deal with minimum code to use sockets. In this specific example, the measurement tool had to just initialize the socket structure

members (line 4 to 9) to create socket connection over IPv4 protocol to be used throughout the application.

Middleware Application constants

This section lists all the constants built as part of the middleware application. These constants are created to ease socket structure (*st_mySocket*) initialization and also to be used as input parameters to the middleware application functions. The list also includes the constants used by the control thread and event handler thread for inter thread communication and the constants used by the measurement tool *netaware* for bandwidth and latency measurements. These are divided based on different context the middleware application uses this constants.

1. Constants to be used to initialize socket structure *st_mySocket*

Constants	Value	Use
SOCKET_SERVER	0	Specifies that an application is a server application. Assigned to the structure member *vst_isseerver*
SOCKET_CLIENT	1	Specifies that an application is a client application. Assigned to the structure member *vst_isseerver*
SOCKET_TCP	0	Specifies to create a TCP socket connection. To be passed as a value to the second input parameter while calling the middleware application function *fn_socket().*
SOCKET_UDP	1	Specifies to create a UDP socket connection. To be passed as a value to the second input parameter while calling the middleware application function *fn_socket().*
SOCKET_MULTICAST	2	Specifies Multicast socket connection. To be passed as a value to the second input parameter for the middleware application function *fn_socket().* Future

		use.
SOCKET_TOOL	3	Specifies not to create control threads when *fn_socket()* is called
SOCKET_APP	4	Specifies to create control threads when *fn_socket()* is called
IPV_BOTH	1	Specifies to create socket connections for both IPv4 and IPv6 protocol. To be passed as a value to the third input parameter while calling the middleware application function *fn_socket().*
IPV_4	4	Specifies to create socket connections for only IPv4. To be passed as a value to the third input parameter while calling the middleware application function *fn_socket().* This constant value is also used to initialize the socket structure member *vst_version_use* to specify the desired IP protocol to be used
IPV_6	6	Specifies to create socket connections for only IPv4. To be passed as a value to the third input parameter while calling the middleware application function *fn_socket().* This constant value is also used to initialize the socket structure member *vst_version_use* to specify the desired IP protocol to be used
NO_DATA_TX	1	Specifies not to switch the socket descriptor (IP protocol) even if the corresponding socket structure *st_mySocket* is marked to have both IP socket connections. To be used by the middleware application functions *write()*/send() and *read()/recv()* .

2. *fn_socket()* error codes:

Constants	Value	Error messages
ERR_UNABLE_TO_GET_HOST NAME	1	Unable to get Hostname
ERR_INVALID_HOST_NAME	2	Invalid host name
ERR_UNABLE_TO_CREATE_SO CKET	3	Unable to create socket
ERR_UNABLE_TO_SET_SOCKE T_OPT	4	Unable to set socket options
ERR_UNABLE_TO_BIND	5	Unable to bind socket to the given port
ERR_UNABLE_TO_CONNECT	6	Unable to connect to the given server port
ERR_UNABLE_TO_LISTEN	7	Unable to listen on the given port

ERR_UNABLE_TO_RETR_ADD _STRUC	8	Unable to retrieve address structure for given address hint structure
ERR_EAI_ADDRFAMILY	9	The specified network host does not have any network addresses in the address family for the IP version
ERR_EAI_FAMILY	10	The requested address family is not supported at all for the IP version
ERR_EAI_SOCKTYPE	11	The requested socket type is not supported at all for the IP version
ERR_UNABLE_CLOSE	12	Unable to close the socket
ERR_SELECT_TIMEDOUT	13	Select timed out
ERR_INVALID_SOCKET_TYPE	25	Invalid socket type

The error codes retuned by the function *fn_socket()* are converted to corresponding error messages by using the function *fn_getErrorString()*. An example below shows how to use the function *fn_getErrorString()*.

```
char vlErrorMsg[size];
vlError = fn_socket(&vg_latencySocket,SOCKET_TCP,IP_BOTH);

if(vlError != SUCCESS){
     vlErrorMsg  = fn_getErrorString(vlError);
   }
```

3. Bandwidth Measurement:

Constants	Value	Use
BW_MAX_SIZE	32 KB	Max packet size that can be sent for Bandwidth Measurement
BW_MAX_PACKETS	192	Max number of times the packets can be sent for Bandwidth Measurement
BW_DELAY	30	Delay between two Bandwidth Measurement sessions
BW_MAX_RX_SIZE	6 KB	Max Bandwidth Receive size

4. Latency Measurement:

Constants	Value	Use
LATENCY_MAX_SIZE	56	Max packet size that can be sent for latency Measurement
LATENCY_DELAY	5	Delay between two latency Measurement sessions

5. Control Thread:

Constants	Value	Use
BANDWIDTH_INFO	1	Specifies control packet is for bandwidth
IP_SWITCH	4	Specifies control packet is to indicate IP protocol switch
IP_PROTO_SWITCH	1	Constant assigned to flag that indicates switch on receiving an event

6. POSIX threads and Error messages:

Constants	Value	Use
TX_THREAD	0x0002	Specifies transmit thread
RX_THREAD	0x0004	Specifies receive thread
ECHO_THREAD	0x0008	Specifies echo thread
PTHREAD_DEATT	0	Specifies a thread is detached thread
PTHREAD_ATT	1	Specifies a thread is attached thread

Constants	Value	Error message
ERR_PTHREAD_NO_SUPPORT	19	This OS does not support pthreads
ERR_PTHREAD_UNABLE_TO_INIT	20	Unable to initialize pthreads
ERR_PTHREAD_NOT_DETACHABLE	21	Unable to make pthreads detachable
ERR_PTHREAD_NOT_ATTACHABLE	22	Unable to make pthreads attachable
ERR_PTHREAD_UNABLE_TO_CREATE	23	Unable to create pthreads
ERR_PTHREAD_UNABLE_TO_DEST_ATTR	24	Unable to destroy pthread attribute

7. Others:

Constants	Value	Use
ONE_KB	1024	1 KB
ONE_MB	ONE_KB*ONE_KB	1 MB
ONE_GB	ONE_MB*ONE_KB	1 GB
SUCCESS	0	Constant to indicate success and returned after successful completion for middleware application functions
TRUE	1	Constant to indicate true
FALSE	1	Constant to indicate false
SIZE_OF_DATA	2*ONE_KB	Actual data size sent while reading/writing/sending/recving data from file
ERR_TIMEOUT	27	Timeout occurred
ERR_NO_MEM	26	Not enough memory available

Appendix C –Capacity Measurement Log File

Measurement Log File

The sample output of the log file at client/receiver (*netaware_rcv)* while measuring the capacity over IPv6 protocol is as below. Specifying value for -o option with a file name logs the output of each measurement phases. The default file name is *netaware_capacity.output*.

```
pathrate run from web.rnet.missouri.edu to peregrine.rnet.missouri.edu on Fri
May 31 8:16:55 2013
IP protocol : IPv6
       --> Average round-trip time: 0.1ms

--> Minimum acceptable packet pair dispersion: 6 usec
       Train length: 2 ->    2381 Mbps NOT Acceptable measurement
(ignored)
       Train length: 3 ->    2381 Mbps NOT Acceptable measurement
(ignored)
       Train length: 4 ->    2551 Mbps NOT Acceptable measurement
(ignored)
       Train length: 5 ->    2381 Mbps NOT Acceptable measurement
(ignored)
       Train length: 6 ->    2381 Mbps NOT Acceptable measurement
(ignored)
       Train length: 8 ->    1157 Mbps Acceptable measurement
       Train length: 10 ->   1165 Mbps Acceptable measurement
       Train length: 12 ->   909 Mbps Acceptable measurement
       Train length: 16 ->   998 Mbps Acceptable measurement
       Train length: 20 ->   958 Mbps Acceptable measurement
       Train length: 24 ->   978 Mbps Acceptable measurement
       Train length: 28 ->   968 Mbps Acceptable measurement
       Train length: 32 ->   907 Mbps Acceptable measurement
       Train length: 36 ->   976 Mbps Acceptable measurement
       Train length: 40 ->   963 Mbps Acceptable measurement
       Train length: 44 ->   951 Mbps Acceptable measurement
       Train length: 48 ->   961 Mbps Acceptable measurement
       --> Maximum train length: 48 packets


--Preliminary measurements with increasing packet train lengths--
  Train length: 2 ->  2381 Mbps  1984 Mbps  1984 Mbps  1984 Mbps  2381 Mbps
1984 Mbps  627 Mbps
  Train length: 3 ->  2381 Mbps  2164 Mbps  2164 Mbps  2164 Mbps  2381 Mbps
2381 Mbps  2164 Mbps
  Train length: 4 ->  2381 Mbps  2381 Mbps  2381 Mbps  2381 Mbps  1428 Mbps
2232 Mbps  2381 Mbps
```

```
  Train length: 5 ->  2381 Mbps  1536 Mbps  2267 Mbps  2267 Mbps  2267 Mbps
2381 Mbps  2267 Mbps
  Train length: 6 ->  1215 Mbps  2289 Mbps  2381 Mbps  2289 Mbps  2381 Mbps
2289 Mbps  2289 Mbps
  Train length: 7 ->  2304 Mbps  2304 Mbps  2232 Mbps  2304 Mbps  2381 Mbps
2304 Mbps  2381 Mbps
  Train length: 8 ->  1174 Mbps  1190 Mbps  1157 Mbps  1157 Mbps  1190 Mbps
1096 Mbps  1174 Mbps
  Train length: 9 ->  1134 Mbps  1176 Mbps  1147 Mbps  1147 Mbps  1147 Mbps
1147 Mbps  1161 Mbps
  Train length: 10 ->  1050 Mbps  1165 Mbps  1165 Mbps  1165 Mbps  1177 Mbps
1177 Mbps  1177 Mbps
```

--> Capacity Resolution: 56 Mbps

```
-- Phase I: Detect possible capacity modes --

      -> Train length:  2 - Packet size:  600B ->  0% completed
      Measurement-1: 960 Mbps  (5 usec) (ignored)
      Measurement-2: 960 Mbps  (5 usec) (ignored)
      Measurement-3: 800 Mbps  (6 usec) (ignored)
      Measurement-4: 1200 Mbps  (4 usec) (ignored)
      Measurement-5: 960 Mbps  (5 usec) (ignored)
      Measurement-6: 960 Mbps  (5 usec) (ignored)
      Measurement-7: 960 Mbps  (5 usec) (ignored)
      Measurement-8: 960 Mbps  (5 usec) (ignored)
      Measurement-9: 960 Mbps  (5 usec) (ignored)
      Measurement-10: 960 Mbps  (5 usec) (ignored)
      Measurement-11: 960 Mbps  (5 usec) (ignored)
      Measurement-12: 1200 Mbps  (4 usec) (ignored)
      Measurement-13: 960 Mbps  (5 usec) (ignored)
      Measurement-14: 1200 Mbps  (4 usec) (ignored)
      Measurement-15: 960 Mbps  (5 usec) (ignored)
      Measurement-16: 960 Mbps  (5 usec) (ignored)
      Measurement-17: 960 Mbps  (5 usec) (ignored)
      Measurement-18: 800 Mbps  (6 usec) (ignored)
      Measurement-19: 1200 Mbps  (4 usec) (ignored)
      Measurement-20: 1200 Mbps  (4 usec) (ignored)
      Measurement-21: 960 Mbps  (5 usec) (ignored)
      Measurement-22: 960 Mbps  (5 usec) (ignored)
      Measurement-23: 960 Mbps  (5 usec) (ignored)
      Measurement-24: 1200 Mbps  (4 usec) (ignored)
      Measurement-25: 1200 Mbps  (4 usec) (ignored)

      Too many ignored measurements..
      Adjust train length: 3 packets
      Adjust packet size: 898 bytes

      -> Train length:  3 - Packet size:  898B ->  2% completed
      Measurement-1: 306 Mbps  (47 usec)
      Measurement-2: 334 Mbps  (43 usec)
      Measurement-3: 327 Mbps  (44 usec)
      Measurement-4: 312 Mbps  (46 usec)
      Measurement-5: 327 Mbps  (44 usec)
      Measurement-6: 1105 Mbps  (13 usec)
      Measurement-7: 463 Mbps  (31 usec)
      Measurement-8: 334 Mbps  (43 usec)
      Measurement-9: 306 Mbps  (47 usec)
      Measurement-10: 334 Mbps  (43 usec)
      Measurement-11: 312 Mbps  (46 usec)
```

```
Measurement-12: 306 Mbps  (47 usec)
Measurement-13: 299 Mbps  (48 usec)
Measurement-14: 299 Mbps  (48 usec)
Measurement-15: 299 Mbps  (48 usec)
Measurement-16: 299 Mbps  (48 usec)
Measurement-17: 342 Mbps  (42 usec)
Measurement-18: 299 Mbps  (48 usec)
Measurement-19: 299 Mbps  (48 usec)
Measurement-20: 299 Mbps  (48 usec)
Measurement-21: 299 Mbps  (48 usec)
Measurement-22: 299 Mbps  (48 usec)
Measurement-23: 299 Mbps  (48 usec)
Measurement-24: 306 Mbps  (47 usec)
Measurement-25: 299 Mbps  (48 usec)
-> Train length:  3 - Packet size:  921B ->  5% completed
Measurement-1: 1134 Mbps  (13 usec)
Measurement-2: 343 Mbps  (43 usec)
Measurement-3: 307 Mbps  (48 usec)
Measurement-4: 307 Mbps  (48 usec)
Measurement-5: 307 Mbps  (48 usec)
Measurement-6: 307 Mbps  (48 usec)
Measurement-7: 301 Mbps  (49 usec)
Measurement-8: 301 Mbps  (49 usec)
Measurement-9: 307 Mbps  (48 usec)
Measurement-10: 307 Mbps  (48 usec)
Measurement-11: 314 Mbps  (47 usec)
Measurement-12: 343 Mbps  (43 usec)
Measurement-13: 307 Mbps  (48 usec)
Measurement-14: 307 Mbps  (48 usec)
Measurement-15: 307 Mbps  (48 usec)
Measurement-16: 307 Mbps  (48 usec)
Measurement-17: 314 Mbps  (47 usec)
Measurement-18: 307 Mbps  (48 usec)
Measurement-19: 295 Mbps  (50 usec)
Measurement-20: 307 Mbps  (48 usec)
Measurement-21: 314 Mbps  (47 usec)
Measurement-22: 343 Mbps  (43 usec)
Measurement-23: 307 Mbps  (48 usec)
Measurement-24: 307 Mbps  (48 usec)
Measurement-25: 314 Mbps  (47 usec)
-> Train length:  3 - Packet size:  944B ->  7% completed
Measurement-1: 1162 Mbps  (13 usec)
Measurement-2: 315 Mbps  (48 usec)
Measurement-3: 315 Mbps  (48 usec)
Measurement-4: 315 Mbps  (48 usec)
Measurement-5: 315 Mbps  (48 usec)
Measurement-6: 315 Mbps  (48 usec)
Measurement-7: 360 Mbps  (42 usec)
Measurement-8: 315 Mbps  (48 usec)
Measurement-9: 315 Mbps  (48 usec)
Measurement-10: 315 Mbps  (48 usec)
Measurement-11: 321 Mbps  (47 usec)
Measurement-12: 315 Mbps  (48 usec)
Measurement-13: 315 Mbps  (48 usec)
Measurement-14: 308 Mbps  (49 usec)
Measurement-15: 159 Mbps  (95 usec)
Measurement-16: 302 Mbps  (50 usec)
Measurement-17: 378 Mbps  (40 usec)
Measurement-18: 296 Mbps  (51 usec)
Measurement-19: 315 Mbps  (48 usec)
Measurement-20: 182 Mbps  (83 usec)
```

```
Measurement-21: 360 Mbps  (42 usec)
Measurement-22: 368 Mbps  (41 usec)
Measurement-23: 755 Mbps  (20 usec)
Measurement-24: 336 Mbps  (45 usec)
Measurement-25: 1678 Mbps  (9 usec) (ignored)
-> Train length:  3 - Packet size:  967B -> 10% completed
Measurement-1: 1719 Mbps  (9 usec) (ignored)
Measurement-2: 329 Mbps  (47 usec)
Measurement-3: 360 Mbps  (43 usec)
Measurement-4: 360 Mbps  (43 usec)
Measurement-5: 368 Mbps  (42 usec)
Measurement-6: 344 Mbps  (45 usec)
Measurement-7: 352 Mbps  (44 usec)
Measurement-8: 344 Mbps  (45 usec)
Measurement-9: 336 Mbps  (46 usec)
Measurement-10: 336 Mbps  (46 usec)
Measurement-11: 329 Mbps  (47 usec)
Measurement-12: 397 Mbps  (39 usec)
Measurement-13: 316 Mbps  (49 usec)
Measurement-14: 322 Mbps  (48 usec)
Measurement-15: 316 Mbps  (49 usec)
Measurement-16: 322 Mbps  (48 usec)
Measurement-17: 316 Mbps  (49 usec)
Measurement-18: 322 Mbps  (48 usec)
Measurement-19: 322 Mbps  (48 usec)
Measurement-20: 1407 Mbps  (11 usec) (ignored)
Measurement-21: 1289 Mbps  (12 usec) (ignored)
Measurement-22: 1289 Mbps  (12 usec) (ignored)
Measurement-23: 1190 Mbps  (13 usec)
Measurement-24: 1289 Mbps  (12 usec) (ignored)
Measurement-25: 1719 Mbps  (9 usec) (ignored)
-> Train length:  3 - Packet size:  990B -> 12% completed
Measurement-1: 1760 Mbps  (9 usec) (ignored)
Measurement-2: 1320 Mbps  (12 usec) (ignored)
Measurement-3: 1320 Mbps  (12 usec) (ignored)
Measurement-4: 1320 Mbps  (12 usec) (ignored)
Measurement-5: 1320 Mbps  (12 usec) (ignored)
Measurement-6: 1218 Mbps  (13 usec)
Measurement-7: 1320 Mbps  (12 usec) (ignored)
Measurement-8: 1218 Mbps  (13 usec)
Measurement-9: 1320 Mbps  (12 usec) (ignored)
Measurement-10: 323 Mbps  (49 usec)
Measurement-11: 1320 Mbps  (12 usec) (ignored)
Measurement-12: 1320 Mbps  (12 usec) (ignored)
Measurement-13: 1320 Mbps  (12 usec) (ignored)
Measurement-14: 1320 Mbps  (12 usec) (ignored)
Measurement-15: 1320 Mbps  (12 usec) (ignored)
Measurement-16: 1320 Mbps  (12 usec) (ignored)
Measurement-17: 1440 Mbps  (11 usec) (ignored)
Measurement-18: 1218 Mbps  (13 usec)
Measurement-19: 1320 Mbps  (12 usec) (ignored)
Measurement-20: 1320 Mbps  (12 usec) (ignored)
Measurement-21: 1320 Mbps  (12 usec) (ignored)
Measurement-22: 1320 Mbps  (12 usec) (ignored)
Measurement-23: 1320 Mbps  (12 usec) (ignored)
Measurement-24: 317 Mbps  (50 usec)
Measurement-25: 323 Mbps  (49 usec)

Too many ignored measurements..
Adjust train length: 4 packets
Adjust packet size: 1288 bytes
```

```
-> Train length:  4 - Packet size: 1288B -> 15% completed
Measurement-1: 2208 Mbps  (14 usec) (ignored)
Measurement-2: 2208 Mbps  (14 usec) (ignored)
Measurement-3: 2208 Mbps  (14 usec) (ignored)
Measurement-4: 2208 Mbps  (14 usec) (ignored)
Measurement-5: 2378 Mbps  (13 usec) (ignored)
Measurement-6: 2378 Mbps  (13 usec) (ignored)
Measurement-7: 2378 Mbps  (13 usec) (ignored)
Measurement-8: 2378 Mbps  (13 usec) (ignored)
Measurement-9: 2378 Mbps  (13 usec) (ignored)
Measurement-10: 2208 Mbps  (14 usec) (ignored)
Measurement-11: 2208 Mbps  (14 usec) (ignored)
Measurement-12: 1818 Mbps  (17 usec) (ignored)
Measurement-13: 2208 Mbps  (14 usec) (ignored)
Measurement-14: 2208 Mbps  (14 usec) (ignored)
Measurement-15: 2208 Mbps  (14 usec) (ignored)
Measurement-16: 2208 Mbps  (14 usec) (ignored)
Measurement-17: 2208 Mbps  (14 usec) (ignored)
Measurement-18: 859 Mbps  (36 usec)
Measurement-19: 2378 Mbps  (13 usec) (ignored)
Measurement-20: 2061 Mbps  (15 usec) (ignored)
Measurement-21: 2208 Mbps  (14 usec) (ignored)
Measurement-22: 2208 Mbps  (14 usec) (ignored)
Measurement-23: 937 Mbps  (33 usec)
Measurement-24: 2208 Mbps  (14 usec) (ignored)
Measurement-25: 883 Mbps  (35 usec)

Too many ignored measurements..
Adjust train length: 5 packets
Adjust packet size: 1488 bytes

-> Train length:  5 - Packet size: 1488B -> 17% completed
Measurement-1: 2267 Mbps  (21 usec) (ignored)
Measurement-2: 2267 Mbps  (21 usec) (ignored)
Measurement-3: 2506 Mbps  (19 usec) (ignored)
Measurement-4: 2381 Mbps  (20 usec) (ignored)
Measurement-5: 2381 Mbps  (20 usec) (ignored)
Measurement-6: 2381 Mbps  (20 usec) (ignored)
Measurement-7: 2381 Mbps  (20 usec) (ignored)
Measurement-8: 2381 Mbps  (20 usec) (ignored)
Measurement-9: 2381 Mbps  (20 usec) (ignored)
Measurement-10: 2506 Mbps  (19 usec) (ignored)
Measurement-11: 2381 Mbps  (20 usec) (ignored)
Measurement-12: 2267 Mbps  (21 usec) (ignored)
Measurement-13: 2381 Mbps  (20 usec) (ignored)
Measurement-14: 2381 Mbps  (20 usec) (ignored)
Measurement-15: 2381 Mbps  (20 usec) (ignored)
Measurement-16: 2381 Mbps  (20 usec) (ignored)
Measurement-17: 2381 Mbps  (20 usec) (ignored)
Measurement-18: 2381 Mbps  (20 usec) (ignored)
Measurement-19: 2381 Mbps  (20 usec) (ignored)
Measurement-20: 2506 Mbps  (19 usec) (ignored)
Measurement-21: 2381 Mbps  (20 usec) (ignored)
Measurement-22: 2381 Mbps  (20 usec) (ignored)
Measurement-23: 2381 Mbps  (20 usec) (ignored)
Measurement-24: 2506 Mbps  (19 usec) (ignored)
Measurement-25: 2506 Mbps  (19 usec) (ignored)

Too many ignored measurements..
Adjust train length: 6 packets
```

```
Adjust packet size: 1488 bytes

-> Train length:  6 - Packet size: 1488B -> 20% completed
Measurement-1: 2381 Mbps  (25 usec) (ignored)
Measurement-2: 2381 Mbps  (25 usec) (ignored)
Measurement-3: 2381 Mbps  (25 usec) (ignored)
Measurement-4: 2381 Mbps  (25 usec) (ignored)
Measurement-5: 2480 Mbps  (24 usec) (ignored)
Measurement-6: 2381 Mbps  (25 usec) (ignored)
Measurement-7: 1266 Mbps  (47 usec)
Measurement-8: 1266 Mbps  (47 usec)
Measurement-9: 1294 Mbps  (46 usec)
Measurement-10: 2381 Mbps  (25 usec) (ignored)
Measurement-11: 2289 Mbps  (26 usec) (ignored)
Measurement-12: 1323 Mbps  (45 usec)
Measurement-13: 2381 Mbps  (25 usec) (ignored)
Measurement-14: 2381 Mbps  (25 usec) (ignored)
Measurement-15: 1266 Mbps  (47 usec)
Measurement-16: 1294 Mbps  (46 usec)
Measurement-17: 2381 Mbps  (25 usec) (ignored)
Measurement-18: 2381 Mbps  (25 usec) (ignored)
Measurement-19: 2381 Mbps  (25 usec) (ignored)
Measurement-20: 2381 Mbps  (25 usec) (ignored)
Measurement-21: 1266 Mbps  (47 usec)
Measurement-22: 1123 Mbps  (53 usec)
Measurement-23: 1294 Mbps  (46 usec)
Measurement-24: 2381 Mbps  (25 usec) (ignored)
Measurement-25: 2289 Mbps  (26 usec) (ignored)

Too many ignored measurements..
Adjust train length: 7 packets
Adjust packet size: 1488 bytes

-> Train length:  7 - Packet size: 1488B -> 22% completed
Measurement-1: 2381 Mbps  (30 usec) (ignored)
Measurement-2: 2232 Mbps  (32 usec) (ignored)
Measurement-3: 2304 Mbps  (31 usec) (ignored)
Measurement-4: 2304 Mbps  (31 usec) (ignored)
Measurement-5: 2381 Mbps  (30 usec) (ignored)
Measurement-6: 2381 Mbps  (30 usec) (ignored)
Measurement-7: 2381 Mbps  (30 usec) (ignored)
Measurement-8: 1400 Mbps  (51 usec)
Measurement-9: 2164 Mbps  (33 usec) (ignored)
Measurement-10: 1050 Mbps  (68 usec)
Measurement-11: 965 Mbps  (74 usec)
Measurement-12: 1488 Mbps  (48 usec)
Measurement-13: 1488 Mbps  (48 usec)
Measurement-14: 687 Mbps  (104 usec)
Measurement-15: 661 Mbps  (108 usec)
Measurement-16: 2381 Mbps  (30 usec) (ignored)
Measurement-17: 2381 Mbps  (30 usec) (ignored)
Measurement-18: 2381 Mbps  (30 usec) (ignored)
Measurement-19: 2381 Mbps  (30 usec) (ignored)
Measurement-20: 2381 Mbps  (30 usec) (ignored)
Measurement-21: 1020 Mbps  (70 usec)
Measurement-22: 2381 Mbps  (30 usec) (ignored)
Measurement-23: 2381 Mbps  (30 usec) (ignored)
Measurement-24: 978 Mbps  (73 usec)
Measurement-25: 2381 Mbps  (30 usec) (ignored)

Too many ignored measurements..
```

```
Adjust train length: 8 packets
Adjust packet size: 1488 bytes

-> Train length:  8 - Packet size: 1488B -> 25% completed
Measurement-1: 1208 Mbps  (69 usec)
Measurement-2: 1244 Mbps  (67 usec)
Measurement-3: 1225 Mbps  (68 usec)
Measurement-4: 1225 Mbps  (68 usec)
Measurement-5: 1208 Mbps  (69 usec)
Measurement-6: 1225 Mbps  (68 usec)
Measurement-7: 1190 Mbps  (70 usec)
Measurement-8: 1174 Mbps  (71 usec)
Measurement-9: 1157 Mbps  (72 usec)
Measurement-10: 1174 Mbps  (71 usec)
Measurement-11: 1082 Mbps  (77 usec)
Measurement-12: 1126 Mbps  (74 usec)
Measurement-13: 1282 Mbps  (65 usec)
Measurement-14: 1190 Mbps  (70 usec)
Measurement-15: 1174 Mbps  (71 usec)
Measurement-16: 1174 Mbps  (71 usec)
Measurement-17: 1225 Mbps  (68 usec)
Measurement-18: 1190 Mbps  (70 usec)
Measurement-19: 1190 Mbps  (70 usec)
Measurement-20: 1174 Mbps  (71 usec)
Measurement-21: 1225 Mbps  (68 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 1190 Mbps  (70 usec)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1190 Mbps  (70 usec)
-> Train length:  8 - Packet size: 1488B -> 27% completed
Measurement-1: 1174 Mbps  (71 usec)
Measurement-2: 1174 Mbps  (71 usec)
Measurement-3: 1174 Mbps  (71 usec)
Measurement-4: 1157 Mbps  (72 usec)
Measurement-5: 1174 Mbps  (71 usec)
Measurement-6: 1068 Mbps  (78 usec)
Measurement-7: 2315 Mbps  (36 usec) (ignored)
Measurement-8: 1174 Mbps  (71 usec)
Measurement-9: 1096 Mbps  (76 usec)
Measurement-10: 936 Mbps  (89 usec)
Measurement-11: 1096 Mbps  (76 usec)
Measurement-12: 1141 Mbps  (73 usec)
Measurement-13: 1157 Mbps  (72 usec)
Measurement-14: 1157 Mbps  (72 usec)
Measurement-15: 1323 Mbps  (63 usec)
Measurement-16: 1225 Mbps  (68 usec)
Measurement-17: 1029 Mbps  (81 usec)
Measurement-18: 1190 Mbps  (70 usec)
Measurement-19: 1208 Mbps  (69 usec)
Measurement-20: 1174 Mbps  (71 usec)
Measurement-21: 1174 Mbps  (71 usec)
Measurement-22: 1208 Mbps  (69 usec)
Measurement-23: 1096 Mbps  (76 usec)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1190 Mbps  (70 usec)
-> Train length:  8 - Packet size: 1488B -> 30% completed
Measurement-1: 1068 Mbps  (78 usec)
Measurement-2: 1016 Mbps  (82 usec)
Measurement-3: 631 Mbps  (132 usec)
Measurement-4: 1208 Mbps  (69 usec)
Measurement-5: 1263 Mbps  (66 usec)
```

```
Measurement-6: 1190 Mbps  (70 usec)
Measurement-7: 1208 Mbps  (69 usec)
Measurement-8: 2451 Mbps  (34 usec) (ignored)
Measurement-9: 1225 Mbps  (68 usec)
Measurement-10: 1126 Mbps  (74 usec)
Measurement-11: 1225 Mbps  (68 usec)
Measurement-12: 958 Mbps  (87 usec)
Measurement-13: 1282 Mbps  (65 usec)
Measurement-14: 1225 Mbps  (68 usec)
Measurement-15: 712 Mbps  (117 usec)
Measurement-16: 1208 Mbps  (69 usec)
Measurement-17: 969 Mbps  (86 usec)
Measurement-18: 1208 Mbps  (69 usec)
Measurement-19: 969 Mbps  (86 usec)
Measurement-20: 1244 Mbps  (67 usec)
Measurement-21: 1111 Mbps  (75 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 1282 Mbps  (65 usec)
Measurement-24: 1174 Mbps  (71 usec)
Measurement-25: 1244 Mbps  (67 usec)
-> Train length:  8 - Packet size: 1488B -> 32% completed
Measurement-1: 1111 Mbps  (75 usec)
Measurement-2: 1126 Mbps  (74 usec)
Measurement-3: 1082 Mbps  (77 usec)
Measurement-4: 1082 Mbps  (77 usec)
Measurement-5: 1263 Mbps  (66 usec)
Measurement-6: 1225 Mbps  (68 usec)
Measurement-7: 1225 Mbps  (68 usec)
Measurement-8: 1111 Mbps  (75 usec)
Measurement-9: 1096 Mbps  (76 usec)
Measurement-10: 1157 Mbps  (72 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1157 Mbps  (72 usec)
Measurement-13: 1174 Mbps  (71 usec)
Measurement-14: 2315 Mbps  (36 usec) (ignored)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1141 Mbps  (73 usec)
Measurement-17: 1126 Mbps  (74 usec)
Measurement-18: 1174 Mbps  (71 usec)
Measurement-19: 1174 Mbps  (71 usec)
Measurement-20: 1157 Mbps  (72 usec)
Measurement-21: 1096 Mbps  (76 usec)
Measurement-22: 1141 Mbps  (73 usec)
Measurement-23: 1157 Mbps  (72 usec)
Measurement-24: 1157 Mbps  (72 usec)
Measurement-25: 1208 Mbps  (69 usec)
-> Train length:  8 - Packet size: 1488B -> 35% completed
Measurement-1: 1157 Mbps  (72 usec)
Measurement-2: 1174 Mbps  (71 usec)
Measurement-3: 1190 Mbps  (70 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1174 Mbps  (71 usec)
Measurement-6: 1157 Mbps  (72 usec)
Measurement-7: 1157 Mbps  (72 usec)
Measurement-8: 1174 Mbps  (71 usec)
Measurement-9: 1208 Mbps  (69 usec)
Measurement-10: 1157 Mbps  (72 usec)
Measurement-11: 1174 Mbps  (71 usec)
Measurement-12: 980 Mbps  (85 usec)
Measurement-13: 1462 Mbps  (57 usec)
Measurement-14: 1208 Mbps  (69 usec)
```

```
Measurement-15: 1190 Mbps  (70 usec)
Measurement-16: 1190 Mbps  (70 usec)
Measurement-17: 1190 Mbps  (70 usec)
Measurement-18: 1174 Mbps  (71 usec)
Measurement-19: 1082 Mbps  (77 usec)
Measurement-20: 1190 Mbps  (70 usec)
Measurement-21: 1190 Mbps  (70 usec)
Measurement-22: 1225 Mbps  (68 usec)
Measurement-23: 2381 Mbps  (35 usec) (ignored)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1157 Mbps  (72 usec)
-> Train length:  8 - Packet size: 1488B -> 37% completed
Measurement-1: 1263 Mbps  (66 usec)
Measurement-2: 1157 Mbps  (72 usec)
Measurement-3: 1208 Mbps  (69 usec)
Measurement-4: 1208 Mbps  (69 usec)
Measurement-5: 1190 Mbps  (70 usec)
Measurement-6: 764 Mbps  (109 usec)
Measurement-7: 1225 Mbps  (68 usec)
Measurement-8: 1190 Mbps  (70 usec)
Measurement-9: 1190 Mbps  (70 usec)
Measurement-10: 2252 Mbps  (37 usec) (ignored)
Measurement-11: 1225 Mbps  (68 usec)
Measurement-12: 2193 Mbps  (38 usec) (ignored)
Measurement-13: 1263 Mbps  (66 usec)
Measurement-14: 1190 Mbps  (70 usec)
Measurement-15: 969 Mbps  (86 usec)
Measurement-16: 2137 Mbps  (39 usec) (ignored)
Measurement-17: 1208 Mbps  (69 usec)
Measurement-18: 1225 Mbps  (68 usec)
Measurement-19: 1190 Mbps  (70 usec)
Measurement-20: 1190 Mbps  (70 usec)
Measurement-21: 1190 Mbps  (70 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 1263 Mbps  (66 usec)
Measurement-24: 1225 Mbps  (68 usec)
Measurement-25: 992 Mbps  (84 usec)
-> Train length:  8 - Packet size: 1488B -> 40% completed
Measurement-1: 1111 Mbps  (75 usec)
Measurement-2: 1208 Mbps  (69 usec)
Measurement-3: 1111 Mbps  (75 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1190 Mbps  (70 usec)
Measurement-6: 1174 Mbps  (71 usec)
Measurement-7: 1141 Mbps  (73 usec)
Measurement-8: 1141 Mbps  (73 usec)
Measurement-9: 1082 Mbps  (77 usec)
Measurement-10: 1208 Mbps  (69 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1190 Mbps  (70 usec)
Measurement-13: 1174 Mbps  (71 usec)
Measurement-14: 1174 Mbps  (71 usec)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1157 Mbps  (72 usec)
Measurement-17: 1141 Mbps  (73 usec)
Measurement-18: 1174 Mbps  (71 usec)
Measurement-19: 1157 Mbps  (72 usec)
Measurement-20: 1157 Mbps  (72 usec)
Measurement-21: 1157 Mbps  (72 usec)
Measurement-22: 1157 Mbps  (72 usec)
Measurement-23: 1174 Mbps  (71 usec)
```

```
Measurement-24: 1157 Mbps  (72 usec)
Measurement-25: 1174 Mbps  (71 usec)
-> Train length:  8 - Packet size: 1488B -> 42% completed
Measurement-1: 1157 Mbps  (72 usec)
Measurement-2: 1157 Mbps  (72 usec)
Measurement-3: 1174 Mbps  (71 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1174 Mbps  (71 usec)
Measurement-7: 2381 Mbps  (35 usec) (ignored)
Measurement-8: 1174 Mbps  (71 usec)
Measurement-9: 1141 Mbps  (73 usec)
Measurement-10: 1190 Mbps  (70 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1174 Mbps  (71 usec)
Measurement-14: 1157 Mbps  (72 usec)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1157 Mbps  (72 usec)
Measurement-17: 1244 Mbps  (67 usec)
Measurement-18: 1190 Mbps  (70 usec)
Measurement-19: 1174 Mbps  (71 usec)
Measurement-20: 1190 Mbps  (70 usec)
Measurement-21: 1157 Mbps  (72 usec)
Measurement-22: 1157 Mbps  (72 usec)
Measurement-23: 1157 Mbps  (72 usec)
Measurement-24: 1208 Mbps  (69 usec)
Measurement-25: 1174 Mbps  (71 usec)
-> Train length:  8 - Packet size: 1488B -> 45% completed
Measurement-1: 1157 Mbps  (72 usec)
Measurement-2: 1263 Mbps  (66 usec)
Measurement-3: 1126 Mbps  (74 usec)
Measurement-4: 1515 Mbps  (55 usec)
Measurement-5: 1111 Mbps  (75 usec)
Measurement-6: 1141 Mbps  (73 usec)
Measurement-7: 1244 Mbps  (67 usec)
Measurement-8: 1174 Mbps  (71 usec)
Measurement-9: 1208 Mbps  (69 usec)
Measurement-10: 1208 Mbps  (69 usec)
Measurement-11: 1190 Mbps  (70 usec)
Measurement-12: 1208 Mbps  (69 usec)
Measurement-13: 1208 Mbps  (69 usec)
Measurement-14: 1208 Mbps  (69 usec)
Measurement-15: 1208 Mbps  (69 usec)
Measurement-16: 1263 Mbps  (66 usec)
Measurement-17: 1225 Mbps  (68 usec)
Measurement-18: 980 Mbps  (85 usec)
Measurement-19: 1190 Mbps  (70 usec)
Measurement-20: 1004 Mbps  (83 usec)
Measurement-21: 1111 Mbps  (75 usec)
Measurement-22: 1111 Mbps  (75 usec)
Measurement-23: 1111 Mbps  (75 usec)
Measurement-24: 1126 Mbps  (74 usec)
Measurement-25: 1225 Mbps  (68 usec)
-> Train length:  8 - Packet size: 1488B -> 47% completed
Measurement-1: 1190 Mbps  (70 usec)
Measurement-2: 1190 Mbps  (70 usec)
Measurement-3: 1190 Mbps  (70 usec)
Measurement-4: 1141 Mbps  (73 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1174 Mbps  (71 usec)
```

```
Measurement-7: 1174 Mbps  (71 usec)
Measurement-8: 1437 Mbps  (58 usec)
Measurement-9: 1208 Mbps  (69 usec)
Measurement-10: 1174 Mbps  (71 usec)
Measurement-11: 1190 Mbps  (70 usec)
Measurement-12: 1190 Mbps  (70 usec)
Measurement-13: 1190 Mbps  (70 usec)
Measurement-14: 1208 Mbps  (69 usec)
Measurement-15: 1190 Mbps  (70 usec)
Measurement-16: 1174 Mbps  (71 usec)
Measurement-17: 1174 Mbps  (71 usec)
Measurement-18: 1208 Mbps  (69 usec)
Measurement-19: 1174 Mbps  (71 usec)
Measurement-20: 1157 Mbps  (72 usec)
Measurement-21: 1141 Mbps  (73 usec)
Measurement-22: 1141 Mbps  (73 usec)
Measurement-23: 1174 Mbps  (71 usec)
Measurement-24: 1174 Mbps  (71 usec)
Measurement-25: 1157 Mbps  (72 usec)
-> Train length:  8 - Packet size: 1488B -> 50% completed
Measurement-1: 1157 Mbps  (72 usec)
Measurement-2: 1157 Mbps  (72 usec)
Measurement-3: 1174 Mbps  (71 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1157 Mbps  (72 usec)
Measurement-7: 1174 Mbps  (71 usec)
Measurement-8: 1157 Mbps  (72 usec)
Measurement-9: 1157 Mbps  (72 usec)
Measurement-10: 1141 Mbps  (73 usec)
Measurement-11: 1174 Mbps  (71 usec)
Measurement-12: 1157 Mbps  (72 usec)
Measurement-13: 1190 Mbps  (70 usec)
Measurement-14: 1174 Mbps  (71 usec)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1157 Mbps  (72 usec)
Measurement-17: 1174 Mbps  (71 usec)
Measurement-18: 1157 Mbps  (72 usec)
Measurement-19: 1174 Mbps  (71 usec)
Measurement-20: 1157 Mbps  (72 usec)
Measurement-21: 1157 Mbps  (72 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 1208 Mbps  (69 usec)
Measurement-24: 1157 Mbps  (72 usec)
Measurement-25: 1190 Mbps  (70 usec)
-> Train length:  8 - Packet size: 1488B -> 52% completed
Measurement-1: 1190 Mbps  (70 usec)
Measurement-2: 2032 Mbps  (41 usec) (ignored)
Measurement-3: 1244 Mbps  (67 usec)
Measurement-4: 1141 Mbps  (73 usec)
Measurement-5: 1225 Mbps  (68 usec)
Measurement-6: 1055 Mbps  (79 usec)
Measurement-7: 1323 Mbps  (63 usec)
Measurement-8: 980 Mbps  (85 usec)
Measurement-9: 1208 Mbps  (69 usec)
Measurement-10: 2381 Mbps  (35 usec) (ignored)
Measurement-11: 1244 Mbps  (67 usec)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1263 Mbps  (66 usec)
Measurement-14: 1208 Mbps  (69 usec)
Measurement-15: 1208 Mbps  (69 usec)
```

```
Measurement-16: 1225 Mbps  (68 usec)
Measurement-17: 1111 Mbps  (75 usec)
Measurement-18: 1157 Mbps  (72 usec)
Measurement-19: 1126 Mbps  (74 usec)
Measurement-20: 969 Mbps  (86 usec)
Measurement-21: 1208 Mbps  (69 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 1174 Mbps  (71 usec)
Measurement-24: 1208 Mbps  (69 usec)
Measurement-25: 1157 Mbps  (72 usec)
-> Train length:  8 - Packet size: 1488B -> 55% completed
Measurement-1: 980 Mbps  (85 usec)
Measurement-2: 1174 Mbps  (71 usec)
Measurement-3: 1190 Mbps  (70 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1096 Mbps  (76 usec)
Measurement-7: 947 Mbps  (88 usec)
Measurement-8: 1157 Mbps  (72 usec)
Measurement-9: 1157 Mbps  (72 usec)
Measurement-10: 1157 Mbps  (72 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1190 Mbps  (70 usec)
Measurement-14: 1157 Mbps  (72 usec)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1141 Mbps  (73 usec)
Measurement-17: 1174 Mbps  (71 usec)
Measurement-18: 1157 Mbps  (72 usec)
Measurement-19: 1157 Mbps  (72 usec)
Measurement-20: 1208 Mbps  (69 usec)
Measurement-21: 1174 Mbps  (71 usec)
Measurement-22: 1157 Mbps  (72 usec)
Measurement-23: 1174 Mbps  (71 usec)
Measurement-24: 1174 Mbps  (71 usec)
Measurement-25: 1174 Mbps  (71 usec)
-> Train length:  8 - Packet size: 1488B -> 57% completed
Measurement-1: 1157 Mbps  (72 usec)
Measurement-2: 1157 Mbps  (72 usec)
Measurement-3: 1174 Mbps  (71 usec)
Measurement-4: 1157 Mbps  (72 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1141 Mbps  (73 usec)
Measurement-7: 1225 Mbps  (68 usec)
Measurement-8: 1190 Mbps  (70 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1157 Mbps  (72 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1157 Mbps  (72 usec)
Measurement-13: 1174 Mbps  (71 usec)
Measurement-14: 1174 Mbps  (71 usec)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1141 Mbps  (73 usec)
Measurement-17: 1244 Mbps  (67 usec)
Measurement-18: 1174 Mbps  (71 usec)
Measurement-19: 1141 Mbps  (73 usec)
Measurement-20: 1174 Mbps  (71 usec)
Measurement-21: 1174 Mbps  (71 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 1225 Mbps  (68 usec)
Measurement-24: 1244 Mbps  (67 usec)
```

```
Measurement-25: 2315 Mbps  (36 usec) (ignored)
-> Train length:  8 - Packet size: 1488B -> 60% completed
Measurement-1: 1389 Mbps  (60 usec)
Measurement-2: 1208 Mbps  (69 usec)
Measurement-3: 1488 Mbps  (56 usec)
Measurement-4: 1263 Mbps  (66 usec)
Measurement-5: 1282 Mbps  (65 usec)
Measurement-6: 1190 Mbps  (70 usec)
Measurement-7: 1244 Mbps  (67 usec)
Measurement-8: 1225 Mbps  (68 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1174 Mbps  (71 usec)
Measurement-11: 947 Mbps  (88 usec)
Measurement-12: 1208 Mbps  (69 usec)
Measurement-13: 1208 Mbps  (69 usec)
Measurement-14: 1263 Mbps  (66 usec)
Measurement-15: 1029 Mbps  (81 usec)
Measurement-16: 1190 Mbps  (70 usec)
Measurement-17: 1190 Mbps  (70 usec)
Measurement-18: 1225 Mbps  (68 usec)
Measurement-19: 1543 Mbps  (54 usec)
Measurement-20: 1126 Mbps  (74 usec)
Measurement-21: 1208 Mbps  (69 usec)
Measurement-22: 1208 Mbps  (69 usec)
Measurement-23: 1244 Mbps  (67 usec)
Measurement-24: 1174 Mbps  (71 usec)
Measurement-25: 1190 Mbps  (70 usec)
-> Train length:  8 - Packet size: 1488B -> 62% completed
Measurement-1: 1190 Mbps  (70 usec)
Measurement-2: 1141 Mbps  (73 usec)
Measurement-3: 1190 Mbps  (70 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1174 Mbps  (71 usec)
Measurement-7: 1174 Mbps  (71 usec)
Measurement-8: 1190 Mbps  (70 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1190 Mbps  (70 usec)
Measurement-11: 1174 Mbps  (71 usec)
Measurement-12: 1190 Mbps  (70 usec)
Measurement-13: 1174 Mbps  (71 usec)
Measurement-14: 1157 Mbps  (72 usec)
Measurement-15: 1174 Mbps  (71 usec)
Measurement-16: 1157 Mbps  (72 usec)
Measurement-17: 1190 Mbps  (70 usec)
Measurement-18: 1174 Mbps  (71 usec)
Measurement-19: 1157 Mbps  (72 usec)
Measurement-20: 1157 Mbps  (72 usec)
Measurement-21: 1174 Mbps  (71 usec)
Measurement-22: 1157 Mbps  (72 usec)
Measurement-23: 1174 Mbps  (71 usec)
Measurement-24: 2315 Mbps  (36 usec) (ignored)
Measurement-25: 1208 Mbps  (69 usec)
-> Train length:  8 - Packet size: 1488B -> 65% completed
Measurement-1: 1208 Mbps  (69 usec)
Measurement-2: 1157 Mbps  (72 usec)
Measurement-3: 1190 Mbps  (70 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1190 Mbps  (70 usec)
Measurement-6: 1174 Mbps  (71 usec)
Measurement-7: 1157 Mbps  (72 usec)
```

```
Measurement-8: 1190 Mbps  (70 usec)
Measurement-9: 1157 Mbps  (72 usec)
Measurement-10: 1190 Mbps  (70 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1244 Mbps  (67 usec)
Measurement-13: 1190 Mbps  (70 usec)
Measurement-14: 1174 Mbps  (71 usec)
Measurement-15: 1190 Mbps  (70 usec)
Measurement-16: 1157 Mbps  (72 usec)
Measurement-17: 1174 Mbps  (71 usec)
Measurement-18: 1412 Mbps  (59 usec)
Measurement-19: 1174 Mbps  (71 usec)
Measurement-20: 622 Mbps  (134 usec)
Measurement-21: 1225 Mbps  (68 usec)
Measurement-22: 1263 Mbps  (66 usec)
Measurement-23: 1111 Mbps  (75 usec)
Measurement-24: 1225 Mbps  (68 usec)
Measurement-25: 1323 Mbps  (63 usec)
-> Train length:  8 - Packet size: 1488B -> 67% completed
Measurement-1: 1174 Mbps  (71 usec)
Measurement-2: 1208 Mbps  (69 usec)
Measurement-3: 2315 Mbps  (36 usec) (ignored)
Measurement-4: 1225 Mbps  (68 usec)
Measurement-5: 1141 Mbps  (73 usec)
Measurement-6: 1244 Mbps  (67 usec)
Measurement-7: 1225 Mbps  (68 usec)
Measurement-8: 689 Mbps  (121 usec)
Measurement-9: 1157 Mbps  (72 usec)
Measurement-10: 1244 Mbps  (67 usec)
Measurement-11: 1208 Mbps  (69 usec)
Measurement-12: 1068 Mbps  (78 usec)
Measurement-13: 1225 Mbps  (68 usec)
Measurement-14: 1141 Mbps  (73 usec)
Measurement-15: 1004 Mbps  (83 usec)
Measurement-16: 1126 Mbps  (74 usec)
Measurement-17: 1208 Mbps  (69 usec)
Measurement-18: 1016 Mbps  (82 usec)
Measurement-19: 1126 Mbps  (74 usec)
Measurement-20: 1208 Mbps  (69 usec)
Measurement-21: 1174 Mbps  (71 usec)
Measurement-22: 1190 Mbps  (70 usec)
Measurement-23: 1174 Mbps  (71 usec)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1141 Mbps  (73 usec)
-> Train length:  8 - Packet size: 1488B -> 70% completed
Measurement-1: 1174 Mbps  (71 usec)
Measurement-2: 1174 Mbps  (71 usec)
Measurement-3: 1174 Mbps  (71 usec)
Measurement-4: 1157 Mbps  (72 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1157 Mbps  (72 usec)
Measurement-7: 1174 Mbps  (71 usec)
Measurement-8: 1157 Mbps  (72 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1157 Mbps  (72 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1174 Mbps  (71 usec)
Measurement-14: 1157 Mbps  (72 usec)
Measurement-15: 1208 Mbps  (69 usec)
Measurement-16: 1157 Mbps  (72 usec)
```

```
Measurement-17: 1157 Mbps  (72 usec)
Measurement-18: 2381 Mbps  (35 usec) (ignored)
Measurement-19: 1174 Mbps  (71 usec)
Measurement-20: 1157 Mbps  (72 usec)
Measurement-21: 1141 Mbps  (73 usec)
Measurement-22: 1157 Mbps  (72 usec)
Measurement-23: 1462 Mbps  (57 usec)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1244 Mbps  (67 usec)
-> Train length:  8 - Packet size: 1488B -> 72% completed
Measurement-1: 1190 Mbps  (70 usec)
Measurement-2: 1208 Mbps  (69 usec)
Measurement-3: 1174 Mbps  (71 usec)
Measurement-4: 1190 Mbps  (70 usec)
Measurement-5: 1174 Mbps  (71 usec)
Measurement-6: 1190 Mbps  (70 usec)
Measurement-7: 1208 Mbps  (69 usec)
Measurement-8: 1190 Mbps  (70 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1190 Mbps  (70 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1157 Mbps  (72 usec)
Measurement-14: 1174 Mbps  (71 usec)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1157 Mbps  (72 usec)
Measurement-17: 1263 Mbps  (66 usec)
Measurement-18: 1157 Mbps  (72 usec)
Measurement-19: 2315 Mbps  (36 usec) (ignored)
Measurement-20: 1225 Mbps  (68 usec)
Measurement-21: 1208 Mbps  (69 usec)
Measurement-22: 1157 Mbps  (72 usec)
Measurement-23: 1515 Mbps  (55 usec)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1190 Mbps  (70 usec)
-> Train length:  8 - Packet size: 1488B -> 75% completed
Measurement-1: 2315 Mbps  (36 usec) (ignored)
Measurement-2: 1208 Mbps  (69 usec)
Measurement-3: 1263 Mbps  (66 usec)
Measurement-4: 2381 Mbps  (35 usec) (ignored)
Measurement-5: 2451 Mbps  (34 usec) (ignored)
Measurement-6: 2315 Mbps  (36 usec) (ignored)
Measurement-7: 744 Mbps  (112 usec)
Measurement-8: 772 Mbps  (108 usec)
Measurement-9: 1225 Mbps  (68 usec)
Measurement-10: 1208 Mbps  (69 usec)
Measurement-11: 1225 Mbps  (68 usec)
Measurement-12: 1208 Mbps  (69 usec)
Measurement-13: 1208 Mbps  (69 usec)
Measurement-14: 1244 Mbps  (67 usec)
Measurement-15: 1174 Mbps  (71 usec)
Measurement-16: 1208 Mbps  (69 usec)
Measurement-17: 1225 Mbps  (68 usec)
Measurement-18: 980 Mbps  (85 usec)
Measurement-19: 1068 Mbps  (78 usec)
Measurement-20: 1323 Mbps  (63 usec)
Measurement-21: 1225 Mbps  (68 usec)
Measurement-22: 1208 Mbps  (69 usec)
Measurement-23: 1208 Mbps  (69 usec)
Measurement-24: 1174 Mbps  (71 usec)
Measurement-25: 2315 Mbps  (36 usec) (ignored)
```

```
-> Train length:  8 - Packet size: 1488B -> 77% completed
Measurement-1: 1174 Mbps  (71 usec)
Measurement-2: 1190 Mbps  (70 usec)
Measurement-3: 1174 Mbps  (71 usec)
Measurement-4: 2137 Mbps  (39 usec) (ignored)
Measurement-5: 1190 Mbps  (70 usec)
Measurement-6: 1190 Mbps  (70 usec)
Measurement-7: 1174 Mbps  (71 usec)
Measurement-8: 1174 Mbps  (71 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1190 Mbps  (70 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1174 Mbps  (71 usec)
Measurement-14: 1263 Mbps  (66 usec)
Measurement-15: 1190 Mbps  (70 usec)
Measurement-16: 1004 Mbps  (83 usec)
Measurement-17: 969 Mbps  (86 usec)
Measurement-18: 969 Mbps  (86 usec)
Measurement-19: 980 Mbps  (85 usec)
Measurement-20: 969 Mbps  (86 usec)
Measurement-21: 1174 Mbps  (71 usec)
Measurement-22: 1488 Mbps  (56 usec)
Measurement-23: 980 Mbps  (85 usec)
Measurement-24: 2778 Mbps  (30 usec) (ignored)
Measurement-25: 1344 Mbps  (62 usec)
-> Train length:  8 - Packet size: 1488B -> 80% completed
Measurement-1: 2381 Mbps  (35 usec) (ignored)
Measurement-2: 1282 Mbps  (65 usec)
Measurement-3: 1323 Mbps  (63 usec)
Measurement-4: 1208 Mbps  (69 usec)
Measurement-5: 1190 Mbps  (70 usec)
Measurement-6: 1190 Mbps  (70 usec)
Measurement-7: 1225 Mbps  (68 usec)
Measurement-8: 1225 Mbps  (68 usec)
Measurement-9: 1462 Mbps  (57 usec)
Measurement-10: 1190 Mbps  (70 usec)
Measurement-11: 2381 Mbps  (35 usec) (ignored)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1141 Mbps  (73 usec)
Measurement-14: 1111 Mbps  (75 usec)
Measurement-15: 1225 Mbps  (68 usec)
Measurement-16: 1157 Mbps  (72 usec)
Measurement-17: 622 Mbps  (134 usec)
Measurement-18: 617 Mbps  (135 usec)
Measurement-19: 1190 Mbps  (70 usec)
Measurement-20: 2381 Mbps  (35 usec) (ignored)
Measurement-21: 1302 Mbps  (64 usec)
Measurement-22: 2451 Mbps  (34 usec) (ignored)
Measurement-23: 1244 Mbps  (67 usec)
Measurement-24: 1225 Mbps  (68 usec)
Measurement-25: 1773 Mbps  (47 usec)
-> Train length:  8 - Packet size: 1488B -> 82% completed
Measurement-1: 1244 Mbps  (67 usec)
Measurement-2: 916 Mbps  (91 usec)
Measurement-3: 1225 Mbps  (68 usec)
Measurement-4: 1225 Mbps  (68 usec)
Measurement-5: 1773 Mbps  (47 usec)
Measurement-6: 1208 Mbps  (69 usec)
Measurement-7: 1157 Mbps  (72 usec)
Measurement-8: 1244 Mbps  (67 usec)
```

```
Measurement-9: 1208 Mbps  (69 usec)
Measurement-10: 1157 Mbps  (72 usec)
Measurement-11: 1190 Mbps  (70 usec)
Measurement-12: 1190 Mbps  (70 usec)
Measurement-13: 1208 Mbps  (69 usec)
Measurement-14: 1190 Mbps  (70 usec)
Measurement-15: 1141 Mbps  (73 usec)
Measurement-16: 1208 Mbps  (69 usec)
Measurement-17: 1190 Mbps  (70 usec)
Measurement-18: 1462 Mbps  (57 usec)
Measurement-19: 1225 Mbps  (68 usec)
Measurement-20: 1208 Mbps  (69 usec)
Measurement-21: 1225 Mbps  (68 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 2315 Mbps  (36 usec) (ignored)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1174 Mbps  (71 usec)
-> Train length:  8 - Packet size: 1488B -> 85% completed
Measurement-1: 1190 Mbps  (70 usec)
Measurement-2: 1174 Mbps  (71 usec)
Measurement-3: 1190 Mbps  (70 usec)
Measurement-4: 1208 Mbps  (69 usec)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1157 Mbps  (72 usec)
Measurement-7: 1174 Mbps  (71 usec)
Measurement-8: 1157 Mbps  (72 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1157 Mbps  (72 usec)
Measurement-11: 1174 Mbps  (71 usec)
Measurement-12: 1462 Mbps  (57 usec)
Measurement-13: 694 Mbps  (120 usec)
Measurement-14: 1190 Mbps  (70 usec)
Measurement-15: 1244 Mbps  (67 usec)
Measurement-16: 1190 Mbps  (70 usec)
Measurement-17: 1190 Mbps  (70 usec)
Measurement-18: 1174 Mbps  (71 usec)
Measurement-19: 1190 Mbps  (70 usec)
Measurement-20: 1174 Mbps  (71 usec)
Measurement-21: 1190 Mbps  (70 usec)
Measurement-22: 1190 Mbps  (70 usec)
Measurement-23: 1244 Mbps  (67 usec)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1208 Mbps  (69 usec)
-> Train length:  8 - Packet size: 1488B -> 87% completed
Measurement-1: 1174 Mbps  (71 usec)
Measurement-2: 1174 Mbps  (71 usec)
Measurement-3: 1190 Mbps  (70 usec)
Measurement-4: 1174 Mbps  (71 usec)
Measurement-5: 1174 Mbps  (71 usec)
Measurement-6: 1174 Mbps  (71 usec)
Measurement-7: 1174 Mbps  (71 usec)
Measurement-8: 1263 Mbps  (66 usec)
Measurement-9: 1190 Mbps  (70 usec)
Measurement-10: 1174 Mbps  (71 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1174 Mbps  (71 usec)
Measurement-13: 1157 Mbps  (72 usec)
Measurement-14: 1389 Mbps  (60 usec)
Measurement-15: 1174 Mbps  (71 usec)
Measurement-16: 1190 Mbps  (70 usec)
Measurement-17: 1190 Mbps  (70 usec)
```

```
Measurement-18: 1190 Mbps  (70 usec)
Measurement-19: 1263 Mbps  (66 usec)
Measurement-20: 1488 Mbps  (56 usec)
Measurement-21: 1366 Mbps  (61 usec)
Measurement-22: 2315 Mbps  (36 usec) (ignored)
Measurement-23: 1244 Mbps  (67 usec)
Measurement-24: 1773 Mbps  (47 usec)
Measurement-25: 1736 Mbps  (48 usec)
-> Train length:  8 - Packet size: 1488B -> 90% completed
Measurement-1: 936 Mbps  (89 usec)
Measurement-2: 1773 Mbps  (47 usec)
Measurement-3: 1225 Mbps  (68 usec)
Measurement-4: 992 Mbps  (84 usec)
Measurement-5: 1141 Mbps  (73 usec)
Measurement-6: 725 Mbps  (115 usec)
Measurement-7: 1208 Mbps  (69 usec)
Measurement-8: 1190 Mbps  (70 usec)
Measurement-9: 1208 Mbps  (69 usec)
Measurement-10: 1190 Mbps  (70 usec)
Measurement-11: 1190 Mbps  (70 usec)
Measurement-12: 1225 Mbps  (68 usec)
Measurement-13: 1208 Mbps  (69 usec)
Measurement-14: 1190 Mbps  (70 usec)
Measurement-15: 1190 Mbps  (70 usec)
Measurement-16: 1174 Mbps  (71 usec)
Measurement-17: 1174 Mbps  (71 usec)
Measurement-18: 1225 Mbps  (68 usec)
Measurement-19: 1190 Mbps  (70 usec)
Measurement-20: 1157 Mbps  (72 usec)
Measurement-21: 1157 Mbps  (72 usec)
Measurement-22: 1141 Mbps  (73 usec)
Measurement-23: 1141 Mbps  (73 usec)
Measurement-24: 1190 Mbps  (70 usec)
Measurement-25: 1157 Mbps  (72 usec)
-> Train length:  8 - Packet size: 1488B -> 92% completed
Measurement-1: 1157 Mbps  (72 usec)
Measurement-2: 1157 Mbps  (72 usec)
Measurement-3: 1157 Mbps  (72 usec)
Measurement-4: 2315 Mbps  (36 usec) (ignored)
Measurement-5: 1157 Mbps  (72 usec)
Measurement-6: 1174 Mbps  (71 usec)
Measurement-7: 1157 Mbps  (72 usec)
Measurement-8: 1208 Mbps  (69 usec)
Measurement-9: 1174 Mbps  (71 usec)
Measurement-10: 1174 Mbps  (71 usec)
Measurement-11: 1157 Mbps  (72 usec)
Measurement-12: 1157 Mbps  (72 usec)
Measurement-13: 1157 Mbps  (72 usec)
Measurement-14: 1157 Mbps  (72 usec)
Measurement-15: 1157 Mbps  (72 usec)
Measurement-16: 1190 Mbps  (70 usec)
Measurement-17: 1157 Mbps  (72 usec)
Measurement-18: 1174 Mbps  (71 usec)
Measurement-19: 1190 Mbps  (70 usec)
Measurement-20: 1190 Mbps  (70 usec)
Measurement-21: 1157 Mbps  (72 usec)
Measurement-22: 1174 Mbps  (71 usec)
Measurement-23: 1174 Mbps  (71 usec)
Measurement-24: 1174 Mbps  (71 usec)
Measurement-25: 1157 Mbps  (72 usec)
-> Train length:  8 - Packet size: 1488B -> 95% completed
```

```
        Measurement-1: 1157 Mbps  (72 usec)
        Measurement-2: 1157 Mbps  (72 usec)
        Measurement-3: 2252 Mbps  (37 usec) (ignored)
        Measurement-4: 1190 Mbps  (70 usec)
        Measurement-5: 1174 Mbps  (71 usec)
        Measurement-6: 1174 Mbps  (71 usec)
        Measurement-7: 1190 Mbps  (70 usec)
        Measurement-8: 1225 Mbps  (68 usec)
        Measurement-9: 1174 Mbps  (71 usec)
        Measurement-10: 1174 Mbps  (71 usec)
        Measurement-11: 1174 Mbps  (71 usec)
        Measurement-12: 1366 Mbps  (61 usec)
        Measurement-13: 1126 Mbps  (74 usec)
        Measurement-14: 1263 Mbps  (66 usec)
        Measurement-15: 1208 Mbps  (69 usec)
        Measurement-16: 1208 Mbps  (69 usec)
        Measurement-17: 1208 Mbps  (69 usec)
        Measurement-18: 1208 Mbps  (69 usec)
        Measurement-19: 980 Mbps  (85 usec)
        Measurement-20: 1208 Mbps  (69 usec)
        Measurement-21: 1190 Mbps  (70 usec)
        Measurement-22: 2315 Mbps  (36 usec) (ignored)
        Measurement-23: 1225 Mbps  (68 usec)
        Measurement-24: 1225 Mbps  (68 usec)
        Measurement-25: 1701 Mbps  (49 usec)
        -> Train length:  8 - Packet size: 1488B -> 97% completed
        Measurement-1: 737 Mbps  (113 usec)
        Measurement-2: 1208 Mbps  (69 usec)
        Measurement-3: 980 Mbps  (85 usec)
        Measurement-4: 1190 Mbps  (70 usec)
        Measurement-5: 1208 Mbps  (69 usec)
        Measurement-6: 1208 Mbps  (69 usec)
        Measurement-7: 1174 Mbps  (71 usec)
        Measurement-8: 1208 Mbps  (69 usec)
        Measurement-9: 1437 Mbps  (58 usec)
        Measurement-10: 1190 Mbps  (70 usec)
        Measurement-11: 1208 Mbps  (69 usec)
        Measurement-12: 1174 Mbps  (71 usec)
        Measurement-13: 1208 Mbps  (69 usec)
        Measurement-14: 1190 Mbps  (70 usec)
        Measurement-15: 1208 Mbps  (69 usec)
        Measurement-16: 1190 Mbps  (70 usec)
        Measurement-17: 1157 Mbps  (72 usec)
        Measurement-18: 1157 Mbps  (72 usec)
        Measurement-19: 1190 Mbps  (70 usec)
        Measurement-20: 1174 Mbps  (71 usec)
        Measurement-21: 1157 Mbps  (72 usec)
        Measurement-22: 1157 Mbps  (72 usec)
        Measurement-23: 1157 Mbps  (72 usec)
        Measurement-24: 1208 Mbps  (69 usec)
        Measurement-25: 1174 Mbps  (71 usec)
```

-- Local modes : In Phase I --

*** Mode: 1156 Mbps to 1212 Mbps - 290 measurements**
Modal bell: 674 measurements - low : 1050 Mbps - high : 1412 Mbps
*** Mode: 299 Mbps to 355 Mbps - 65 measurements**
Modal bell: 91 measurements - low : 159 Mbps - high : 463 Mbps
*** Mode: 965 Mbps to 1021 Mbps - 20 measurements**
Modal bell: 43 measurements - low : 859 Mbps - high : 1068 Mbps

```
-- Phase II: Estimate Average Dispersion Rate (ADR) --

      -- Number of trains: 500 - Train length: 48 - Packet size: 1488B
      Measurement-   1 out of 500: 942 Mbps  (594 usec)
      Measurement-   2 out of 500: 971 Mbps  (576 usec)
      Measurement-   3 out of 500: 955 Mbps  (586 usec)
      Measurement-   4 out of 500: 971 Mbps  (576 usec)
      Measurement-   5 out of 500: 854 Mbps  (655 usec)
      Measurement-   6 out of 500: 960 Mbps  (583 usec)
      Measurement-   7 out of 500: 965 Mbps  (580 usec)
      Measurement-   8 out of 500: 963 Mbps  (581 usec)
      Measurement-   9 out of 500: 965 Mbps  (580 usec)
      Measurement-  10 out of 500: 966 Mbps  (579 usec)
      Measurement-  11 out of 500: 961 Mbps  (582 usec)
      Measurement-  12 out of 500: 952 Mbps  (588 usec)
      Measurement-  13 out of 500: 950 Mbps  (589 usec)
      Measurement-  14 out of 500: 943 Mbps  (593 usec)
      Measurement-  15 out of 500: 945 Mbps  (592 usec)
      Measurement-  16 out of 500: 939 Mbps  (596 usec)
      Measurement-  17 out of 500: 950 Mbps  (589 usec)
      Measurement-  18 out of 500: 963 Mbps  (581 usec)
      Measurement-  19 out of 500: 939 Mbps  (596 usec)
      Measurement-  20 out of 500: 970 Mbps  (577 usec)
      Measurement-  21 out of 500: 919 Mbps  (609 usec)
      Measurement-  22 out of 500: 911 Mbps  (614 usec)
      Measurement-  23 out of 500: 937 Mbps  (597 usec)
      Measurement-  24 out of 500: 952 Mbps  (588 usec)
      Measurement-  25 out of 500: 950 Mbps  (589 usec)
      Measurement-  26 out of 500: 952 Mbps  (588 usec)
      Measurement-  27 out of 500: 932 Mbps  (600 usec)
      Measurement-  28 out of 500: 952 Mbps  (588 usec)
      Measurement-  29 out of 500: 982 Mbps  (570 usec)
      Measurement-  30 out of 500: 952 Mbps  (588 usec)
      Measurement-  31 out of 500: 958 Mbps  (584 usec)
      Measurement-  32 out of 500: 952 Mbps  (588 usec)
      Measurement-  33 out of 500: 945 Mbps  (592 usec)
      Measurement-  34 out of 500: 950 Mbps  (589 usec)
      Measurement-  35 out of 500: 945 Mbps  (592 usec)
      Measurement-  36 out of 500: 952 Mbps  (588 usec)
      Measurement-  37 out of 500: 942 Mbps  (594 usec)
      Measurement-  38 out of 500: 960 Mbps  (583 usec)
      Measurement-  39 out of 500: 963 Mbps  (581 usec)
      Measurement-  40 out of 500: 966 Mbps  (579 usec)
      Measurement-  41 out of 500: 968 Mbps  (578 usec)
      Measurement-  42 out of 500: 907 Mbps  (617 usec)
      Measurement-  43 out of 500: 880 Mbps  (636 usec)
      Measurement-  44 out of 500: 968 Mbps  (578 usec)
      Measurement-  45 out of 500: 965 Mbps  (580 usec)
      Measurement-  46 out of 500: 968 Mbps  (578 usec)
      Measurement-  47 out of 500: 948 Mbps  (590 usec)
      Measurement-  48 out of 500: 942 Mbps  (594 usec)
      Measurement-  49 out of 500: 966 Mbps  (579 usec)
      Measurement-  50 out of 500: 963 Mbps  (581 usec)
      Measurement-  51 out of 500: 953 Mbps  (587 usec)
      Measurement-  52 out of 500: 968 Mbps  (578 usec)
      Measurement-  53 out of 500: 970 Mbps  (577 usec)
      Measurement-  54 out of 500: 948 Mbps  (590 usec)
      Measurement-  55 out of 500: 945 Mbps  (592 usec)
      Measurement-  56 out of 500: 914 Mbps  (612 usec)
```

```
Measurement-  57 out of 500: 952 Mbps  (588 usec)
Measurement-  58 out of 500: 952 Mbps  (588 usec)
Measurement-  59 out of 500: 952 Mbps  (588 usec)
Measurement-  60 out of 500: 952 Mbps  (588 usec)
Measurement-  61 out of 500: 953 Mbps  (587 usec)
Measurement-  62 out of 500: 950 Mbps  (589 usec)
Measurement-  63 out of 500: 952 Mbps  (588 usec)
Measurement-  64 out of 500: 953 Mbps  (587 usec)
Measurement-  65 out of 500: 932 Mbps  (600 usec)
Measurement-  66 out of 500: 985 Mbps  (568 usec)
Measurement-  67 out of 500: 953 Mbps  (587 usec)
Measurement-  68 out of 500: 950 Mbps  (589 usec)
Measurement-  69 out of 500: 948 Mbps  (590 usec)
Measurement-  70 out of 500: 948 Mbps  (590 usec)
Measurement-  71 out of 500: 966 Mbps  (579 usec)
Measurement-  72 out of 500: 947 Mbps  (591 usec)
Measurement-  73 out of 500: 960 Mbps  (583 usec)
Measurement-  74 out of 500: 943 Mbps  (593 usec)
Measurement-  75 out of 500: 961 Mbps  (582 usec)
Measurement-  76 out of 500: 966 Mbps  (579 usec)
Measurement-  77 out of 500: 963 Mbps  (581 usec)
Measurement-  78 out of 500: 953 Mbps  (587 usec)
Measurement-  79 out of 500: 968 Mbps  (578 usec)
Measurement-  80 out of 500: 968 Mbps  (578 usec)
Measurement-  81 out of 500: 968 Mbps  (578 usec)
Measurement-  82 out of 500: 985 Mbps  (568 usec)
Measurement-  83 out of 500: 968 Mbps  (578 usec)
Measurement-  84 out of 500: 976 Mbps  (573 usec)
Measurement-  85 out of 500: 965 Mbps  (580 usec)
Measurement-  86 out of 500: 992 Mbps  (564 usec)
Measurement-  87 out of 500: 975 Mbps  (574 usec)
Measurement-  88 out of 500: 952 Mbps  (588 usec)
Measurement-  89 out of 500: 985 Mbps  (568 usec)
Measurement-  90 out of 500: 947 Mbps  (591 usec)
Measurement-  91 out of 500: 940 Mbps  (595 usec)
Measurement-  92 out of 500: 952 Mbps  (588 usec)
Measurement-  93 out of 500: 952 Mbps  (588 usec)
Measurement-  94 out of 500: 948 Mbps  (590 usec)
Measurement-  95 out of 500: 961 Mbps  (582 usec)
Measurement-  96 out of 500: 1019 Mbps  (549 usec)
Measurement-  97 out of 500: 956 Mbps  (585 usec)
Measurement-  98 out of 500: 920 Mbps  (608 usec)
Measurement-  99 out of 500: 950 Mbps  (589 usec)
Measurement- 100 out of 500: 982 Mbps  (570 usec)
Measurement- 101 out of 500: 978 Mbps  (572 usec)
Measurement- 102 out of 500: 985 Mbps  (568 usec)
Measurement- 103 out of 500: 952 Mbps  (588 usec)
Measurement- 104 out of 500: 952 Mbps  (588 usec)
Measurement- 105 out of 500: 943 Mbps  (593 usec)
Measurement- 106 out of 500: 966 Mbps  (579 usec)
Measurement- 107 out of 500: 948 Mbps  (590 usec)
Measurement- 108 out of 500: 926 Mbps  (604 usec)
Measurement- 109 out of 500: 983 Mbps  (569 usec)
Measurement- 110 out of 500: 947 Mbps  (591 usec)
Measurement- 111 out of 500: 950 Mbps  (589 usec)
Measurement- 112 out of 500: 942 Mbps  (594 usec)
Measurement- 113 out of 500: 955 Mbps  (586 usec)
Measurement- 114 out of 500: 966 Mbps  (579 usec)
Measurement- 115 out of 500: 970 Mbps  (577 usec)
Measurement- 116 out of 500: 983 Mbps  (569 usec)
Measurement- 117 out of 500: 858 Mbps  (652 usec)
```

```
Measurement- 118 out of 500: 880 Mbps  (636 usec)
Measurement- 119 out of 500: 894 Mbps  (626 usec)
Measurement- 120 out of 500: 911 Mbps  (614 usec)
Measurement- 121 out of 500: 1012 Mbps  (553 usec)
Measurement- 122 out of 500: 970 Mbps  (577 usec)
Measurement- 123 out of 500: 947 Mbps  (591 usec)
Measurement- 124 out of 500: 948 Mbps  (590 usec)
Measurement- 125 out of 500: 948 Mbps  (590 usec)
Measurement- 126 out of 500: 960 Mbps  (583 usec)
Measurement- 127 out of 500: 948 Mbps  (590 usec)
Measurement- 128 out of 500: 987 Mbps  (567 usec)
Measurement- 129 out of 500: 982 Mbps  (570 usec)
Measurement- 130 out of 500: 1001 Mbps  (559 usec)
Measurement- 131 out of 500: 980 Mbps  (571 usec)
Measurement- 132 out of 500: 922 Mbps  (607 usec)
Measurement- 133 out of 500: 955 Mbps  (586 usec)
Measurement- 134 out of 500: 1042 Mbps  (537 usec)
Measurement- 135 out of 500: 947 Mbps  (591 usec)
Measurement- 136 out of 500: 987 Mbps  (567 usec)
Measurement- 137 out of 500: 932 Mbps  (600 usec)
Measurement- 138 out of 500: 953 Mbps  (587 usec)
Measurement- 139 out of 500: 976 Mbps  (573 usec)
Measurement- 140 out of 500: 952 Mbps  (588 usec)
Measurement- 141 out of 500: 829 Mbps  (675 usec)
Measurement- 142 out of 500: 952 Mbps  (588 usec)
Measurement- 143 out of 500: 943 Mbps  (593 usec)
Measurement- 144 out of 500: 948 Mbps  (590 usec)
Measurement- 145 out of 500: 945 Mbps  (592 usec)
Measurement- 146 out of 500: 939 Mbps  (596 usec)
Measurement- 147 out of 500: 940 Mbps  (595 usec)
Measurement- 148 out of 500: 777 Mbps  (720 usec)
Measurement- 149 out of 500: 788 Mbps  (710 usec)
Measurement- 150 out of 500: 966 Mbps  (579 usec)
Measurement- 151 out of 500: 943 Mbps  (593 usec)
Measurement- 152 out of 500: 950 Mbps  (589 usec)
Measurement- 153 out of 500: 877 Mbps  (638 usec)
Measurement- 154 out of 500: 881 Mbps  (635 usec)
Measurement- 155 out of 500: 898 Mbps  (623 usec)
Measurement- 156 out of 500: 920 Mbps  (608 usec)
Measurement- 157 out of 500: 975 Mbps  (574 usec)
Measurement- 158 out of 500: 966 Mbps  (579 usec)
Measurement- 159 out of 500: 952 Mbps  (588 usec)
Measurement- 160 out of 500: 945 Mbps  (592 usec)
Measurement- 161 out of 500: 992 Mbps  (564 usec)
Measurement- 162 out of 500: 978 Mbps  (572 usec)
Measurement- 163 out of 500: 973 Mbps  (575 usec)
Measurement- 164 out of 500: 952 Mbps  (588 usec)
Measurement- 165 out of 500: 950 Mbps  (589 usec)
Measurement- 166 out of 500: 956 Mbps  (585 usec)
Measurement- 167 out of 500: 953 Mbps  (587 usec)
Measurement- 168 out of 500: 965 Mbps  (580 usec)
Measurement- 169 out of 500: 953 Mbps  (587 usec)
Measurement- 170 out of 500: 950 Mbps  (589 usec)
Measurement- 171 out of 500: 923 Mbps  (606 usec)
Measurement- 172 out of 500: 950 Mbps  (589 usec)
Measurement- 173 out of 500: 955 Mbps  (586 usec)
Measurement- 174 out of 500: 952 Mbps  (588 usec)
Measurement- 175 out of 500: 952 Mbps  (588 usec)
Measurement- 176 out of 500: 966 Mbps  (579 usec)
Measurement- 177 out of 500: 971 Mbps  (576 usec)
Measurement- 178 out of 500: 950 Mbps  (589 usec)
```

```
Measurement- 179 out of 500: 943 Mbps  (593 usec)
Measurement- 180 out of 500: 948 Mbps  (590 usec)
Measurement- 181 out of 500: 948 Mbps  (590 usec)
Measurement- 182 out of 500: 947 Mbps  (591 usec)
Measurement- 183 out of 500: 960 Mbps  (583 usec)
Measurement- 184 out of 500: 948 Mbps  (590 usec)
Measurement- 185 out of 500: 737 Mbps  (759 usec)
Measurement- 186 out of 500: 966 Mbps  (579 usec)
Measurement- 187 out of 500: 789 Mbps  (709 usec)
Measurement- 188 out of 500: 976 Mbps  (573 usec)
Measurement- 189 out of 500: 961 Mbps  (582 usec)
Measurement- 190 out of 500: 970 Mbps  (577 usec)
Measurement- 191 out of 500: 966 Mbps  (579 usec)
Measurement- 192 out of 500: 889 Mbps  (629 usec)
Measurement- 193 out of 500: 961 Mbps  (582 usec)
Measurement- 194 out of 500: 970 Mbps  (577 usec)
Measurement- 195 out of 500: 965 Mbps  (580 usec)
Measurement- 196 out of 500: 955 Mbps  (586 usec)
Measurement- 197 out of 500: 968 Mbps  (578 usec)
Measurement- 198 out of 500: 968 Mbps  (578 usec)
Measurement- 199 out of 500: 966 Mbps  (579 usec)
Measurement- 200 out of 500: 968 Mbps  (578 usec)
Measurement- 201 out of 500: 980 Mbps  (571 usec)
Measurement- 202 out of 500: 968 Mbps  (578 usec)
Measurement- 203 out of 500: 1010 Mbps  (554 usec)
Measurement- 204 out of 500: 961 Mbps  (582 usec)
Measurement- 205 out of 500: 963 Mbps  (581 usec)
Measurement- 206 out of 500: 948 Mbps  (590 usec)
Measurement- 207 out of 500: 955 Mbps  (586 usec)
Measurement- 208 out of 500: 934 Mbps  (599 usec)
Measurement- 209 out of 500: 952 Mbps  (588 usec)
Measurement- 210 out of 500: 950 Mbps  (589 usec)
Measurement- 211 out of 500: 952 Mbps  (588 usec)
Measurement- 212 out of 500: 987 Mbps  (567 usec)
Measurement- 213 out of 500: 952 Mbps  (588 usec)
Measurement- 214 out of 500: 952 Mbps  (588 usec)
Measurement- 215 out of 500: 948 Mbps  (590 usec)
Measurement- 216 out of 500: 943 Mbps  (593 usec)
Measurement- 217 out of 500: 948 Mbps  (590 usec)
Measurement- 218 out of 500: 923 Mbps  (606 usec)
Measurement- 219 out of 500: 960 Mbps  (583 usec)
Measurement- 220 out of 500: 943 Mbps  (593 usec)
Measurement- 221 out of 500: 963 Mbps  (581 usec)
Measurement- 222 out of 500: 942 Mbps  (594 usec)
Measurement- 223 out of 500: 939 Mbps  (596 usec)
Measurement- 224 out of 500: 965 Mbps  (580 usec)
Measurement- 225 out of 500: 950 Mbps  (589 usec)
Measurement- 226 out of 500: 950 Mbps  (589 usec)
Measurement- 227 out of 500: 973 Mbps  (575 usec)
Measurement- 228 out of 500: 960 Mbps  (583 usec)
Measurement- 229 out of 500: 878 Mbps  (637 usec)
Measurement- 230 out of 500: 887 Mbps  (631 usec)
Measurement- 231 out of 500: 956 Mbps  (585 usec)
Measurement- 232 out of 500: 963 Mbps  (581 usec)
Measurement- 233 out of 500: 952 Mbps  (588 usec)
Measurement- 234 out of 500: 952 Mbps  (588 usec)
Measurement- 235 out of 500: 945 Mbps  (592 usec)
Measurement- 236 out of 500: 934 Mbps  (599 usec)
Measurement- 237 out of 500: 950 Mbps  (589 usec)
Measurement- 238 out of 500: 950 Mbps  (589 usec)
Measurement- 239 out of 500: 950 Mbps  (589 usec)
```

```
Measurement- 240 out of 500: 948 Mbps  (590 usec)
Measurement- 241 out of 500: 950 Mbps  (589 usec)
Measurement- 242 out of 500: 999 Mbps  (560 usec)
Measurement- 243 out of 500: 1008 Mbps  (555 usec)
Measurement- 244 out of 500: 966 Mbps  (579 usec)
Measurement- 245 out of 500: 955 Mbps  (586 usec)
Measurement- 246 out of 500: 929 Mbps  (602 usec)
Measurement- 247 out of 500: 952 Mbps  (588 usec)
Measurement- 248 out of 500: 988 Mbps  (566 usec)
Measurement- 249 out of 500: 947 Mbps  (591 usec)
Measurement- 250 out of 500: 952 Mbps  (588 usec)
Measurement- 251 out of 500: 762 Mbps  (734 usec)
Measurement- 252 out of 500: 948 Mbps  (590 usec)
Measurement- 253 out of 500: 960 Mbps  (583 usec)
Measurement- 254 out of 500: 999 Mbps  (560 usec)
Measurement- 255 out of 500: 987 Mbps  (567 usec)
Measurement- 256 out of 500: 953 Mbps  (587 usec)
Measurement- 257 out of 500: 916 Mbps  (611 usec)
Measurement- 258 out of 500: 759 Mbps  (737 usec)
Measurement- 259 out of 500: 975 Mbps  (574 usec)
Measurement- 260 out of 500: 776 Mbps  (721 usec)
Measurement- 261 out of 500: 997 Mbps  (561 usec)
Measurement- 262 out of 500: 978 Mbps  (572 usec)
Measurement- 263 out of 500: 982 Mbps  (570 usec)
Measurement- 264 out of 500: 965 Mbps  (580 usec)
Measurement- 265 out of 500: 884 Mbps  (633 usec)
Measurement- 266 out of 500: 965 Mbps  (580 usec)
Measurement- 267 out of 500: 958 Mbps  (584 usec)
Measurement- 268 out of 500: 881 Mbps  (635 usec)
Measurement- 269 out of 500: 963 Mbps  (581 usec)
Measurement- 270 out of 500: 936 Mbps  (598 usec)
Measurement- 271 out of 500: 982 Mbps  (570 usec)
Measurement- 272 out of 500: 996 Mbps  (562 usec)
Measurement- 273 out of 500: 948 Mbps  (590 usec)
Measurement- 274 out of 500: 940 Mbps  (595 usec)
Measurement- 275 out of 500: 999 Mbps  (560 usec)
Measurement- 276 out of 500: 988 Mbps  (566 usec)
Measurement- 277 out of 500: 1019 Mbps  (549 usec)
Measurement- 278 out of 500: 926 Mbps  (604 usec)
Measurement- 279 out of 500: 1036 Mbps  (540 usec)
Measurement- 280 out of 500: 919 Mbps  (609 usec)
Measurement- 281 out of 500: 976 Mbps  (573 usec)
Measurement- 282 out of 500: 990 Mbps  (565 usec)
Measurement- 283 out of 500: 955 Mbps  (586 usec)
Measurement- 284 out of 500: 940 Mbps  (595 usec)
Measurement- 285 out of 500: 975 Mbps  (574 usec)
Measurement- 286 out of 500: 968 Mbps  (578 usec)
Measurement- 287 out of 500: 961 Mbps  (582 usec)
Measurement- 288 out of 500: 963 Mbps  (581 usec)
Measurement- 289 out of 500: 948 Mbps  (590 usec)
Measurement- 290 out of 500: 945 Mbps  (592 usec)
Measurement- 291 out of 500: 885 Mbps  (632 usec)
Measurement- 292 out of 500: 987 Mbps  (567 usec)
Measurement- 293 out of 500: 960 Mbps  (583 usec)
Measurement- 294 out of 500: 973 Mbps  (575 usec)
Measurement- 295 out of 500: 963 Mbps  (581 usec)
Measurement- 296 out of 500: 980 Mbps  (571 usec)
Measurement- 297 out of 500: 800 Mbps  (699 usec)
Measurement- 298 out of 500: 968 Mbps  (578 usec)
Measurement- 299 out of 500: 976 Mbps  (573 usec)
Measurement- 300 out of 500: 848 Mbps  (660 usec)
```

```
Measurement- 301 out of 500: 992 Mbps  (564 usec)
Measurement- 302 out of 500: 982 Mbps  (570 usec)
Measurement- 303 out of 500: 889 Mbps  (629 usec)
Measurement- 304 out of 500: 904 Mbps  (619 usec)
Measurement- 305 out of 500: 958 Mbps  (584 usec)
Measurement- 306 out of 500: 963 Mbps  (581 usec)
Measurement- 307 out of 500: 953 Mbps  (587 usec)
Measurement- 308 out of 500: 997 Mbps  (561 usec)
Measurement- 309 out of 500: 987 Mbps  (567 usec)
Measurement- 310 out of 500: 985 Mbps  (568 usec)
Measurement- 311 out of 500: 966 Mbps  (579 usec)
Measurement- 312 out of 500: 1042 Mbps  (537 usec)
Measurement- 313 out of 500: 956 Mbps  (585 usec)
Measurement- 314 out of 500: 1012 Mbps  (553 usec)
Measurement- 315 out of 500: 953 Mbps  (587 usec)
Measurement- 316 out of 500: 953 Mbps  (587 usec)
Measurement- 317 out of 500: 926 Mbps  (604 usec)
Measurement- 318 out of 500: 1014 Mbps  (552 usec)
Measurement- 319 out of 500: 958 Mbps  (584 usec)
Measurement- 320 out of 500: 928 Mbps  (603 usec)
Measurement- 321 out of 500: 857 Mbps  (653 usec)
Measurement- 322 out of 500: 963 Mbps  (581 usec)
Measurement- 323 out of 500: 966 Mbps  (579 usec)
Measurement- 324 out of 500: 953 Mbps  (587 usec)
Measurement- 325 out of 500: 950 Mbps  (589 usec)
Measurement- 326 out of 500: 885 Mbps  (632 usec)
Measurement- 327 out of 500: 961 Mbps  (582 usec)
Measurement- 328 out of 500: 948 Mbps  (590 usec)
Measurement- 329 out of 500: 1001 Mbps  (559 usec)
Measurement- 330 out of 500: 976 Mbps  (573 usec)
Measurement- 331 out of 500: 950 Mbps  (589 usec)
Measurement- 332 out of 500: 978 Mbps  (572 usec)
Measurement- 333 out of 500: 982 Mbps  (570 usec)
Measurement- 334 out of 500: 922 Mbps  (607 usec)
Measurement- 335 out of 500: 952 Mbps  (588 usec)
Measurement- 336 out of 500: 992 Mbps  (564 usec)
Measurement- 337 out of 500: 845 Mbps  (662 usec)
Measurement- 338 out of 500: 968 Mbps  (578 usec)
Measurement- 339 out of 500: 963 Mbps  (581 usec)
Measurement- 340 out of 500: 970 Mbps  (577 usec)
Measurement- 341 out of 500: 961 Mbps  (582 usec)
Measurement- 342 out of 500: 966 Mbps  (579 usec)
Measurement- 343 out of 500: 983 Mbps  (569 usec)
Measurement- 344 out of 500: 958 Mbps  (584 usec)
Measurement- 345 out of 500: 956 Mbps  (585 usec)
Measurement- 346 out of 500: 948 Mbps  (590 usec)
Measurement- 347 out of 500: 950 Mbps  (589 usec)
Measurement- 348 out of 500: 966 Mbps  (579 usec)
Measurement- 349 out of 500: 1001 Mbps  (559 usec)
Measurement- 350 out of 500: 953 Mbps  (587 usec)
Measurement- 351 out of 500: 950 Mbps  (589 usec)
Measurement- 352 out of 500: 1097 Mbps  (510 usec)
Measurement- 353 out of 500: 997 Mbps  (561 usec)
Measurement- 354 out of 500: 942 Mbps  (594 usec)
Measurement- 355 out of 500: 961 Mbps  (582 usec)
Measurement- 356 out of 500: 966 Mbps  (579 usec)
Measurement- 357 out of 500: 960 Mbps  (583 usec)
Measurement- 358 out of 500: 937 Mbps  (597 usec)
Measurement- 359 out of 500: 952 Mbps  (588 usec)
Measurement- 360 out of 500: 965 Mbps  (580 usec)
Measurement- 361 out of 500: 983 Mbps  (569 usec)
```

```
Measurement- 362 out of 500: 952 Mbps  (588 usec)
Measurement- 363 out of 500: 961 Mbps  (582 usec)
Measurement- 364 out of 500: 961 Mbps  (582 usec)
Measurement- 365 out of 500: 945 Mbps  (592 usec)
Measurement- 366 out of 500: 920 Mbps  (608 usec)
Measurement- 367 out of 500: 746 Mbps  (750 usec)
Measurement- 368 out of 500: 970 Mbps  (577 usec)
Measurement- 369 out of 500: 813 Mbps  (688 usec)
Measurement- 370 out of 500: 982 Mbps  (570 usec)
Measurement- 371 out of 500: 955 Mbps  (586 usec)
Measurement- 372 out of 500: 982 Mbps  (570 usec)
Measurement- 373 out of 500: 980 Mbps  (571 usec)
Measurement- 374 out of 500: 973 Mbps  (575 usec)
Measurement- 375 out of 500: 997 Mbps  (561 usec)
Measurement- 376 out of 500: 1121 Mbps  (499 usec)
Measurement- 377 out of 500: 874 Mbps  (640 usec)
Measurement- 378 out of 500: 978 Mbps  (572 usec)
Measurement- 379 out of 500: 894 Mbps  (626 usec)
Measurement- 380 out of 500: 936 Mbps  (598 usec)
Measurement- 381 out of 500: 940 Mbps  (595 usec)
Measurement- 382 out of 500: 965 Mbps  (580 usec)
Measurement- 383 out of 500: 985 Mbps  (568 usec)
Measurement- 384 out of 500: 945 Mbps  (592 usec)
Measurement- 385 out of 500: 958 Mbps  (584 usec)
Measurement- 386 out of 500: 953 Mbps  (587 usec)
Measurement- 387 out of 500: 994 Mbps  (563 usec)
Measurement- 388 out of 500: 1028 Mbps  (544 usec)
Measurement- 389 out of 500: 983 Mbps  (569 usec)
Measurement- 390 out of 500: 1163 Mbps  (481 usec)
Measurement- 391 out of 500: 970 Mbps  (577 usec)
Measurement- 392 out of 500: 948 Mbps  (590 usec)
Measurement- 393 out of 500: 992 Mbps  (564 usec)
Measurement- 394 out of 500: 1008 Mbps  (555 usec)
Measurement- 395 out of 500: 983 Mbps  (569 usec)
Measurement- 396 out of 500: 922 Mbps  (607 usec)
Measurement- 397 out of 500: 961 Mbps  (582 usec)
Measurement- 398 out of 500: 955 Mbps  (586 usec)
Measurement- 399 out of 500: 961 Mbps  (582 usec)
Measurement- 400 out of 500: 961 Mbps  (582 usec)
Measurement- 401 out of 500: 956 Mbps  (585 usec)
Measurement- 402 out of 500: 746 Mbps  (750 usec)
Measurement- 403 out of 500: 985 Mbps  (568 usec)
Measurement- 404 out of 500: 749 Mbps  (747 usec)
Measurement- 405 out of 500: 973 Mbps  (575 usec)
Measurement- 406 out of 500: 965 Mbps  (580 usec)
Measurement- 407 out of 500: 968 Mbps  (578 usec)
Measurement- 408 out of 500: 932 Mbps  (600 usec)
Measurement- 409 out of 500: 982 Mbps  (570 usec)
Measurement- 410 out of 500: 968 Mbps  (578 usec)
Measurement- 411 out of 500: 892 Mbps  (627 usec)
Measurement- 412 out of 500: 882 Mbps  (634 usec)
Measurement- 413 out of 500: 892 Mbps  (627 usec)
Measurement- 414 out of 500: 978 Mbps  (572 usec)
Measurement- 415 out of 500: 891 Mbps  (628 usec)
Measurement- 416 out of 500: 950 Mbps  (589 usec)
Measurement- 417 out of 500: 958 Mbps  (584 usec)
Measurement- 418 out of 500: 1010 Mbps  (554 usec)
Measurement- 419 out of 500: 950 Mbps  (589 usec)
Measurement- 420 out of 500: 963 Mbps  (581 usec)
Measurement- 421 out of 500: 958 Mbps  (584 usec)
Measurement- 422 out of 500: 990 Mbps  (565 usec)
```

```
Measurement- 423 out of 500: 990 Mbps  (565 usec)
Measurement- 424 out of 500: 1048 Mbps  (534 usec)
Measurement- 425 out of 500: 1046 Mbps  (535 usec)
Measurement- 426 out of 500: 939 Mbps  (596 usec)
Measurement- 427 out of 500: 1161 Mbps  (482 usec)
Measurement- 428 out of 500: 939 Mbps  (596 usec)
Measurement- 429 out of 500: 965 Mbps  (580 usec)
Measurement- 430 out of 500: 978 Mbps  (572 usec)
Measurement- 431 out of 500: 960 Mbps  (583 usec)
Measurement- 432 out of 500: 987 Mbps  (567 usec)
Measurement- 433 out of 500: 965 Mbps  (580 usec)
Measurement- 434 out of 500: 987 Mbps  (567 usec)
Measurement- 435 out of 500: 992 Mbps  (564 usec)
Measurement- 436 out of 500: 922 Mbps  (607 usec)
Measurement- 437 out of 500: 952 Mbps  (588 usec)
Measurement- 438 out of 500: 958 Mbps  (584 usec)
Measurement- 439 out of 500: 983 Mbps  (569 usec)
Measurement- 440 out of 500: 746 Mbps  (750 usec)
Measurement- 441 out of 500: 961 Mbps  (582 usec)
Measurement- 442 out of 500: 968 Mbps  (578 usec)
Measurement- 443 out of 500: 961 Mbps  (582 usec)
Measurement- 444 out of 500: 975 Mbps  (574 usec)
Measurement- 445 out of 500: 983 Mbps  (569 usec)
Measurement- 446 out of 500: 953 Mbps  (587 usec)
Measurement- 447 out of 500: 843 Mbps  (664 usec)
Measurement- 448 out of 500: 865 Mbps  (647 usec)
Measurement- 449 out of 500: 945 Mbps  (592 usec)
Measurement- 450 out of 500: 887 Mbps  (631 usec)
Measurement- 451 out of 500: 965 Mbps  (580 usec)
Measurement- 452 out of 500: 948 Mbps  (590 usec)
Measurement- 453 out of 500: 953 Mbps  (587 usec)
Measurement- 454 out of 500: 997 Mbps  (561 usec)
Measurement- 455 out of 500: 952 Mbps  (588 usec)
Measurement- 456 out of 500: 958 Mbps  (584 usec)
Measurement- 457 out of 500: 907 Mbps  (617 usec)
Measurement- 458 out of 500: 952 Mbps  (588 usec)
Measurement- 459 out of 500: 996 Mbps  (562 usec)
Measurement- 460 out of 500: 994 Mbps  (563 usec)
Measurement- 461 out of 500: 963 Mbps  (581 usec)
Measurement- 462 out of 500: 987 Mbps  (567 usec)
Measurement- 463 out of 500: 965 Mbps  (580 usec)
Measurement- 464 out of 500: 923 Mbps  (606 usec)
Measurement- 465 out of 500: 917 Mbps  (610 usec)
Measurement- 466 out of 500: 976 Mbps  (573 usec)
Measurement- 467 out of 500: 937 Mbps  (597 usec)
Measurement- 468 out of 500: 950 Mbps  (589 usec)
Measurement- 469 out of 500: 961 Mbps  (582 usec)
Measurement- 470 out of 500: 965 Mbps  (580 usec)
Measurement- 471 out of 500: 973 Mbps  (575 usec)
Measurement- 472 out of 500: 997 Mbps  (561 usec)
Measurement- 473 out of 500: 963 Mbps  (581 usec)
Measurement- 474 out of 500: 988 Mbps  (566 usec)
Measurement- 475 out of 500: 947 Mbps  (591 usec)
Measurement- 476 out of 500: 887 Mbps  (631 usec)
Measurement- 477 out of 500: 947 Mbps  (591 usec)
Measurement- 478 out of 500: 939 Mbps  (596 usec)
Measurement- 479 out of 500: 952 Mbps  (588 usec)
Measurement- 480 out of 500: 963 Mbps  (581 usec)
Measurement- 481 out of 500: 948 Mbps  (590 usec)
Measurement- 482 out of 500: 997 Mbps  (561 usec)
Measurement- 483 out of 500: 846 Mbps  (661 usec)
```

```
       Measurement- 484 out of 500: 971 Mbps  (576 usec)
       Measurement- 485 out of 500: 958 Mbps  (584 usec)
       Measurement- 486 out of 500: 966 Mbps  (579 usec)
       Measurement- 487 out of 500: 953 Mbps  (587 usec)
       Measurement- 488 out of 500: 910 Mbps  (615 usec)
       Measurement- 489 out of 500: 976 Mbps  (573 usec)
       Measurement- 490 out of 500: 932 Mbps  (600 usec)
       Measurement- 491 out of 500: 958 Mbps  (584 usec)
       Measurement- 492 out of 500: 948 Mbps  (590 usec)
       Measurement- 493 out of 500: 960 Mbps  (583 usec)
       Measurement- 494 out of 500: 985 Mbps  (568 usec)
       Measurement- 495 out of 500: 1004 Mbps  (557 usec)
       Measurement- 496 out of 500: 955 Mbps  (586 usec)
       Measurement- 497 out of 500: 952 Mbps  (588 usec)
       Measurement- 498 out of 500: 996 Mbps  (562 usec)
       Measurement- 499 out of 500: 961 Mbps  (582 usec)
       Measurement- 500 out of 500: 978 Mbps  (572 usec)

-- Local modes : In Phase II --
       * Mode: 940 Mbps to 970 Mbps  - 291 measurements
         Modal bell: 490 measurements - low : 788 Mbps - high : 1163 Mbps

--> Average Dispersion Rate (ADR) estimate: 954 Mbps

--> Possible capacity values:
        1156 Mbps to 1212 Mbps  - Figure of merit: 1725.43
        965 Mbps to 992 Mbps  - Figure of merit: 3.61


------------------------------------------------
Final capacity estimate :  1156 Mbps  to  1212 Mbps
------------------------------------------------
```

Bibliography

1. IPv4 Address Space Exhaustion Wikipedia. [Online] [Cited: February 20, 2013] http://en.wikipedia.org/wiki/IPv4_address_exhaustion.

2. Yuk-Nam Law, Man-Chiu Lai, Wee Lum Tan, Wing Cheong Lau. Empirical Performance of IPv6 vs. IPv4 under a Dual-Stack Environment. Department of Information Engineering The Chinese University of Hong Kong, Shatin, Hong Kong.ICC 2008. IEEE International Conference,Pp 5924-5925. May 2008.

3. Yi Wang ,Shaozhi Ye, Xing Li. Understanding Current IPv6 Performance: A Measurement Study. Department of Electronic Engineering, Tsinghua University,Beijing 100084, P. R. China Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC), Pp 1-2.2005.

4.Larry L. Peterson (Princeton University), Bruce S. Davie (Cisco Systems). Computer Networks, A System Approach. Fourth edition, Morgan Kaufmann Publication, Pp 30-31,70,26-31, 328-329, 376-406, 450-518.2008

5. Christian Huitema.IPv6- The New Internet Protocol. Second edition, Printice Hall, Pp 23-39,94-98.1998.

6. Lorenzo Colitti, Steinar H. Gunderson, Erik Kline, Tiziana Refice. Evaluating IPv6 Adoption in the Internet. Google, Inc.PAM'10 Proceedings of the 11th international conference on Passive and active measurement, Pp 2-3, 6-10. 2010.

7. Kenjiro Cho (Sony Inc), Matthew Luckie (CAIDA), Bradley Huffaker (CAIDA). Identifying IPv6 Network Problems in the Dual-Stack World. SIGCOMM 2004 Network Troubleshooting Workshop, Portland, Oregon,Pp. 283-288.Sep 2008.

8. IPv6 Protocol Wikipedia. [Online] [Cited: February 25, 2013] http://en.wikipedia.org/wiki/IPv6.

9. S. Deering,R.Hinden.Internet Protocol, Version 6 (IPv6) Specification. [Online] 1998. [Cited: February 28, 2013]. http://tools.ietf.org/html/rfc2460.

10. Microsoft Corporation. Introduction to IP version 6,Published.Pp 27-36.September 2003, Updated: January 2008.

11. IPv6 Security Wiki. [Online] [Cited: March 2, 2013] http://ipv6security.wikia.com/wiki/Ipv6_header.

12. D. Borman.TCP and UDP over IPv6 Jumbograms. [Online] 1997. [Cited: March 1, 2013]. http://tools.ietf.org/html/rfc2147.

13. R. Gilligan, E. Nordmark. Transition mechanism for IPv6 Hosts and Routers. [Online] 2000. [Cited: March 3,2013.] http://www.ietf.org/rfc/rfc2893.txt.

14. J. Reynolds - Assigned Numbers. [Online] 2002. [Cited: March 3,2013]. http://tools.ietf.org/html/rfc3232.

15. R S Prasad, M Murry, C. Dovrolis K. Claffy. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools (CAIDA). Network IEEE, Vol 17, Pp 1-5,7-11.Nov-Dec 2003

16. Haijin Yan, Kang Li, Scott Watterson, David Lowenthal. Improving Passive Estimation of TCP Round - Trip Times Using TCP Timestamps. Department of Computer Science, University of Georgia. IP Operations and Management, Proceedings IEEE, Pp 181-185.Oct 2004.

17. S. McCreary, K.C. Claffy.Trends in Wide Area IP Traffic patterns-A view from Ames Internet Exchange Tech. [Online] Feb.2000. [Cited: March 12,2013]. http://www.caida.org/publications/papers/2000/AIX0005/AIX0005.pdf

18. S. Floyd, T. Henderson, A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. [Online] 2004. [Cited: March 11,2013]. http://tools.ietf.org/html/rfc3782

19. M. Mathis, J. Mahdavi, S. Floyd, A. Romanow.TCP Selective Acknowledgment Options.[Online] 1996.[Cited: March 11, 2013].http://tools.ietf.org/html/rfc2018

20. Stefan Sariou, P. Krishna Gummadi, Steven D. Gribble.Sprobe: A Fast Technique for Measurement Bottleneck Bandwidth in Uncooperative Environments.University of Washington.IEEE INFOCOMM.2002.

21. K. Lai, M. Baker.Measuring Link Bandwidths Using a Deterministic model of Packet Delay.Proceedings of ACM SIGCOMM Vol 30, Pp 283-294. Sept 2000.

22. CAIDA.[Online] [Cited: July 2, 2012].http://www.caida.org/tools/taxonomy.

23. OOKLA:-Measuring and Understanding Broadband: Speed, Quality and Application.June2010. http://www.ookla.com/docs/UnderstandingBroadbandMeasurement.pdf

24. W. Richard Stevens.UNIX Network Programming, Networking APIs: Sockets and XTI,Printice Hall Publication Vol 1. Pp xv,4.1998.

25. Mark Mitchell, Jeffrey Oldham, Alex Samuel.Advanced Linux Programming, New Riders Publication, Pp 81,96,116,126.June 2001.

26. Network Socket Wikipedia.[Online] [Cited: March 22, 2013]. http://en.wikipedia.org/wiki/Network_socket.

27. Linux 2.6.7 manual - man 7 signal

28. Bani-Hani Raed M.Building Network Aware Tool to Measure Network Performance Parameters Along an IPv4/IPv6 Network Path. M.S. Thesis, Computer Science Department.University of Missouri.May 2003.

29. Dovrolis C (Georgia Tech), Ramanathan P (University of Wisconsin), Moore D (CAIDA).Packet Dispersion techniques and a capacity estimation methodology.Networking, IEEE/ACM Transactions Vol.12, Pp 963-977.Dec 2004

30. Pathrate Tutorial, Georgia Tech.[Online] [Cited: Jan 9, 2013]. http://www.cc.gatech.edu/fac/Constantinos.Dovrolis/bw-est/pathrate_tutorial.html

31. Interquartile Range Wikipedia.[Online] [Cited: Apr 10, 2013]. http://en.wikipedia.org/wiki/Interquartile_range

32. Haijin Yan, Kang Li, Scott Watterson, David Lowenthal.Improving Passive Estimation of TCP Round-Trip Times Using TCP Timestamps.Department of Computer Science, University of Georgia.IP Operations and Management.Proceedings IEEE Workshop, Pp 181-185.Oct 2004.

33. Open Networking Foundation.Software-Defined Networking: The New Norm for Networks-White paper Pp 7-8.April 2012.

34. P. Chimento, J. Ishac.- Defining Network Capacity.[Online] 2008.[Cited: September 15, 2013]. http://tools.ietf.org/html/rfc5136.

35. M. Allman, V. Paxson.- TCP Congestion Control.[Online] 2009.[Cited: October 15, 2013]. http://tools.ietf.org/html/rfc5681.

36. The GNU Operating System.[Online] [Cited: November 2, 2013]. http://www.gnu.org.

Glossary

1. **IPv6-over-IPv4 Tunnels** - It encapsulates IPv6 packets over IPv4 headers to carry them over the underlying IPv4 routing infrastructures to be delivered to the end node supporting the IPv6 protocol stack.
2. **QoS** - Quality of Service encompasses various metrics starting from offered Bandwidth, low propagation delay, packet loss, CPU utilization and packet jitter.
3. **NAT** - Network Address Translation is a way to conserve IP address space. It allows having host addresses which are not globally unique but unique within limited scope of network, like a corporate network where it resides and communicate with other hosts in the same network. The host can communicate to external world via a NAT box which has unique global address assigned to [4]. Thus many hosts within a sub network communicate using non-unique global IP addresses but one, NAT box having a globally unique address saving address spaces.
4. **MTU** - Maximum Transfer Unit is the maximum size packet that a network path can transfer without any fragmentation from source host to destination host. Deciding the MTU of a network path is one of the crucial aspects of performance measurement tool for accurate measurement.
5. **Interquartile Range** - It is measure of statistical dispersion and equal to the difference between upper and lower quartiles [3]. Quartiles of a dataset values are three points that divide the data set into four equal groups. For example, if datasets has values 2, 4, 5, 7, 8, 9, 10, 12, 15, 16, 18. Quartiles are 5, 9, 15 as Q1, Q2 and Q3 respectively and Interquartile range is Q3 - Q1 which is 15-5 = 10 for this dataset.

6. **POSIX** - Portable Operating System Interface is a family of standards specified by the IEEE for maintaining compatibility between operating systems especially for different variants of UNIX OS.
7. **MUTEX** - A mutex is a special lock that only one thread may lock at a time. If a thread locks a mutex and then a second thread also tries to lock the same mutex, the second thread is blocked, or put on hold. Only when the first thread unlocks the mutex, is the second thread unblocked to resume execution.
8. **GNU** - GNU is Unix-like computer Operating System that is free software developed by the GNU project. GNU project was launched in 1984 to develop the GNU systems that compose of free softwares. GNU is a recursive acronym for "GNU's Not Unix!" [36].